AD-E501 293

Copy 10 of 14 copies

(2)

IDA MEMORANDUM REPORT M-540

IDA AND THE TECHNICAL COOPERATION PROGRAM REAL-TIME SYSTEMS AND Ada WORKSHOP, 21-23 JUNE 1988

James Baldo

June 1988

Prepared for
STARS Joint Program Office

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 88-033875

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1988	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE IDA and the Technical Cooperation Program Real-Time Systems and Ada Workshop, 21-23 June 1988	5. FUNDING NUMBERS MDA 903 84 C 0031 A-134
6. AUTHOR(S) James Baldo	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses 1801 N. Beauregard St. Alexandria, VA 22311-1772	8. PERFORMING ORGANIZATION REPORT NUMBER IDA Memorandum Report M-540
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) STARS Joint Program Office 1400 Wilson Blvd. Arlington, VA 22209-2308	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, unlimited distribution; 2? August 1990.	12b. DISTRIBUTION CODE 2A

13. ABSTRACT *(Maximum 200 words)*

IDA Memorandum Report M-540, IDA and the Technical Cooperation Program Real-Time Systems and Ada Workshop, 21-23 June 1988, documents the results of a workshop held in support of The Technical Cooperation Program (TTCP) and the Office of the Deputy Under Secretary of Defense Research and Advance Technology (ODUSD R&AT). Funding was provided by the STARS Joint Program Office. The objectives were to (1) define requirements for using Ada in real-time systems, (2) identify and clarify known Ada real-time issues, (3) identify near-term and long-term solutions, and (4) provide assessment and recommendations for future research directions. This proceedings documents the results of each working group, summarizes the presentations, and presents the overall findings and recommendations of the workshop.

14. SUBJECT TERMS Ada Programming Language; Real-Time Systems; Timing Abstractions; Kernels; Real-Time System Architectures; Run-Time Performances; Fault Tolerance; Real-Time Scheduling Theory,	15. NUMBER OF PAGES 110
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

IDA MEMORANDUM REPORT M-540

IDA AND THE TECHNICAL COOPERATION PROGRAM REAL-TIME SYSTEMS AND Ada WORKSHOP, 21-23 JUNE 1988

James Baldo

June 1988

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
DARPA Assignment A-134

Contents

List of Tables

Part I

Executive Summary

Overview

On 21-23 June 1988, the Institute for Defense Analyses sponsored a real-time systems and Ada workshop. The workshop was held in support of The Technical Cooperation Program (TTCP) Technical Panel on Software Engineering XTP-2 and the Office of the Deputy Under Secretary of Defense Research and Advance Technology (ODUSD R&AT). The Software Technology for Adaptable, Reliable Systesm (STARS) Joint Program Office provided the funding for both the workshop and publication of this proceedings [1].

The objectives of this workshop were to:

- define requirements for using Ada in real-time systems,
- identify and clarify known Ada real-time issues,
- identify near-term and long-term solutions, and
- provide assessment and recommendations for future research directions.

The defense community was concerned that the Ada programming language had problems which were inhibiting or preventing its application to real-time systems. These concerns were the underlying basis for this workshop, and as a result, the participants spent time in identifying misperceptions and verifying valid Ada real-time issues. A number of real-time researchers were deeply concerned by the indifference of the Ada community to real-time issues that had been previously identified by them [the researchers].

Attendees of this workshop were in general agreement that the Ada standard provides many benefits for building real-time military systems. However, it should also be noted that with any standard, a careful evolutionary path in which a standard changes by subsuming an earlier standard is necessary and critical for its effectiveness.

The IDA/TTCP workshop was organized into morning and afternoon sessions. Several workshop participants were asked to give presentations during the morning sessi ; on research and implementation topics relevant to Ada real-time systems. The purpose of these presentations was to provide an exchange of technical information and stimulate discussion of real-time issues.

The afternoon sessions provided time for three working groups to discuss real-time Ada issues on a specific topic and to generate recommendations. Each group presented an overview of their session, findings, and recommendations at the end of each afternoon in plenary session. The three working group focused on issues with

[1]Prior to the IDA workshop on real-time Ada systems, two International Workshop on Real-Time Ada Issues was held in Moretonhampstead, Devon, UK, sponsored by Ada UK in cooperation with ACM SIGAda. A number of participants of these UK workshop were present and care was taken to avoid redundancy.

respect to their titles: Ada time abstractions; Ada run-time kernel; and Real-time system architecture issues.

This proceedings documents the results of each of the working groups, summarizes the presentations, and presents the overall findings and recommendations of the workshop.

Requirements for using Ada in Real-time Systems

Although Ada is still lacking in some areas for real-time systems, the workshop attendees still support the language for applications in this type environment. The discussion below describes real-time requirements that need to be supported by Ada.

The participants acknowledged that the requirement of real-time computing depended on both the temporal and logical correctness of the system. The requirements of most embedded mission critical applications have stringent deadline-driven timing constraints. These deadline-driven constraints have a major impact on the hardware and software architecture of the real-time system.

The timing requirements of real-time defense systems can be quite complex. A principle that is applied to handle logical complexity is to provide designers and implementors with powerful abstraction mechanisms. In real-time systems, however, this approach for handling time is not available. Generally low-level system parameters are varied until the timing requirements are met. Usually, this is done *ad hoc*, takes a considerable amount of time, and is extremely vulnerable to errors. In order to manage the complexity of real-time systems, it is necessary to be able to abstract the temporal and logical correctness of the system. The capability to specify a timing abstraction in Ada is necessary for implementing real-time systems.

Ada Language Real-time Issues

Ada language issues impacting real-time applications were summarized by the participants of the workshop:

1. The capability to predict and guarantee hard-deadlines and detect missed hard-deadlines is not provided in Ada. The priority inversion problem causes timing to be unpredictable.

2. The capability to define a delay within a specified time bounds is not provided in Ada. The uncertainty of the delay construct makes it use in real-time applications inappropriate.

3. The language forces values of type duration to be used as a parameter for the delay statement. In general, this is an unsafe practice since calendar time

is non-monotonic owing to physical and sociological factors (e.g. daylight savings time, leap seconds, etc.).

4. The language does not provide a sufficient concept of a clock. Since the majority of real-time systems have precise timing constraints, the language should provide standard access to a set of clocks with a range of precision and accuracy.

5. The language does not provide a mechanism to model the underlying interrupt mechanism of the target. This poses many problems since the language cannot efficiently or effectively be utilized across various types of target interrupt mechanisms.

6. The Ada rendezvous (timed entry call) is ambiguous.

7. The language mandatory support of the abort statement, conditional timed entry calls, and the rendezvous are expensive operations and exist e 'en if the feature is not used. This means that information stored by the run-time system and mechanisms to support these operations will be included in each task. [2]

Research Requirements

The workshop participants recommended the Department of Defense *dramatically* increase its support in the following research areas:

- A timing correctness analysis tool for Ada programs is needed that takes into account environment attributes.
- Real-time scheduling theory research should increased to support the predictability needed for complex systems of today and the future.
- Research should be increased to support new mechanisms used to enhance Ada run-time performance.
- Support should be given to research investigating the passing of timing information to the run-time kernel.
- Research should be funded to develop a comprehensive understanding of target architecture dependencies in relation to Ada run-time kernels that will impact the performance and timing characteristics of applications.
- Research is needed to produce methods to quantif architecture dependencies of the run-time system.

[2]For many military systems (e.g. missiles) the embedded target processors and memory, are expendable. Every effort must be made to minimize target processor resources required, thus minimizing costs.

- Research should be continued to investigate the impact that a distributed system has on runtime algorithms and communications mechanisms.
- An increase in research funding is needed to support fault-tolerance applications in distributed real-time Ada systems.

Part II

Working Group I

Chapter 1

Timing Abstraction Issues

1.1 Overview

The init' ' objectives of Working Group I, Ada Time Abstraction, were to discuss requirements for specifying time and language constructs necessary to implement timing requirements. Working Group I agreed that was essential to define a concise and terse set of basic timing requirements for real-time systems and review mechanisms that could be used in the short term to facilitate resolution of timing implementation problems.

Next, they determined if these basic requirements were supported by the Ada language. If not, the group then suggested solutions that were categorized as either short or long term. Specific recomendations for changes to the Ada language were also identified and are found at the end of this chapter.

The rationale for short term recommendations was to minimize the number of language changes. For the short term, mechanisms (possibly in combination) like the ARTEWG (Ada Runtime Environment Working Group) Run-time Standard Interface and guidelines (e.g., static tasks, non-usage of abort construct, etc.) could be used to enable the use of Ada in some real-time application domains. Long-term recommendations may require research to find possible solutions.

With respect to timing issues, the group separated the issues into those of non-predictability and overhead. The following paragraphs identify and briefly discuss each issue.

The group was concerned that the capability to predict and guarantee hard-deadlines and detect missed hard-deadlines is an essential feature of any real-time programming language. Due to the priority inversion problem in Ada, timing is not predictable.

The delay construct semantics are defined such that a task will be delayed greater than or equal to a specified time. The uncertainty of the delay construct makes it use in real-time applications inappropriate.

The current package CALENDAR is misleading and potentially dangerous. It implies that durations are computable by subtracting absolute time of day. In general this is an unsafe practice, since calendar time is non-monotonic, owing to physical and sociological factors (daylight savings time, leap seconds).

There is no alternative for computing the argument to delay for periodic scheduling without using CLOCK. Also, there is no way of finding the true elapsed time between two points without subtracting CLOCK readings (which could lead to such anomalies as negative delays). It is crucial that two types of clock recognized by Ada:

a. One type of clock which is synchronized with the normal view of calendar time (implying re-synchronization and hence non-monotonic).

b. A second type which is monotonically increasing clock for periodic calculations.

An alternative considered would be to replace CALENDAR with a lower level interface to attach various user time packages.

Efficiency of the Ada tasking model raised several questions. The abort statement and conditional timed entry calls must be supported. The mechanism(s) to support these language feature are expensive and always exist, even if the constructs are not used in the application. Information stored by the run-time system and mechanisms to support these operations will be included in each task.

The rendezvous is also an expensive operation that adds to the overhead of a task. An Ada task that did not contain a rendezvous would still be penalized with respect to performance based on the built in support for the feature.

An additional recommendation was made by the Group that Ada vendors should support the interfaces as described in the document, *A Catalog of Interface Features and Options for the Ada Runtime Environment.* The Catalog was produced by the ARTEWG Interfaces Subgroup.

1.2 Timing Abstractions Requirements

The group agreed on the first day that a definition for 'event' was needed since the term was to be used consistently throughout the basic requirements. The following definition for event was adopted by the group:

1. Informal definition of 'event'

   a. Relates to an effect in the external environment (e.g., a voltage appearing on a terminal)

   b. A point in the execution of an Ada program (e.g., an entry to a select);

   - examples are:

- (1) Interrupts
  - (a) External
  - (b) Timer
- (2) Certain statement executions
- (3) Delay expiration
- (4) Synchronization points

Working Group I derived a set of basic Ada time abstraction requirements which are listed below:

1. Need to be able to bound time.

   a. There are two requirements for bounding time:

      (1) Need to bound the elapsed time between events to confirm application requirements.

      (2) Need to bound the execution time of a section of Ada code to assist in scheduling and to detect task overrun.

2. Need to be able to define periodic events.

3. Need a mechanism to handle failures in 1 and 2.

4. Need to provide a mechanism for an application to be able to specify timing behavior (e.g., overload, starvation) or task scheduling policy.

5. Need a model of time to separate concerns of:

   a. Common Clocks (subject to adjustment)

   b. Real-time clocks (monotonic)

6. Need an absolute delay time.

7. Need for immediate asynchronous change in control flow.

8. Need to provide delay statement accuracy:

   a. Need to be able to specify the application requirement.

   b. Need to be able to determine the accuracy actually provided.

9. Need to eliminate overhead support for abort and timed entry call, since this will have an impact on asynchronous change in control flow (assume language supports this capability). Given immediate asynchronous change in control flow, it is critical for real-time applications that no overhead be paid for abort and timed entry call.

1.3 Discussion of Timing Abstractions Requirements

Group I agreed that suggestions for changes to the Ada language reference manual should be treated with caution. This agreement was based on the view that any change should be backed by evidence that the language modification is correct and needed. Prototyping of the change, which is expensive and a long term effort, would be required. Based on this proposed rationale, the group attempted to avoid language changes where possible.

The requirements, research, and language sections were based on the above rationale. It should be noted that many of the short-term solutions (based on adoption of proposed ARTEWG solutions) were selected to provide immediate solutions to current problems.

The following is a summary of the discussions that lead to previous requirements generated by Group I.

1. Bounding of time - The requirements of real-time systems to respond to an event within a prescribed time is essential. Due to changing system loads, the response time for an event may vary. For example, a real-time system is designed around a uniprocessor to service external events from a specified environment. The arrival of several events at the same time could delay the service of an event long enough to cause system failure.

   The Ada language does not provide a mechanism for the implementor to describe the system response time. Instead, the Ada code must be structured and tested for each system implementation. This results in code that is not highly portable and is very implementation dependent.

   It is also highly desirable to have a language mechanism to express the amount of time that a certain segment of code can execute. Currently, two problems exist in Ada that make this mechanism difficult to implement. Since several task can be concurrently ready to run or executing (multiprocessor system), a task in a bounded segment of code could be swapped for execution by another task or blocked by a rendezvous. Another problem is that different compilers produce different sizes of target code for the same target. Also, the same code segment has varying execution times across different targets.

   It should be noted, the timing correctness for a real-time system is of equal importance as its functional correctness. The ability to specify and verify timing is critical. However, most real-time systems are designed and implemented using ad hoc methods.

   Timing specifications would have to be added to the language to incorporate the timing mechanisms discussed. The majority of the group expressed the opinion that more research is needed in timing specifications and decided that this issue was a long-term goal.

2. Periodic events - There are many real-time applications that require cyclic service. Ada does not provide an accurate mechanism to schedule a periodic event. Although the mechanism could be implemented on top of an Ada run-time system, its widespread use should require its support in Ada.

   The majority of the group did not support a language change for specific language constructs that support periodic events. Instead, the mechanism may be realized by using services provided by the run-time system. This could be provided in a uniform manner by using the run-time interface specified by the ARTEWG's *A Catalog of Interface Features and Options for the Ada Runtime Environment.*

3. Detection of timing faults - If the timing requirements in 1 and 2. are not satisfied, an exception should be generated by the system. This is based on the knowledge that many real-time systems can have catastrophic results in the event of a system failure. The ability to detect or recover from a timing fault is essential.

4. Timing behavior and task scheduling specification - The problem of priority inversion currently inhibits the language from being used in a number of real-time application domains. Current Ada schedulers may prevent starvation; however, this does not work well in real-time systems with respect to meeting deadlines.

   Scheduling algorithms, such as the stabilized rate monotonic algorithm, which can meet system deadlines up to 70 percent processor utilization, cannot be used in Ada due to the priority inversion problem. To correct the priority inversion problem in Ada implies a change in the language.

   A potential solution to this problem was presented in the Goodenough and Sha paper at the 2nd International Workshop on Real-Time Ada Issues, May 1988, entitled 'The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks.' Their approach uses the priority inheritance protocol and the priority ceiling protocol. The combination of these two protocols minimizes the blocking of a high priority task by a lower priority task to at most once.

   An implementation of the above system currently is being tested at the Software Engineering Institute (SEI). The majority of the working group favored this approach as it could provide a short term solution for the application of Ada to several real-time domains.

5. Clocks - The need for a real-time clock is essential for most real-time systems. Although access to a real-time clock can be implemented in an ad hoc way for each system designed, it would be more useful to provide a standard interface. To avoid a language change, the group decided that ARTEWG Run-time Interface could provide the appropriate access.

6. Absolute delay - The language does not support an accurate delay, it only guarantees a delay equal or greater to the specified time. Since the ARTEWG Run-time Interface provides a bounded delay, the group suggested that this approach could be used as a short term solution.

7. Asynchronous control - In real-time systems it is frequently necessary for a task to asynchronously transfer control to another task. The Ada language provides the abort and rendezvous mechanisms for transfer of control. However, these mechanisms are inappropriate for asynchronous control. For example, if an event occurs that requires a task to stop executing, it may be unacceptable to wait for the task to reach the next rendezvous before halting. Also, these mechanisms unfortunately are costly and, therefore, difficult to use in real-time systems.

   The abort is expensive and requires all the supporting mechanisms even if the implementation does not use the construct. Due to the overhead associated with the abort, it is unlikely that it can provide the response time needed in most real-time applications.

   The rendezvous cannot guarantee the immediacy required by many real-time systems since the task would have to poll, which is extremely inefficient. Several suggestions with respect to mechanisms and language constructs have recently appeared in the literature on asynchronous control in Ada. The group agreed that further research is needed before a recommendation can be made.

8. Delay statement - The group agreed that the present semantics of the delay construct in general was not very useful for real-time applications. It was decided that the group would define the necessary attributes that the delay construct should support for real-time applications.

   The specification of the time delay can have a wide range of magnitudes (e.g., days to micro seconds). The delay statement must be able to support a range of timing that is at least representative of a broad spectrum of defense systems. Also, a user must be able to determine (or possibly specify) the accuracy of the delay (when will it occur).

   Since the above recommendations would require language changes, the majority of the group agreed that the ARTEWG Run-time Interface, bounded-delays, would be an acceptable short-term solution.

9. Abort and timed entry call task overhead - The abort and time entry call mechanisms are required and cannot be separated from tasks. Ada real-time implementations should not have to use these constructs. The group suggested that a 'light-weight task' be made available to implementors that does have the abort and time entry call overhead. A short-term solution would be to supply a task of this type through the run-time interface (ARTEWG Run-time Interface). The principle is to move high cost tasking functions into the application domain.

1.4 Research Issues

Working Group I discussed several areas of research that are needed to provide solutions, improved performance, and correctness with respect to the basic timing requirements that the group generated. Following is a brief description of the suggested research areas.

The problems of specifying and implementing the specifications of time is not trival. Continuing research in the area of timing specifications and verification is essential. A program timing correctness analysis tool that can take into account environment attributes (e.g., such as processor instruction and context switch times) should be a high priority research item

Real-time scheduling theory research is necessary to support the predictability needed for complex systems of today and the future. Real-time testbeds will be necessary to demonstrate that the scheduling theory is adequate for application to defense systems.

Real-time distributed scheduling theory research will be essential for the majority of the next generation of real-time systems. Timing issues with respect to distributed Ada systems are quite different from those based on a single processor system.

1.5 Recommendations for Language Changes

There was a general agreement about what should be included as recommendations for language changes to the Ada 9X process. The group proposed two recommendations for consideration:

- Consistent treatment of priorities in all resource allocation decisions.
- Development of a mechanism for an application run-time interface to provide a scheduler (similar to 'light-weight tasking').

Part III

Working Group II

Chapter 2

Real-time System Architectures

2.1 Overview

The initial objectives of Working Group II were to identify and understand the issues of Ada and system architectures for real-time systems (RTS). The group defined two areas of consideration:

1. considerations for designing solutions for RTS; and
2. considerations for implementation of those designs including:
    a. the Ada programming language,
    b. the evolving nat ,e of RTS, and
    c. the current execution environments.

The group decided that they would restrict their issues, requirements, and recommendations to specific classes of systems. Systems can be categorized according to three characteristics: concurrent, distributed, and real-time. These characteristics are then used to qualify the following classes of systems:

1. sequential,
2. concurrent,
3. real-time concurrent,
4. distributed, and
5. real-time distributed.

Working Group II focused on real-time concurrent and real-time distributed systems. Real-time systems are generally large and complex. In considering architectural issues, choices that tend not to support large address spaces, concurrent operations (i.e., low process-switching overhead), or priority-based scheduling are likely to be less cost-effective for development of these systems than choices offering relatively more support.

The group discussed the impact of system architecture on performance. It was agreed that acceptable performance is likely to result only when a given architecture does not extract an undue penalty for representing concurrency and interprocess communications (IPC) according to the model that is assumed by the software system.

Ada provides a default process model (the Ada tasking model) and a default interprocess communications model (i.e., synchronization and data transmission by rendezvous). If a program assumes these defaults and the architecture penalizes their use, then unacceptable performance will result. This has been observed in a current standard military processor. This processor's architecture extracts a heavy penalty for task context switching (large overhead for stack copying).

The group agreed that before Ada can be applied to real-time distributed systems, issues such as communication between processes, the use and protection of shared variables, and timing must be resolved. There was also concern that Ada may not adequately support the process of partitioning a real-time distributed system across a set of processors.

The group derived four forms of architectural improvements that could be applied to real-time concurrent and distributed systems:

1. Logic improvements - Such improvements (e.g., smaller gate delays) lead to improved performance across the entire range of programming paradigms.

2. Concurrent system specific - Such improvements are ones that improve performance over the entire class of programs using a concurrent control paradigm. For example, any concurrent application benefits from the addition of register resources to reduce switching times as long as the combined language support environment manages these resources in a reasonable way.

3. Application specific - such improvements improve performance over a limited a range of applications. For example, bit flippers for Fast Fourier Tranfers or multiple Arithmetic Logical Units for coordinate transformers use hardware in a way the accelerates a single class of applications. There is some question as to how such hardware is utilized by the Ada compiler in a modular, portable way.

4. Real-time Ada specific - this type of improvement accelerates those real-time programs that are written Ada. Clearly, there is some overlap between this category and the others. For example, extra register sets as a real-time-Ada-specific improvement as well as a concurrent-system-specific improvement. Tag matching and bounds comparing facilities are similar.

The group also identified that the standardization of Instruction Set Architectures (ISA) has had a negative effect on the use of Ada in real-time systems. The majority of the group agreed that standardized ISAs have not led to standardized hardware, and 32-bit processors seem to be more cost-effective than 16-bit processors.

With respect to the environment of the run-time system, two important points surfaced regarding the low-level environment supported by the processor. First, careful attention should be paid to the protection of system resources by supervisory or privileged states. In modern practice (e.g., High Level Languages) and real-time systems, the concept of self virtualizing architectures does not seem as important. Second, considerably more attention needs to be paid to try to match the interrupt structure of the machine to the scheduling model of the task dispatcher in the run-time system.

2.2 Ada Issues with Respect to Architectures

The working group was concerned that since the majority of real-time systems have precise timing constraints, the language should provide standard access to a set of clocks with a range of precision and accuracy. It was agreed that Ada provides an insufficient concept of a clock.

Due to target architecture dependencies, Ada run-time kernels will impact the performance and timing characteristics of applications. A better understanding of these dependencies is needed. Methods to quantify architecture dependencies of the run-time system are also needed.

The Ada language provides no mechanism to model the underlying interrupt mechanism of the target. This poses many problems since the language cannot efficiently or effectively be utilized across various types of target interrupt mechanisms. The group suggested that there should at least be a mechanism to disable interrupts as needed.

For distributed real-time systems, the issues are complex and not well understood. It was acknowledged that there was a need to insert a mechanism for education for designers and implementors involved with distributed real-time systems.

The issue of how the run-time algorithms and communications change when the system is distributed was discussed briefly. The group agreed that control of the algorithm over the connections is desired. This control may be either implicit or explicit. Two Ada issues were brought up as well: What support is found in Ada for the timing of rendezvous? How can the distributed processes degrade gracefully?

Currently, the Ada rendezvous (timed entry call) is ambiguous. This is particularly a problem for distributed processing. The major problem with the Ada rendezvous in a real-time distributed system is the difficulty in predicting time for its completion. However, this does not preclude using a multiprogram version.

The conclusion of this part of the discussion was that Ada imposes stringent constraints cn the hardware selection/design. The group agreed that distributed support is important, but then split over the issue of whether the support for distribution should be provided in Ada (it currently lacks adequate support) or should be handled by the run-time environment. Some members of the group felt that the notion of distribution should be understood at the Ada level. That is, within the Ada model distributed communication must be facilitated.

However, before it will be feasible to do distributed real-time processing, some issues will have to be resolved. Specifically, an efficient means of communication between processes, thc use and protection of shared variables and other information, and timing are areas of concern.

The group briefly discussed several mechanisms to handle processor failure and process migration or termination and propagation of status. A number of interesting questions were raised and the group concluded that more discussion is needed.

On the subject of performance of distributed Ada real-time systems, the group agreed that the major factor influencing the cost is the support required to handle the Ada semantics (for instance, for a context switch).

The group concluded that a rendezvous across a network using the ISO (International Standards Organization) protocols is extremely inefficient.

The notion of distribution imposes some assumptions on what is considered acceptable performance. The question of what happens increase the distance of distribution is increased and mode of interconnect is raised. Should a typical distribution and a very distant distribution look the same to an Ada rendezvous? Are the semantics the same? Does the application need to know the nature of the distribution? If so maybe the entire problem domain should be described in Ada?

The issue for discussion became *what is the scope of the Ada standard*? Is it necessary, or desirable, to replacc he Ada tasking model for some situations? Given the example of a distributed system and a rendezvous between a ground station and a satellite, the delay is very long. Is this an Ada problem or is it a problem in the system design? Would it be advantageous not to use a rendezvous? Maybe the rendezvous is appropriate for only certain applications. The members of the group disagreed on this point. Some argued that you should only use the features of the language where it it appropriate, while others argued that since the rendezvous was included in the language, it was the appropriate solution to all problems. The group also disagreed on whether using some other mechanism for communication would be a violation of the Ada language. Would the use of some other communication mechanism decrease portability and maintainability?

To understand what an appropriate architecture for Ada real-time systems might be, it is important to understand the impact of Ada on an ISA. That is, the entire system must be considered. Two potential areas of a language that may impact the ISA significantly are scheduling and exception generation.

Addressing the language specific architecture issues, what is specific or unique to Ada? Since the features of Ada may impact the nature of the architecture, these features must be considered. Members of the group acknowledged that there is probably nothing in Ada that does not appear in a combination of other languages. So what are the characteristics of Ada programs? The fundamental characteristics of Ada programs include for example, large size and complex implementation.

Given these characteristics, the group agreed that it is probably insufficient to look to a uniprocessor system for support for the realtime, Ada systems of the future. A multiprocessor architecture would be required. However, each processor in the configuration may be optimized. The discussion turned to the question of scheduling optimization. At what level of granularity (task, procedure or Ada program) would it be desirable or feasible to map to the processors? There was no general agreement on this point.

2.3 Recommendations

Working Group II provided the following recommendations:

1. System issues should not be solved with just software. That is, do not push system design decisions off on the language.
2. Current 16-bit hardware use should be scrutinized in DoD applications, as it is unsuitable for large concurrent Ada applications. These 16-bit architectures could be used, however, at great effort and cost. There is a need to re-evaluate the way hardware is procured and standardized.
3. Standard access to clocks supplying a range of precision and accuracy should be provided.
4. Hardware support for the Ada rendezvous, particularly for distributed systems, should be provided.
5. Architectual features suggested below to improve Ada performance should be considered in DoD applications.
   - Internal registers to be managed by the ISA and
   - Large register sets, hidden or exposed to the ISA depending on the application.

Recommendation one generated an interesting question for the group to consider. Does the language drive the usage? Or does the usage drive the language? Everyone agreed that the language should support the usage, but that the language strongly influences what and how you do things.

The group expressed concerns about recommendation two for real-time distributed systems, with respect to support of timing and performance requirements of these systems. Although many systems may be required to used these 16-bit processors in a distributed application, it is not clear how Ada or any high level language can efficiently support real-time distributed applications with these architectures in any domain.

Part IV

Working Group III

cation domains and built to a set of standard interfaces, would greatly reduce cost and time of reconfigurable Ada run-time kernels.

It was recognized that architecture support for the Ada run-time kernel could greatly enhance performance. The working group indicated that there was a need to exploit possible optimizations.

With increased interest in the use of standards by the defense community, the impact of such standards on the Ada run-time kernel needs to be examined. Acceptance of these standards for use in defense systems can greatly influence the design and effect the performance of the run-time kernel. The group would like to see liaison channels established between standards organizations that are responsible for standards such as communication protocols and distributed data bases.

The group was concerned that the real-time research community was failing to produce input into the Ada insertion and technology process. It was suggested that organizations such as the TTCP and the SIGAda ARTEWG play an active role in these processes. Also, prototyping potential language changes being considered by the Ada 9X was strongly recommended.

Education is a major issue. There is a need to influence the university community to actively pursue a number of education issues related to real-time systems. Although, many current real-time systems have been successfully designed and implemented by ad hoc methods that have utilized techniques from existing research areas, [2] a scientific foundation is needed to support real-time systems. This requires more research and researchers that are uniquely devoted to real-time systems.

Universities need to provide students with skills (course work and laboratory work) that give them a better understanding and capabilities of the issues and engineering principles of designing and implementing real-time systems. It was suggested that Ada lab materials, such as compilers, run-time systems, debuggers, and actual systems for experiments, be made available.

3.2 Research Issues

Working Group III concluded that the current set of criteria and resulting benchmarks and test suites used to evaluate run-time systems is inadequate. An evaluation (criteria including test suites) is needed that includes a view of the applications requirements. An investigation to extend the current set of criteria or research to provide new criteria is required.

The need to convey timing information to the run-time systems was also discussed, since there are number of possible approaches. A study is needed to identify a reasonable solution for real-time Ada systems.

[2]For more detail on this issue, see the report by John A. Stankovic: *Real-Time Computing Systems: The Next Generation*, Tech. Report TR-88-06, COINS, Dept., Univ. of Massachusetts, Jan. 1988

Optimizing or improving run-time performance can be accomplished by changing several environment conditions. Such as stylized use of the Ada language or customizing run-time kernels for a specific target environment. Working Group III has determined that more research is needed to investigate factors and new mechanisms that can be used to enhance run-time performance.

3.3 Recommendations

Working Group III provided the following recommendations:

1. Give guidance in applying Ada to significant aspects of current real-time applications (guidebook for using Ada in real-time applications). This should be accomplished by The Software Engineering Institute in its real-time Ada activities.

2. Include the development of real-time systems in Ada trainingthrough courses and subsidies for real-time Ada lab materials. This should be accomplished within a framework of software engineering and real-time system development.

3. Provide operating systems researchers with challenging real-time problems and greater research emphasis on dead-line driven distributed systems in Ada.

4. Direct real-time application experts to actively contribute and participate in the Ada 9X process. Of particular interest are dead-line driven distributed systems for validation of Ada 9X design. Make sure that the experts are representative of the international Ada community and involved with real-time mission-critical embedded military systems (TTCP could help facilitate this action).

5. Prototype changes that are being considered by the Ada 9X group.

6. Utilize university research to solve issues and develop new technology for run-time systems, which can be provided to vendors in reducing the risks in building and improving the capabilities of their products.

Part V

Summary of Workshop Presentations

Chapter 4

Presentations

4.1 The Fundamental Challenges of Deadline-Driven Computing

Dr. Andre van Tilborg from the Office of Naval Research (ONR) presented an overview of issues challenging real-time systems designers and implementors. The major issue that is causing the most difficulty with real-time systems is complexity. This complexity is a manifestation of increasing functional and timing requirements of present and future defense systems.

The realization of any real-time system depends on the correctness of the logical result of the computations and the time at which the results are produced. The difficulty in attaining the above correctness is exacerbated by the real-time system environment. In general these systems do not permit human interaction and must respond to events on a time scale too short for interaction. Overall, real-time computer systems must be predictable.

The current technology available to design and implement real-time systems is limited. Most systems are realized in an ad hoc manner. Real-time systems are very difficult to understand, analyze, and modify, due to a lack of any scientific theory to cope with real-time issues.

The introduction of time greatly increases the difficulty of developing a suitable theory. Typically, timing requirements are satisfied by building a prototype in the lab:

1. trying different priority assignments,
2. altering the binding between code segments and time slots in cyclical executives, and
3. rewriting segments of code.

It should be noted that timing variances from prototypes to actual systems can cause severe problems in development. This same problem is also evident in system modification (minor changes to the system can cause major impacts in timing).

The author summarized the current shortfalls in developing real-time systems:

- no coherent theoretical scientific model to support real-time system development,
- no ability to write testable requirements and designs involving time and reliability constraints,
- no common benchmarks or synthetic workloads,
- very limited understanding of resource management algorithms involving real-time constraints,
- very limited programming language constructs and run- time systems for real-time applications,
- very limited understanding of real-time constraints in distributed systems and communications, and
- very limited techniques for performance prediction and non-invasive instrumentation of real-time systems.

The ONR is funding a five-year initiative to establish the foundations of a science of real-time computing and derive integrated real-time computing systems methodology that combines theory with software tools. Its research agenda is focused on providing an approach to building real-time systems based on the timing and reliability behaviors of the system.

To accomplish these goals, the program has been structured to investigate and establish valid approaches for comprehensive testable formalisms for the representation and manipulation of time in specification, design, implementation, and testing of real-time computing systems. Another thrust of the initiative is to develop robust real-time scheduling algorithms with predictable/guaranteed performance, and to devise a real-time computing systems theory for distributed and parallel computing systems.

The initiative will begin in fiscal year 1989, with first year efforts concentrated on the start up. A schedule and milestones are listed at the end of this summary. ONR is confident that it is feasible to devise a strong theory of real-time scheduling for uniprocessor systems within the , .xt few years. However, scheduling theory for parallel and distributed systems is much more difficult and will not be available in the short-term.

4.1.1 ONR Real-time Initiative Major Milestones

- FY84-FY88

- Core contracts in codes 1133 and 1211 establish basis for Accelerated Research Initiative.

- FY89
  - Establish real-time computing research efforts and laboratory;
  - Start basic efforts in specification and scheduling;
  - Bring together labs, syscoms, basic researchers to achieve closure;
  - Complete formal investigation of Navy current and projected real-time requirements and define experiments.

- FY90
  - Transition preliminary theoretical results to real-time lab;
  - Complete synthetic workloads;
  - Define canonical problems;
  - Complete solid theory for non-stochastic task arrivals in uniprocessors; and
  - Complete design of prototype scheduling and specification software tools.

4.2 Real-Time Avionics Software Requirements High-Level Overview

Douglass Locke of IBM Systems Integration Division presented an overview of a typical avionics system and issues of using Ada in avionics. A diagram was shown illustrating a set of typical avionic subsystems:

- navigation
- mission planning
- communications control
- weapon deployment
- sensor processing
- pilot/crew interface
- ballistics computation
- image processing
- speech recognition/synthesis.

Each of the above subsystems vary in complexity that is unique to a specific plane and mission.

The requirements of typical real-time avionics response times were listed: navigation - 20hz; display functions - 100ms ; and sensor control - 10hz to 1Khz. These response times set the deadlines of the system. The ability to design a system that can adequately handle these response times as well as meet weight, power, and heat requirements of the system is critical.

Typical avionics software requirements include the following:

- Software correctness must include response time.
- Software reliability is frequently critical to safety.
- Software response must be predictable.
- Software for most avionics systems is designed to handle periodic events.

Avionics application software must execute in environments that provide overall support for the above requirements.

It is essential that services provided by the underlying system that the application uses must have low bounded response times. For instance processor scheduling and memory allocation and deallocation must be very efficient. Therefore the design of application software is dependent upon the performance of these underlying support mechanisms.

Since the application software must be predictable in order to meet system deadlines, the underlying support services used by the application must also be predictable. Therefore services such as file systems and paged storage must be avoided.

There are two typical overall design strategies used for avionics software: timeline sequence and fixed priority multi-tasking (with or without preemption). A timeline sequence can be briefly described as dividing a time domain into a repeating series of time slots. The execution time of an application can be partitioned into slots. High rate function get multiple slots and each function must finish within its slot. The resulting scheduling sequence is completely deterministic.

Fixed priority multi-tasking gives each task a fixed priority (e.g., may be modified for mode changes). The task with the highest priority is always executed. If preemption is not used, then CPU utilization by each task is limited to protect latency requirements for high priority tasks.

Ada is currently being used in Avionics applications. However, several restrictions are placed on its use. Since the above designs avoid synchronization, Ada tasking is not used (rendezvous not needed). Ada access types are not used due to overhead associated with elaboration, allocation and deallocation of objects. The overall software composition consist of Ada procedures that are scheduled by an avionics specific executive.

Using Ada in avionics applications raises several issues. The Ada rendezvous is the only means of communication between tasks (shared variables can be used, however, but are not recommended). Communications between Ada tasks using the rendezvous causes a timing problem with tasks that have hard deadlines. As the rendezvous is synchronous, many real-time applications need an asynchronous mechanism of communication.

The user lacks control over the time domain in Ada. For, example the delay statement cannot guarantee a response to an event within a specified time bounds.

Distributing an Ada program across several processors still poses a number of problems. Communication is the most obvious problem. Distributed system can provide redundancy that can be used to make a system fault-tolerant. However, it is not clear how this redundancy can be utilized via mechanisms within the Ada language.

4.3 Real-Time Scheduling Theory and Ada

Dr. Lui Sha of the Software Engineering Institute (SEI) gave a presentation on real-time scheduling theory and its implications to the Ada tasking model. The scheduling theory reported by Dr. Sha was sponsored by ONR. Recognizing the importance of putting the engineering of large scale real-time system on a firm scientific foundation, ONR recently launched a five-year *Real-time Systems Initiative* to rally computer scientists to build a scientific foundation for distributed real-time systems. The objective of Dr. Sha's talk was to illustrate the importance of using an analytical approach for the design and implementation of real-time system in Ada.

Traditionally, many real-time systems use cyclical executives to schedule concurrent threads of execution. Under this approach, a programmer lays out the execution timeline by hand to serialize the execution of critical sections and to meet task deadlines. While such an approach is adequate for simple systems, it quickly becomes unmanageable for large systems. It is a painful process to iteratively divide high-level language code so the compiled *machine code* segments can be fitted into time slots of a cyclical executive and that critical sections of different tasks do not interleave. Forcing programmers to schedule tasks by fitting machine code slices on a timeline is no better than the outdated approach of managing memory by manual memory overlay. Such an approach often destroys program structure and results in real-time programs that are difficult to understand and maintain.

The Ada tasking model represents a fundamental departure from the cyclical executive model. Indeed, the dynamic preemption of tasks at runtime generates non-deterministic timelines that are at odds with the very idea of the fixed execution timeline required by a cyclical executive. From the viewpoint of real-time scheduling theory, an Ada task represents a concurrent unit for scheduling. As long as the real-time scheduling algorithms are supported by the Ada runtime and the resource utilization bounds on CPU, I/O drivers, and communication media are observed, the timing constraints will be guaranteed. Even if there is a transient overload, a fixed subset of *critical* tasks can still meet their deadlines as long as they are schedulable by themselves. In other words, the integration of Ada tasking with analytical scheduling algorithms allows programmers to meet timing constraints by managing resource requirements and relative task importance. This will make Ada tasking truly useful for real-time applications while also making real-time systems easier to develop and maintain.

Although Ada tasks fit well with the theory at the conceptual level, Ada and the theory differ on the rules for determining when a task is eligible to run and its execution priority. The Ada language rules are inconsistent with respect to the enforce of a prioritized scheduling discipline:

- selective wait: priorities can be ignored;
- entry queue: FIFO, priority must be ignored;

- called task priority: only increased in rendezvous;
- hardware interrupts: always highest priority; and
- CPU allocation: priorities strictly observed.

Such inconsistency may lead to serious problems. For example, if a high priority task calls a lower priority task that is in rendezvous with another low priority task, the rendezvous continues at the priority of the task being served instead of being increased because a high priority task is waiting. Under these circumstances, the high priority task can be blocked as long as there are medium priority jobs able to run.

But there are a variety of workarounds. The most general solution within the constraints of the language is simply to not use pragma PRIORITY at all. If all tasks in a system have no assigned priority, then the scheduler is free to use any convenient algorithm for deciding which eligible task to run. An implementation-dependent pragma could be used to give "scheduling priorities" to tasks, i.e., indications of scheduling importance that would be used in accordance with analytic scheduling algorithms. This approach would even allow "priorities" to be changed dynamically by the programmer, since such changes only affect the scheduling of tasks that, in a legalistic sense, have no Ada priorities at all. The only problem with this approach is that tasks are still queued in FIFO order rather than by priority. However, this problem can often be solved by using a coding style that prevents queues from having more than one task, making the FIFO issue irrelevant. Of course, telling programmers to assign "scheduling priorities" to tasks but not to use pragma PRIORITY, and being careful to avoid queueing tasks surely says we are fighting the language rather than taking advantage of it.

While the Ada tasking model can and should be improved, the workarounds permit the implementation of analytical scheduling algorithm within the existing framework of Ada rules. This subject is discussed in detail in a SEI technical report, *Real-Time Scheduling Theory and Ada* by Lui Sha and John Goodenough. Given that the analytical approach appears to offer significant improvement in Ada based real-time software engineering, the Real-Time Scheduling in Ada project at SEI cooperates with industry and DoD agencies to perform a series of case studies, which include:

- missile application (NWC),
- avionic application (IBM SID (Owego) and NWC), and
- submarine application (IBM SID (Manassas) and NUSC).

These case studies are not being performed just by SEI. In general, the project's approach is to have potential application developers and DoD agencies to develop the case studies with SEI's assistance. The goal of the studies is to show that the analytical approach has the following capabilities:

- can be used in a realistic way;
- does not significantly increase the development risk of initial projects that choose to use the approach;
- has a positive payoff during the design and development stage of a project; (it isn't enough to argue that the main benefits are achieved in the maintenance phase);
- minimizes the amount of retraining needed to use the approach effectively;
- is supported by compiler vendors; and
- is supported with tools that make it easier to apply the theory in practice.

4.4 Real-time Distributed Ada

Anthony Garagaro of Computer Sciences Corporation substituted for Dr. Richard Volz of Texas A&M University in giving a presentation on Real-time Distributed Ada. This presentation introduced language issues in distributing execution of Ada programs and identify areas where research and development are needed.

Recently there has been increased interest in distributed program execution in the Ada community. A large number of the participants at last meeting of the International Workshop on Real-Time Ada Issues agreed that distributed program execution is an important issue. This was in contrast to the first meeting in 1987.

The next generation of real-time systems will be more complex due to their distributed nature. In designing and implementing software in Ada for a distributed systems, a general understanding of how the language supports distributed programs is useful. Distributed systems have additional characteristics in which the language needs to support a mechanism for expression. For instance, knowledge of distribution characteristics (access time) is important to algorithm development.

There should be one language definition to which both distributed and non-distributed systems adhere. It should provide whatever is necessary to express well defined distributed execution semantics.

If changes are needed to accommodate distribution, they should be made to the language definition. The rationale for making a change must be applied consistently across the entire language.

Modifying the language definition for distributed execution should not be used as an excuse to make modifications to the language not required by distributed execution. The language should be modified as little as possible.

It is important to develop a precise and well defined set of terminology that is used to discuss the distribution of Ada. For example, the notion of shared variables is not as clear as one might think because even variables inside a package body may be shared across machines, depending upon what units of the language are distributable.

The last section of this summary defines a set of terms used in descriptions and discussions of distributed Ada. It was emphasize during the presentation that a set of definitions and terms needs to be accepted amongst researchers, designers, and implementors of distributed real-time systems. A single interpretation of the following terms is needed to enable precise exchange of information and discussion.

1. Distributed execution: More than a single processor executes the code of a single program.

2. Global (shared) variables: Variables that appear in the specification of a package that is referenced from code executing on more than one processor.

3. External global (shared) variables: Same as global variables.

4. Internal global (shared) variables: Variables withi: . subprogram, task or package body that, because of distribution of code are referenced from more than one processor.

5. Global (external state (of a package): The set of variables that appear in the specification of a package.

6. Internal State: The set of variables declared within a package, task or subprogram body.

7. Distribution of a program: The process of making the following two decisions:

   a. which segments of code execute on which processors, and

   b. which code and data segments are stored on which memories.

8. Unit of distribution: One or more of the following:

   a. A segment of code that may be caused to execute on a single processor.

   b. A portion of system state (internal or external) that may be caused to reside on one or more memories in the system, or

   c. data type definitions that may be utilized by code executing on more than one processor.

Several definitions for virtual nodes have been adopted, however, it was shown that the concept of virtual node has a board spectrum of interpretations. The following question was presented:

> Is the notion of a virtual node an essential language construct, or is it a useful design methodology?

The notion of a virtual node is based upon a notion of physical nodes, which is directly related to the architecture of the system. There was no attempt to define a physical node.

It was observed that present concepts of virtual node appears to cloud several separate concepts:

1. the need to place restrictions on the language related to distribution,

2. the need to organize a distributed problem, and

3. a desire to further structure programs within the language.

More work is needed on defining the definition and concept of a virtual node.

4.5 Ada Runtime Issues for Real-time Systems

Dr. Ted Baker of Florida State University gave a presentation on Ada Runtime Issues for Real-Time Systems (RTS). The major problem that has been identified foı Ada runtime systems is that the Ada RTS intrudes into traditional operating systems (OS) and executive domains. In the past, an implementor may code the executive or use existing OS calls to gain a finer degree of control over the system (e.g., timing). So by intruding into the OS and executive domains, Ada has preempted the implementor from controlling issues critical to the success of applications.

There are four major issues that face designers of Ada run-time kernel today. The first is meeting the timing constraints of applications. The second, fault recovery and reconfiguration pose, problems due to Ada's inability to support dynamic binding, resource *reallocation*, and program control-flow modification.

The third issue is distributed Ada systems. Several groups are currently researching distributed Ada issues. Some of these issues are the same as those considered for fault recovery and reconfiguration.

The fourth issue is interrupt management and I/O. Control over interrupts within the Ada model have create a number of problems in applications. The same holds for I/O, since it is also highly depended on interrupts.

In general, implementors want more contol over the Ada RTS and more functionality. The current Ada minimal RTS is inadequate for real-time systems.

In an attempt to solve the above problems, four approaches were discussed. At present there is a need for a requirements language that can capture real-time requirements. The Ada language has a number of short comings with respect to implementing real-time systems, for example, expressing timing constraints. Language modifications and additions would be needed. The need for an expert auto-programmer to assist in the implementation of complex real-time systems (e.g., selecting the most efficient design approach for a specific target) could be used to increase system reliability and decrease cost and time.

The above approach is presently beyond the state of the art and would require a long term research plan and possibly some major changes and additions to the language. A second approach would be to utilize compiler options such as pragmas, the linking to different Ada RTS, or the setting of attributes within the RTS. Many Ada compiler vendors offer these capabilities today.

The problem with this approach is that the real-time application domains is broad and it constantly evolving. Solutions (options available today) that provide a solution for a current system will probably be inadequate tomorrow.

A third approach was the user replacing and supplementing RTS components. Two methods were proposed for this approach. The first was to have the user customize the RTS. This provides the user with the capability of changing the implementation of a component within the RTS which has been identified as inadequate for the application. It also enable the user to provide additional functionality that does not exist with the current RTS.

The second method is to provide a clean and stable interface between the code that the compiler generates and the RTS. This work is currently being done with the SIGAda subgroup, Ada Runtime Environment Working Group (ARTEWG) The interface is called the *Model Runtime System Interface* (MRTSI). This would provide the user with the capability of selecting from a set of RTSs, the one that is most suitable for the application.

The fourth approach is to use a RTS interface and allow the user to modify or provide a set of internal components for the RTS. For example, an application may require a specific scheduler (not in violation of the Ada language) that has yet to be implemented by any vendor. Other internal components that a user may want to supply are real-time clocks, storage allocation, or interrupt handlers.

The notion of having all executive functions written in Ada was discussed. This concept provides a method of layering in which the Ada environment would be organized in layers from the bare machine to the application. In order to implement the entire environment in Ada, the language would need to be extended. These extensions will add a few more attributes, representation specifications, and mechanisms to get around strong typing of data.

These extensions are necessary for bootstrapping layers onto the bare machine. There is a need for bootstrapping in the design and startup of the Ada environment. In startup, the system must be loaded and initialize. To account for the evolution of the environment, the system must be designed to enable the bootstrapping of additional functionality.

In general the system starts with a bare machine. The environment must be bootstrapped on this machine. This provides minimum support and it should be noted that this Ada code executes without an Ada run-time system. On top of this layer will be layers that support exceptions, tasking, dynamic storage, fault recovery, distribution, etc.

The following list summarizes the advantages of layering:

- controls complexity,
- allows for system evolution, and
- address the development and start-up processes (bootstrapping).

4.6 Some Issues in Processor Selection for Embedded Systems

Dr. Joseph Linn of the Institute for Defense Analyses gave a presentation that reviews issues in processor selection for an embedded system. The summary below gives a description of the main points made during the presentations.

There several important difference between real-time military embedded application and commercial applications. For military embedded applications the design of the system must account for worst case optimization. This is essential since real-time hard deadlines that are missed in these systems have the potential for catastrophic results.

Real-time military systems usually have strict constraints on system level environments. Constraints on power and size for example, have major impacts and limitations on design.

Another difference between the real-time military applications and commercial applications is the system lifetime. Most military systems are expensive to replace and can also be executing in a mission critical application, which inhibits complete system replacement. Even minor modification to these operational systems may be difficult.

In selecting chips for commercial and military systems, differences between the availability of application code and code compatibility strongly influence selection. There are a large number of commercial application software that can adequately meet system requirements. However, military application software is less abundant and usually must be designed and implemented.

Commercial vendors place large emphasis on object code compatibility with existing products. The DoD places emphasis on source code compatibility.

Appropriate standardization is very effective for reducing costs. Conversely, inappropriate standardization greatly increases cost and could prevent the meeting of requirements. Standardization of application-specific interfaces is essential for competitive procurement.

Two argument were made against ISA standardization:

- ISA standardization does not standardize hardware due to the fact that many boards in a system are unique due to strict environmental constraints. Further, logistics support is not decreased since field replacement is normally the board rather than the chip set.
- The DoD has already taken the position that software notation is to be standardize at the level of the Ada programming language. Thus, low-level standardization is unnecessary.

Architecture is related to performance only by knowledge of the application. The components of the performance equation are not known until the application is known. Further, if the application is known, application-specific figures of merit may also be kno n that are architecture-independent.

It is essential that the appropriate compiler technology be available. If not, architecture enhancements for the applicat on will not be realized.

Three current and popular architectural approaches were reviewed: lean-cycle, Reduced Instruction Set Computer (RISC), and Application Specific Instruction Set Computer (ASISC).

The lean-cycle approach selects the basic cycle of the machine to be the leanest possible that is consistent with dispatching a useful instruction on every cycle. The fundamental concept of the lean-cycle approach is that application programs are implemented in a high-level language. Therefore, hardware capabilities that are not available to the compiler are not available to the application for performance enhancements. This results in all lean-cycle architectures being high-level language directed.

The RISC approach is summarized by the following rules:

- Instructions should be few in number, simple to decode, and of fixed size to avoid decoding bottlenecks.
- Only load and store instructions should reference memory (ALU instructions should not).
- When available hardware is limited, the best performance is obtained when the utilization of non-register hardware is maximized.

Currently, there is evidence that indicates that each of these rules is being obsoleted by advances in technology. Therefore, RISC ISAs standardization does no meet the standardization criteria mentioned previously.

Another approach that system designers can use the *function-first* design approach. This provides a large amount of application-specific information during the design of the hardware. The designer may then select the lowest cost existing architecture that satisfies the given performance and physical constraints.

If no satisfactory candidates exist and keeping in view the need to hold down cycles/instruction and cycle duration, there are ways to incorporate additional function in an architecture to increase performance for a given application, i.e., to develop an ASISC.

In summary this presentation made the following observations:

- DoD has standardized on a high level language; therefore, ISA standardization is at least redundant and probably counterproductive.
- Benchmarks of computer systems are only relevant to the extent that the benchmark captures the performance of the proposed application.

- If application-specific analysis reveals a payoff, designers must be free to utilize application-specific ISA extensions so long as such extensions are consistent with the development schedule.

4.7 Military Aeronautical Communications System (MACS) Ada Presentation

Captain Don Pottier of the Department of National Defense (Ca ada) and Stephen Michell of PRIOR Data Sciences (Canada) presented an overvi of an Ada-based system called the Military Aeronautical Communications System (MACS).

In an attempt to gain experience in using Ada in defense weapon systems, Canada's Department of National Defense sponsored a research and development effort to implement an existing component in a defense system in Ada. The system, Military Aeronautical Communications System, controls radio transmitter hardware that is remotely located. The existing system was implemented in Pascal on a PDP-11/23 and RSX-11M operating system (OS). [1]

The goals of the project was to rewrite the application code in Ada and use as much of the existing hardware as possible. At the time of project startup, a validated Ada compiler for the PDP-11 was not available.

It was decided to purchase a MC68010 processor board that could be placed on the existing Q-bus. There were two advantages to this approach:

1. A validated compiler was available for the MC68010.
2. Existing hardware on the Q-bus could still be utilized.

A Microvax-II having a Q-bus was used in running VMS.

The design of the system was to mirror the MACS Pascal design. The rationale was to reduce the risks of redesigning the system. Since the orginal design used operating system service calls, the Ada based system would have to provide such an interface.

The RSX-11M services used in the orginal system are listed below:

- scheduling of processes,
- communication between processes,
- timer functions,
- interface to serial I/O devices, and
- disk I/O.

Ada tasking provided the scheduling mechanism that was needed from the OS. Rendezvous provided the communications between tasks. Ada delays were used for the timers; however, several customized timers were implemented to provide

[1]PDP-11/23 and RSX-11M are trademarks of the Digital Equipment Corporation

capabilities outside the scope of the delay construct. A package written in Ada was used to provide the interface, operations, and data types needed for the serial I/O device. The design was modified so that the disk was loaded at system booting into memory, a disk driver was not required.

Several problems were encountered by the MACS Ada design and implementation. The Ada interrupt mechanism provided with the compiler was not adequate to support several fast interrupt services of the systems. To circumvent this problem, an assembler module was implemented to buffer data for a calling task.

Using the Ada tasking mechanism (RSX-11M was used in the orginal system) cause some deadlocks to occur that had not been experienced in the Pascal-based system. Customizations to the run-time system were necessary during the development of this system. It was suggested that since this was considered an essential capability, it should be provided explicitly in the development environment.

4.8 Real-time Distributed Database Management

Pat Watson of IBM Federal Systems presented requirements for a real-time distributed database management system (DBMS). The intent of the presentation was to give the audience a perspective of some real-time requirements for distributed database systems. It should be noted that this presentation was given from a system's viewpoint.

4.8.1 System Description

The system environment is an on-board submarine DBMS. It is distributed on a Local Area Networks (LAN) with anywhere from 50 to 500 processors. System must be highly reliable. Redundancy is used to recover from hardware faults and extensive testing is performed on the software to guarantee its reliability during various mission scenarios. The response time of the system can range from 10 msec to many seconds.

There are essentially three resources to management in this system: processors, LANs, and database managements resources. In order to meet the system latency requirements, the design must incorporate the appropriate resource management mechanisms.

Processor technology used in this system will more than likely be either a 680X0, 80X86, or combination of both families. Real-time and distributed run-time kernels will need to be optimized for these processors in the near future.

LANs will be fiber-optic bus/ring topologies. The LANS will be networked together and need to support multidomain operations. A real-time Network Operating System will be used to manage message latency.

DBMS will use optical disks; however, older technologies (magnetic disks) will remain in submarine systems for some time. A special purpose DBMS processor will be needed to meet the real-time requirements of the system. It is likely that a combination of processor memory based DBMSs and RAM disks will be used. File servers will be designed and implemented to take advantage of the above technologies.

4.8.2 System Analysis

The effectiveness of the architectures designed for these systems are evaluated by four measures: (1) predictable system response; (2) high utilization; (3) process level reconfiguration capability; and system software update cycle time. Tasks, messages, and database transactions are dependent on the system to guarantee response within a specified time constraint. In some cases, the time constraint will be required to be highly accurate. The system may also be requested to perform these services periodically or aperiodically.

To achieve a highly utilized system, the design will have to support a high degree of resource sharing. This is typical in many DoD real-time embedded systems due to weight and power constraints of current platforms. Utilization of resources also helps lower the cost of the system.

These systems place another constraint: resources must remain stable over a broad spectrum of loads. For a high degree of reliability, the system will include a number of redundant components.

The system must also be able to reconfigure itself in the event of a resource failure. It is vital that the reconfiguration process sheds processes that are least critical. Reconfiguration time must be kept to a minimum. There should be an efficient closed form solution for finding viable reconfiguration options.

During the operational lifetime of the system, changes to the software must not require large regression time for testing modules which are not changed. This type of isolation from testing is difficult to achieve.

4.8.3 Predictable Timing

It was observed that one of the major causes of failure in real-time distributed data base systems is timing. This is a result of poor system management of shared system resources. The solution to this problem is to assure that each shared resource be schedulable: Direct Memory Access (DMA) control, I/O channel, data bus, disk, display, database manager, and microprocessor/hardware controller.

In order to assure that each shared resource is schedulable, it is necessary to be able to predict timing behavior. This is critical in meeting requirements for highly utilized real-time distributed systems with hard/firm deadlines. The following list is a set of system behaviors that require predicatable timing: responsiveness, operability, reconfiguration, availability, reduction in test time for complex real-time systems, cost, rapid system update, and system readiness.

4.8.4 System Partitioning

The submarine system described in this presentation was based on a set of assumptions that apply to mission-oriented real-time distributed systems that are within the application domain of submarine platforms. The first assumption is that only message and data base management interfaces between processes are allowed. Although a process could consist of tasks that share memory, all interprocess communication are still restricted to messages or database transactions.

The rationale for the above communication scheme is that it enables task within a process to be implemented so that real-time performance requirements can be met. It enforces software modularity that would greatly assist in system maintenance and test. A standard communication interface that facilitates system integration for components. And the system would be easy to update because data would be accessible to new software.

The second assumption is that the only system tasks not known at design time are database queries.

4.8.5 Real-time Database Performance

Real-time distributed DBMSs performance will be effected by the following:

- subset of relational access capabilities,
- distributed database/file management,
- precompiled queries,
- data maintained in application format,
- memory and disk based relational tables,
- time driven scheduling,
- separate file management facilities,
- separate file management facilities, and
- selective use of consistency mechanisms.

With respect to system performance, database consistency could be considered harmful in a real-time system. In many cases, consistency is not always required and since consistency requires blocking, real time deadlines have a greater potential to be missed and results in unpredicatable timing behavior. The design of the system should be based on selectively choosing consistency preserving mechanisms.

4.8.6 Summary

At present there does not exist an integrated system wide approach to managing tasks, messages and database transaction latencies. Until an unified approach is identified or created, system development, testing, and maintenance will be costly.

Since the distributed/real-time system describe here is typical of many DoD applications, predicting and management of network latency will need a formal model to base their design. Otherwise, it wil be impossible to consider either the reuse of components (where new design will be required) on the design and implementations of other systems to reduce cost and time future or extending current systems.

Share resources (e.g., CPUs and I/O channels) must be scheduled based on user specified response requirements. In a distributed system this may involve a combination of resources hosted on a number of machines throughout the network. As mentioned above, managing network latency is difficult (due to lack of formal methods) and designers need an integrated system approach.

4.9 Review of Current Ada compiler and Run-time System Support for Real-time Systems

Mike Kamrad of TRW presented a set of Ada run-time concerns and activities that the Ada Run-time Environment Working Group (ARTEWG) has identified and is currently evaluating solutions and approaches to the problems.

Using Ada as the standard programming language for DoD systems has resulted in giving an impression that great improvements in software reliability and reduce lifecycle costs will occur immediately. Ada is also perceived as providing easy solutions to difficult application domains that are real-time, distributed, and highly reliable.

The above expectations have created a number of management problems ranging from the improper application of the language to overoptimistic design, implementation, and testing schedules (directly proportional to cost). The Ada community is now faced with achieving a set of ambitious promises (greatly improved software productivity) and general perception by program offices and management as Ada being the *ultimate software solution.*

The initial set of validated Ada compilers were lacking in the performance that was needed to meet the stringent requirements of embedded real-time DoD systems. Improvements in Ada compilers and run-time systems over the past few years have enable applications to be used in more critical DoD systems. Although the improvements are an encouraging sign, Ada usage in a number of application domains is still limited.

The developer of applications in Ada is confronted with little knowledge about the Ada run-time environment. For instance, mechanisms that are used to support memory management, tasking, and interrupts could have major impacts on the design of the software, yet mechanisms that support these functions can vary greatly. The degree of control that the user has over the run-time system may inhibit Ada use in real-time applications.

The Ada vendor community has had to devote much of its resources in the early eighties with respect to validating their compilers and spreading their technology as widely as possible. Performance took a back seat, although as noted previously, recent improvements have been encouraging. This has resulted in a lack of knowledge with Ada vendors about applications and supporting executives.

Ada has been designed to be used across a large number of application domains within the DoD that require construction and maintenance of large programs. The goal of standardizing on Ada was two-fold: developers were required to be familar with only one language and a standardized support environment for creating, editing, compiling, testing, and documenting programs.

As projects began to use Ada, a number of problems were exposed with several software methods. Programming in the large for systems that are real-time and support mission critical functions have traditionally been approached in an ad hoc manner with respect to design, implementation, and testing. A number of the software methods being applied had not been used extensively in the development phase of a system or only for a specific class of applications.

The following concerns have been summarized in the Table 4.1:

Concerns	Example
Promise of great improvement	Software productivity
Perception of use	Real-time Distributed Systems
Failure of implementations to meet expectations	Performance
Lack of knowledge about run-time environment	Different mechanism to support tasking and memory management
Compiler support for specific application domains	Avionics
Problems with current software methods	Analysis of Real-time systems

Table 4.1: Concerns of Ada Community

The ARTEWG has supported a number of activities that are currently addressing issues that are concerns to Ada designers and implementors of real-time embedded systems. The group has produced a white paper entitled, *The Challenge of Ada Runtime Environments.*

This paper noted that the first generation of Ada runtime environments were inadequate with respect to size, tailorable, or a level of performance for real-time embedded systems. Traditionally, the designer and implementor of a system would write their own executive, therefore decisions on performance and size were in their control. As the Ada run-time environment is produced by a vendor, modifications to the kernel are much more expensive with respect to cost and time.

It is difficult for a vendor to customize a kernel that can be used across a board spectrum of real-time applications. Vendors have few resources to supply and support application specific runtime environments since the cost and amount of resources that would be required would be prohibited. Also, communication between vendors and users of runtime environments is lacking, making feedback from problems in the field difficult and thus increasing the time between improved runtime environment versions.

To provide assistance in solving the above problems, the white paper recommends a set of strategies and tasks on detailed investigation of technology and requirements, evaluation through proofs-of-concept, and changes in DoD and Tri-Services policies that will accelerate Ada runtime environment technology and successfully insert that technology into the development of embedded real-time applications.

ARTEWG has reviewed and analyzed the Ada Language Reference manual for runtime implementation dependencies. This has been documented in, *Catalog of Ada Runtime Implementation Dependencies.* This document provide a list that explicitly shows where runtime vendors have flexibility in the implementation of their kernel's.

The benefits of cataloguing runtime implementation dependencies is that it provides guidelines for selecting a compiler that best fits its behavior requirements.

As with many other fields, a common vocabulary and accepted models and concepts are necessary for precise and accurate communication. The ARTEWG paper, *A Framework for Describing Ada Runtime Environments*, describe a proposed framework for the Ada runtime environment.

The document provides tutorial information in the form of a historical perspective on runtime environments. This information is logically sequenced leading the reader to a current description of the Ada runtime environment. A glossary is included that gives a concise and consistent set of terms used throughout the paper.

Ada has been applied to a number of applications. In order to assist the Ada community understanding of current and future DoD embedded systems requirements on the Ada runtime environment, an analysis was performed on various applications.

The analysis was a set of questions that asked to implementors involved with an applications that were related to the runtime requirements. The information obtain from the questionnaires has been correlated and documented as *SIGAda ARTEWG Survey of Application Requirements.* A tutorial is included describes mission platforms and types of mission functions the application supports.

4.10 U.K. Ministry of Defence Ada Evaluation System

Mike Looney of the U.K. Admiralty Research Establishment Ministry of Defense (MOD) presented an overview of a system that evaluates Ada environments.

Ada is the single preferred high-order language for defense real-time computer systems in the U.K., as of 1 July 1987. The language has not been mandated due to political problems that may have resulted.

The MOD has developed an evaluation suite to access the performance of validated Ada compilers. The suite is over 100K lines of Ada code (200 test programs) and analyzes the following areas:

- quality of generated code,
- quality of diagnostics and informational output,
- compile-time performance, and
- compile-time and run-time limitations

The tests were targeted to four different types of systems: embedded, distributed, large scale, and real-time.

A test bed as been constructed to run the test suite on various machines and compilers. It is expected that more machines will be added to the testbed and compilers to evaluate. Tests that evaluate task pre-emption, interrupts, calls to services outside the Ada environment, asynchronous I/O, scheduling, and memory management are still in need of refinement.

Ray Foulkes of Yardly Ltd. presented several concerns that currently exist within the U.K. about using Ada for real-time systems.

The concern of the overhead involved with context switching between tasks is an issue that the Ada community has been struggling with and continues to try to improve upon since the first set of validated compilers became available. In real-time systems, a design can be severely penalized on a target and compiler that does not support an efficient context switching mechanism.

It has been observed that to avoid such problems, many project teams are prohibiting or severly limiting the use of the Ada tasking. The problem with this approach, is that it prevents the designer and implementor from taking advantages of all the benefits of Ada tasking.

Another attempt to gain enhancements in performance has been to place some of the application into the run-time kernel (e.g., device handlers). This was refer to as *creeping assemble intrusion.* This essentially increases the run-time kernel size and decreases the application size. Most run-time kernels today are implemented in assembly language. This defeats one of the advantages of implementing an application in a high order language.

Some designs have called for additional hardware to support or enhance the Ada language. The disadvantage with this approach is that it increase the power, space, and weight requirements to support the additional hardware. In many cases, the system cannot afford the extra costs.

In the U.K., the lowest cost bidder wins the contract regardless of the high ordered programming languaged used. To meet the performance requirements of many real-time systems, Ada may be more costly and therefore discourage its use.

Two points should be noted about the above statements. First, the initial use of Ada (due to its maturity and lack of experience appling it to real-time systems)will be more costly than languages currently being used by the real-time community. Second, life-cycle costs are not taken into account into the above contracts in the U.K.

The current use of Ada in the U.K. will be prohibited since it presently cannot be competitive on a cost basis with other langauges. One of the major cost savings that Ada provides is in the maintance phase of the software life-cycle. This is not currently being emphasized in the U.K.

In conclusion, an important point was raised with respect to using an elegant model (such as the tasking model) for design, resulting in a solution that is inelegant. The concern is that if Ada is to used for real-time systems, it must provide an overall elegant solution.

4.11 Architecture Optimization Approaches

Dr. Joseph Linn of the Institute for Defense Analyses (IDA) gave a presentation of joint research between the Institute for Defense Analyses and Stanford University, entitled *The Architect's Workbench*, the following paragraphs are an overview of this work.

Technology is reducing delay and increasing the density of new processor chips. It is reasonable to consider that CPU performance can improved by adding resources. This will occur if the average cycles per instruction does not increase.

One possible analysis to determine if performance will improve is to design a machine that uses the extra resources, write a compiler for the machine, write the application code, compile the application code, and execute the application on a simulator for the machine or on the machine (if available). The problem with this approach is that it is expensive.

Another approach, the focus of this research, is to develop a set of tools and methods to facilitate CPU performance analysis so as to help determine exactly how to add resources to the system. The major objectives of this research are listed below:

- develop a design/analysis tool for application-specific architectures for embedded applications;
- allow early evaluation of architecture and architecture/compiler pairs; and
- allow cost and performance predictions of proposed systems.

There are several possible approaches that could be taken for the development of such a design/analysis tool. Expert intuition could be used; however, past experience has demonstrated that experts make mistakes (probability of errors is proportional size) and therefore lowers the reliability of the tool. Another approach is opcode counting, which is considered unscientific. This technique (used without dynamic optimization) cannot generate unbiased results. Opcode counting using dynamic peephole optimization is theoretically possible; however, the best known techniques for doing peephole optimization is three instructions.

The selected approach for this research is to develop functional architecture simulations based on descriptions of the proposed architecture(s). This is accomplished by generating an architecture-independent representation of the flow-graph of the application program (which for this work is Ada).

From the flow graph, the code is input into an optimizer. The optimized code then input into an architecture simulator, which includes the *architecture description*, to generate a dynamic block-level trace with an address trace nested within.

Next, a cache simulator is used on the memory trace generated from the previous steps against various memory hierarchy simulations to determine performance. A technology analysis tool is also available to measure cost parameter, for example silicon area.

It was noted that most compiler optimizations are architecture-independent and preserve the flow graph of a program. Thus, the flow graph of a program after optimization is largely independent of the target architecture, although for certain optimization for architectures, this is not true.

The previous concern is important for Ada. There are certain characteristics of Ada such as constraint checks that can cause a program to run inefficiently. Constraint propagation can be used to make the program run efficiently. Problems like these may make it necessary to include multiple Ada front-ends to *The Architect's Workbench.*

4.12 Real-Time Communications

Alfred Weaver of the Department of Computer Science at the University of Virginia presented a summary of performance data that was gathered from experiments on a Local Area Network (LAN) running communication protocols based on the International Standards Organization (ISO) Open Systems Interconnection (OSI). The relationship of this work to real-time Ada systems is based on the use of protocol standards (i.e., OSI) for communication between a distributed Ada system on a LAN.

The testbed consisted of two host CPUs (based on an Intel 80286 CPU 6.0 MHz) with one megabyte of RAM. The physical communications media was a iSBX554 token bus. The software was supplied by Intel and is an early implementation version of the OSI communication protocols for the data link (8802/4) and ISO 8073 Class 4 Transport. Experiments were performed with only two hosts on the token bus.

The network layer was bypassed since all communications would be within the LAN. It should be noted that overhead associated with an existing network layer would have an impact on the performance.

Throughput and delay measurements were made at the data link and transport layers. The experiments adjusted several parameters at the two layers: retransmission timeouts at the transport port layer; maximum window size at the transport layer; and message size at both the transport and data link layer.

The results indicated that system throughput under the best conditions could act as a bottleneck for distributed applications (e.g., Ada tasks) needing to transfer information. Although more efficient OSI implementations will be developed in the future, there is some concern about their application in real-time systems.

4.13 A System for Evaluating Ada Implementations using Synthesized Benchmarks

John Knight of the Software Productivity Consortium presented an Ada evaluation system that uses synthesized benchmarks instead of the traditional benchmarks to provide specific implementation information with respect to system loading. The presentation addressed the current dilemma that a software manager must cope with attempting to analyze the feasibility of an Ada software design based on preselected hardware and system constraints.

Many past and current systems have been built with the hardware selected first and then having the software designed to meet system performance and constraints. The targets for these systems were generally simple synchronous design, and the software was a mix of high-level (e.g., FORTRAN) and assembly languages.

Current and future systems characteristics will be complex targets, asynchronous design, and will use Ada for the implementation of the software. The current group of Ada compilers have been in existence for about five to seven years. Although there has been a steady history of improvements in compilers and run-time systems, performance differences on a target bearing different loads can be considerable.

Since software development is costly, there is a need to predict if a software design (implemented in Ada) can meet system performance and constraints. To reduce this risk, a tool is needed to estimate performance.

Classic benchmarks have been used in the past to assess relative performance. However, these results give only the following information: which hardware is fastest; which compiler is fastest; and is the new version faster than the old. The following is a list of problems with existing benchmarks:

- None addresses specifics of a particular application.
- Few make provision for dependence on load.
- None allows parametric studies. The following areas need to be considered:
    1. There might be discontinuities in performance.
    2. There might be feature interaction.

What is needed is a method that assesses the application's absolute performance with respect to real-time deadlines and memory limits. A *benchmark synthesis system* has been developed that takes as input a load description that will match the target environment. The load description is a special purpose language that can describes the following details:

- input and output activity,
- exception existence and rates of occurrence,

- anticipated computational activity,
- data volume,
- input and output rates,
- tasking structure, and
- task synchronizations.

The load description is then compiled by a *Benchmark Synthesizer* compiler. When the expected load has been defined, the system synthesizes an Ada program that, when executed, subjects the target environment to the load defined in the benchmark description. The Ada program is compiled and then executed on the target hardware, various timing measurements are made, and a postprocessor generates a performance report from the measurement data.

Three timing problems were identified that had to be overcome by the benchmark synthesis system:

1. inaccurate measurements (low resolution clocks),
2. overhead from clock sampling, and
3. overhead of randomization.

Nothing can be done about clock resolution; however, by taking a large number of samples and averaging this will reduce the error in clock variations.

Problem two is caused by sampling the clock and storing the time in memory. This essentially effects the timing of the system. To resolve this problem, the operational code was timed less the measurement code[2] absent and present. This was handled by the standard method of executing a code segment with and without the code of interest. The difference in time between the two will give the time value of the code under measure (clock overhead is factored out). The effects of the randomization code were handled in the same manner.

Compile-time optimizations can also be a problem for the system. For instance if the benchmark yields to optimization and the application does not, then the actual system would be inadequate with respect to performance. The system ensures that most major optimizations only applicable to the benchmark will be defeated.

The following is list is a set of conclusions from the presentation:

- Adoption of Ada requires reduction of perceived performance risk.
- Benchmarking is an appropriate technique.

[2]In addition to the Ada code that is executed on the target to represent the actual system, measurement code is added to record certain events during running of the benchmark.

- Implementations may have performance discontinuities.
- Existing benchmark sets cannot answer the question, *will my application meet imposed constraints on this target using this implementation?*
- Tailored benchmarks can be synthesized.
- Parametric studies can be performed.
- Accurate timing is difficult.
- Large amounts of processor time and disk space are required to execute the benchmark synthesis system.

4.14 2nd International Workshop on Real-Time Ada Issues Recommendations

Several members of this workshop participated in the 2nd International Workshop on Real-Time Ada Issues sponsored by the Ada UK in cooperation with ACM SIGAda and the USAF held in Moretonhampstead, Devon, UK on 31 May - June 3 1988. The UK workshop generated several recommendations that were presented by Anthony Gargaro at this workshop. Below is a list of the UK workshop recommendation and voting[3]:

SCHEDULING

Recommendations

- Standard should facilitate user-defined scheduling strategies (31-1-0)
- Standard should be revised to clarify rules regarding *priorities* (29-0-1) including:
  1. FIFO queues
  2. Interrupts
  3. CPU allocation
  4. Static priorities

DISTRIBUTED EXECUTION

Recommendations

- Standard should address units of distribution (20-3-7).
- Virtual node offers potential solution for units of distribution (25-4-2).
- Specific issues to be addressed by 9X should include:
  1. pragma shared (26-0-5)
  2. conditional/timed entry calls (31-0-0)
  3. anomalous instance of delay (28-1-2)

[3]The voting can be interpretated: (Yes-No-abstained).

ASYNCHRONOUS TOC

Recommendations

- Standard should facilitate the following requirements (30-0-1):
    1. Response to asynchronous events
    2. Fault recovery
    3. Mode change
    4. Partial computation
- Standard should include specific support for asynchronous exceptions (23-0-8).

FAULT TOLERANCY

Recommendations:

- Standard should facilitate application controlled reconfiguration and recovery (30-0-1).
- Standard should provide minimal language support for reconfiguration and recovery (15-0-15).
- Standard should define semantics for failure and distributed execution (19-1-10).

Part VI

Appendices

Glossary

ACM	Association for Computing Machinery
ALU	Arithmetic Logical Unit
ARTEWG	Ada Runtime Environment Working Group
ASISC	Application Specific Instruction Set Computer
CPU	Central Processoring Unit
DBMS	Database Management System
FFT	Fast Fourier Transform
FIFO	First In First Out
ISO	International Standards Organization
HZ	Hertz
IDA	Institute for Defense Analyses
IPC	Interprocess Communications
ISA	Instruction Set Architecture
ISO	International Standards Organization
KHZ	Kilohertz
MACS	Military Aeronautical Communications System
MOD	Ministry of Defense
MRTSI	Model Runtime System Interface
MS	Milliseconds
ONR	Office of Naval Research
OS	Operating System
OSI	Open Systems Interconnection
OSUSD	Office of the Deputy Under Secretary of Defense
RAM	Random Access Memory
R&AT	Reasearch and Advance Technology
RISC	Reduced Instruction Set Computer
SEl	Software Engineering Institute
SIG	Special Interest Group
TTCP	The Technical Cooperation Program

List of Workshop Attendees

Mr. Phil Andrews	SPAWAR, Navy Sea Command MS CP6/880 Washington, DC 20362 (703) 692-3231
Dr. Ted Baker	Florida State University 1304 Leewood Drive Tallahassee, FL 32312 (904) 385-8923 ajpo.sei.cmu.edu
Mr. James Baldo Jr.	Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311 (703) 824-5516 baldo@ida.org
Mr. Joseph Batz	Ada Information Clearing House 3D139 1211 S. Fern St. (C107) Pentagon Washington, D.C. 20301 (703) 694-0211 jbatz@ajpo.sei.cmu.edu
Ms. Mary Bender	Army CECOM, Cntr Soft. Eng. AMSEL-RD-SE-AST Fort Monmouth, NJ 07703 (201) 544-2105 mbender@ajpo.sei.cmu.edu
Mr. Clyde Chittister	Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213 (412) 268-7781 chittister@sei.cmu.edu
Dr. Jorge L. Diaz-Herrera	George Mason University, CS 4400 University Drive Fairfax, VA 22030 (703) 323-2713 jdiaz@gmuvax

Mr. Ray Foulkes	Yark Limited Charing Cross Tower Glasgow G24PP United Kingdom 44 41 2042737
Mr. Ed Gallagher	Army CECOM, Cntr Soft Eng AMSEL-RD-SE-AST Fort Monmouth, NJ 07703 (201) 544-4149 egallagh@ajpo.sei.cmu.edu
Gargaro, Mr. Anthony	Computer Sciences Corporation 304 West Route 38 Moorestown, NJ 08057 (602) 234-1100 x5791 gargaro@ajpo.sei.cmu.edu
Dr. Karen D. Gordon	Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311 (703) 845-3591 gordon@ida.org
Mr. Tom Griest	LabTek Corporation 8 Lunar Drive Woodbridge, CT 06528 (203) 389-4001 griest@ajpo.sei.cmu.edu
Mr. Alec Grindlay	SPAWAR 3242 Washington, DC 20363-5100 (202) 692-9207
Ms. Deborah Heystek	Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311 (703) 824-5513 heystek@ida.org
Dr. Norman Howes	Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311 (703) 824-3533

Mr. Robert Johnston	Dept. of National Defense 101 Colonel By Drive Ottawa, Ontario K1A OKZ (613) 944-8620
Mr. Mike Kamrad	TRW Federal Systems Group Command Support Division FP2/321 One Federal Systems Park Dr. Fairfax, VA 22033 mkamrad@ajpo.sei.cmu.edu
Dr. John C. Knight	Software Productivity Consortium 1880 N. Campus Comns. Dr. Reston, VA 22091 (703) 391-1849 knight@software.org
Dr. John F. Kramer	Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311 (703) 824-5504 kramer@ida.org
Mr. Claude Labbe	Defense Research Estab, Valcartier PO Box 8800 Courcelette, Quebec GOA1R0 (418) 844-4346 jclaude@dmc-crc.arpa
Dr. Joseph Linn	Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311 (703) 824-5502 jlinn@ida.org
Mr. Robert Little	British Defence Staff British Embassy 3100 Massachusetts Ave Washington, DC 20008 (202) 898-4618 rlittle@a.isi.edu

Dr. Douglass Locke	IBM Fed. Sys. Div./Owego Plan Bodle Hill Road Owego, NY 13827 (607) 751-4291 locke@cs.cmu.edu
Mr. Mike Looney	Admiralty Research Estab Portsdown AXC4, BLK3 Portsmouth P064AA United Kingdom 705 219999 x2330 mjl%uic.mod.are-pn%uk.mod.relay
Mr. Steve Michell	Prior Data Sciences 240 Michael Cowpland Drive Kanata, Ontario K2M1P6 Canada (613) 591-7235
Mr. Offer Pazy	Intermeterics 733 Concord Avenue Cambridge, MA 02138 (617) 661-1840 offer@inmet.inmet.com
Captain Don Pottier	National Defense Headquarters 101 Col. By Drive Ottawa, Ontario K1A0K2 Canada (613) 996-6891
Mr. Tom Quiggle	TeleSoft 5959 Cornerstone Court West San Diego, CA 92121 (619) 457-2700 telesoft!tom@ucsd.edu
Mr. Clyde Roby	Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311 (703) 824-5536 roby@ida.org

Dr. Lui Sha	Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213 (412) 268-5875 sha@k.cs.cmu.edu
Mr. Leland Szewerenko	Tartan Laboratories 477 Melwood Ave Pittsburgh, PA 15213 (412) 621-2210
Mr. William Taylor	Ferranti International Ty Coch Way Cwmbian, Gwent NP44 7XX United Kingdom 44 6333 71111
Mr. Pat Watson	IBM Federal System Division 9500 Godwin Drive Manassas, VA 22110 (703) 367-4536 watson@k.cs.cmu.edu
Dr. Alfred C. Weaver	Univ. of Virginia Dept. of Comp. Sci., Thorton Hall Charlottesville, VA 22903 (804) 979-7529 acw@cs.virginia.edu
Dr. Andre vanTilborg	Office of Naval Research Code 1133, 800 N. Quincy St. Arlington, VA (202) 696-4302 avantil@nswc-wo.arpa

A Framework for Describing Ada® Runtime Environments

Proposed
by
Ada Runtime Environment Working Group
of
SIGAda

15 October 1987

Abstract

The concept of a runtime environment to support program execution has always been associated with application software - it has only been with programming languages like Ada that the concept has become more apparent and significant to application effectiveness. The purpose of this paper is to explain the basic elements of Ada runtime environments in order to support the presentation of ARTEWG documents, such as the Catalogue of Runtime Implementation Dependencies. Additionally, this paper may be useful as a framework (or common vocabulary) for buyers and builders of Ada runtime environments to discuss details of Ada runtime environments.

The paper begins with a historical perspective on runtime environments to show how the technology evolved to the current state exhibited by Ada runtime environments. This leads into a proposed framework for describing runtime environments, including a taxonomy. A concise and consistent set of terms, which are summarized in a glossary, explains different elements of Ada runtime environments.

1. Historical Perspective on Runtime Environments

Early in software development, software engineers wrote software applications on a bare computer with few or no supporting mechanisms. The bare computer represented all theunderlyingcomputing resources available to the software applications. Software was expressed in terms of a subset of the bare computer features, as Figure 1 shows, because not all the features of the bare computer were used by the application.

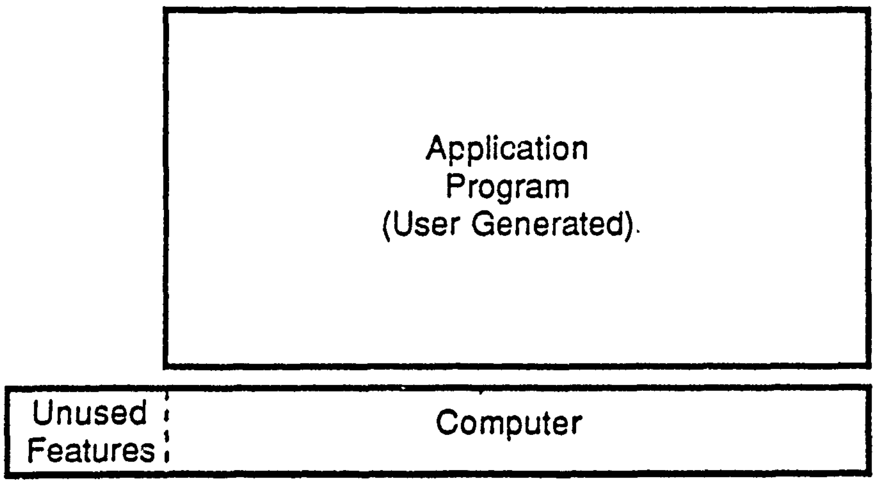

Figure 1. Application Program Execution

In this environment the software engineer had complete freedom to use any programming mechanism or convention which the bare computer offered for the application. Additionally, the software engineer was responsible for providing all the implementation details of the application program. With complete control the engineer was able to fully customize the software to meet the requirements of an application. But reusability of software developed in this environment was minimal because software whose programming mechanisms differed or conflicted was difficult to combine in the construction of new applications.

1.1 Rudiments of Runtime Environments

As more application software was written, software engineers began to adopt **common conventions** for writing software. This was especially true when more than one software engineer was involved in producing the software. These conventions were adopted for reliability and interoperability purposes. Conventions made the job of writing software simpler and easier by suppressing unnecessary implementation details. These conventions frequently included how the set of registers was to be used and how subroutines were to interface with one another. Additionally, as repetitious code sequences were recognized, macro generators were developed so that shorthand notations could be used to reproduce those sequences.

Concurrently, software engineers recognized the need for more powerful software abstractions than those offered by the underlying computing resources of the bare computer. For example, stacks were recognized as a means to support recursive programming, and arrays, records, and list structures were adopted to handle data in a more abstract manner. Similarly, mechanisms and algorithms were developed to perform the onerous tasks of input and output. All these concepts required common conventions on how the data was to be constructed and how the structures were to be manipulated. In some cases, such as handling input-output and mathematical calculations, sets of prewritten subroutines were developed which software engineers could include in their application software.

These programming mechanisms, namely predefined subroutines and common programming conventions for data and code structures, were the **basic elements of a program runtime environment.** They created for the software engineer a more abstract and therefore more effective underlying computing resource than that of the bare computer. Figure 2 illustrates how these elements are integrated. As before, the application software executed on the bare computer using a subset of the machine features, but now the application software was split between the software generated directly by the engineer and the predefined subroutines that the application required. The predefined subroutines were selected from libraries of such subroutines available to the software engineer. Some parts of the user-generated program were directly supported by the predefined subroutines while the remainder of the program had direct access to the bare computer, as the interface between the user-generated program and the predefined subroutines and bare computer in Figure 2 shows. Within the user-generated program the software engineer adhered to coding conventions for manipulating abstract data structures and for calling subroutines, which resulted in fragments of similar code sequences throughout the generated software. Likewise the software engineer utilized abstract data structures, such as records, arrays, and stacks. Altogether the predefined subroutines, the abstract data structures, and the coding conventions helped to create an **abstract machine** for the generated application software which used them.

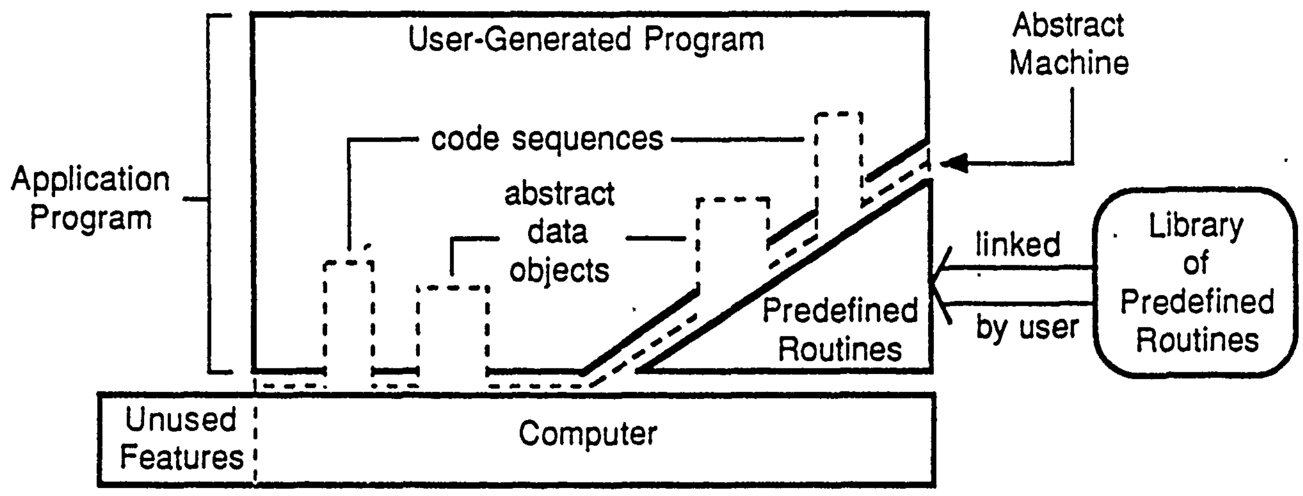

Figure 2. Early Use of Runtime Environments

It was important that the software engineer adhered to these conventions because of the interdependence of the predefined subroutines and the common coding conventions. These conventions ensured the safe coexistence of the user-generated code with the predefined subroutines.

Strictly speaking, each application program created its own abstract machine. The definition and use of abstract data structures and the common coding conventions, as well as the use of subroutines from the libraries of predefined subroutines, could differ with each application program, in effect creating for that program its own interpretation of the underlying computing resources. Over time, programs for similar applications adopted similar concepts of runtime environments, resulting in commonality of abstract data structures, code sequences, and predefined subroutines.

1.2 Introduction of Automated Software Support Tools

The development of system support software, such as **executives** and **operating systems** (for the purposes of this paper, operating systems are associated with general purpose computers and executives are associated with embedded computers) and **programming language compilers**, were built, in part, as a means to enforce and automate the establishment of effective runtime environments. The job of providing the runtime environment generally was jointly supported by executives and the programming languages compilers. Executives provided a predefined set of subroutines that enabled programs to share underlying computing resources and utilities while the programming language compilers provided the data and code conventions and the interface to the predefined subroutines that supported the program language abstractions.

1.2.1 The Impact of Executives

As the predefined subroutines became more powerful, useful, and available to software engineers, the frequently used subroutines were collected into a single package of subroutines. Very often the subroutines in this package were integrated and optimized for better performance to encourage the use of these subroutines. This package of subroutines acted as an extension of the computer, providing more expressive power to software engineers. Typically this package of subroutines (whose contents were usually defined by committee) contained more capability than most applications required, which meant that some portion of that package of subroutines remained unused. This package of subroutines provided more convenience to the user as well as increased reusability and reduced development cost of the application software; this was at the expense of less specific support and performance for the application. As long as the performance of the package of subroutines was sufficient to meet application requirements, it was sensible to use the package of subroutines. Over time this package of subroutines evolved into the sophisticated operating systems and executives of today.

Executives extended the underlying computing resources that were available to the application and therefore the runtime environment could be augmented to account for this. With the introduction of executives, as Figure 3 shows, the runtime environment could be changed by taking advantage of some portion of the executive. The runtime environment would also be composed of any additional predefined subroutines selected by the user, and code conventions and abstract data structures selected by the user or required by the executive. Since the choice of using the executive was optional, the elements of the runtime environment supported by the executive were optional. As Figure 3 illustrates, the unused portion of the executive was additional cost that the application had to bear for the convenience of using the executive.

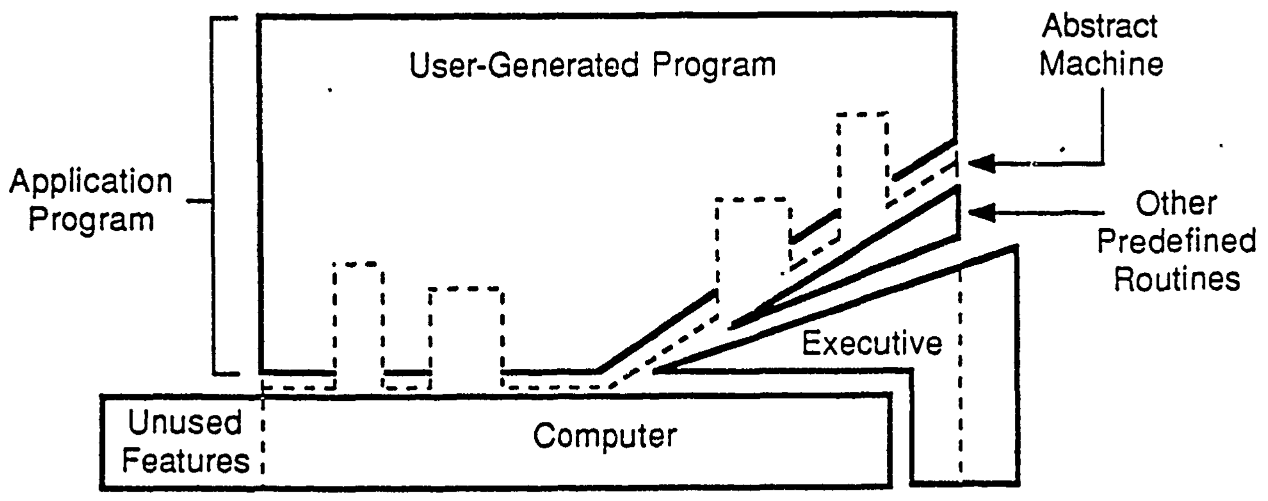

Figure 3. Introduction of Executive in Runtime Environment

For performance and efficiency reasons, applications were divided into multiple threads of control and multiple applications shared the same underlying computing resource. This underlying computing resource was permanently extended by the use of the executive or operating system, as Figure 4 shows. These separate threads of control had to share the extended underlying computing resources that the bare computer and the executive or operating system provided. The executive was responsible for managing the concurrency and communication among these various threads of control. Adherence to the common coding conventions was paramount in this extended operating environment. Without it there could be no safe coexistence among the threads of control.

In this situation the use of the executive was no longer optional. Consequently the runtime environment always included some portion of the executive. It would also contain any other predefined subroutines needed by the application, as well as the code conventions and data structures required by the executive and the other predefined subroutines. The runtime environment of an application was highly dependent on the executive. Changing executives required substantial change to the other parts of the runtime environment to accommodate conventions expected by the executive.

The development of operating systems grew rapidly, with each computer manufacturer providing operating systems uniquely tuned to their hardware. UNIX® is an interesting anomaly in that it was adapted to more than one computer system. This was due to its capabilities and ease of use which overcame any lack of performance over that of the "native" operating system for that computer system.

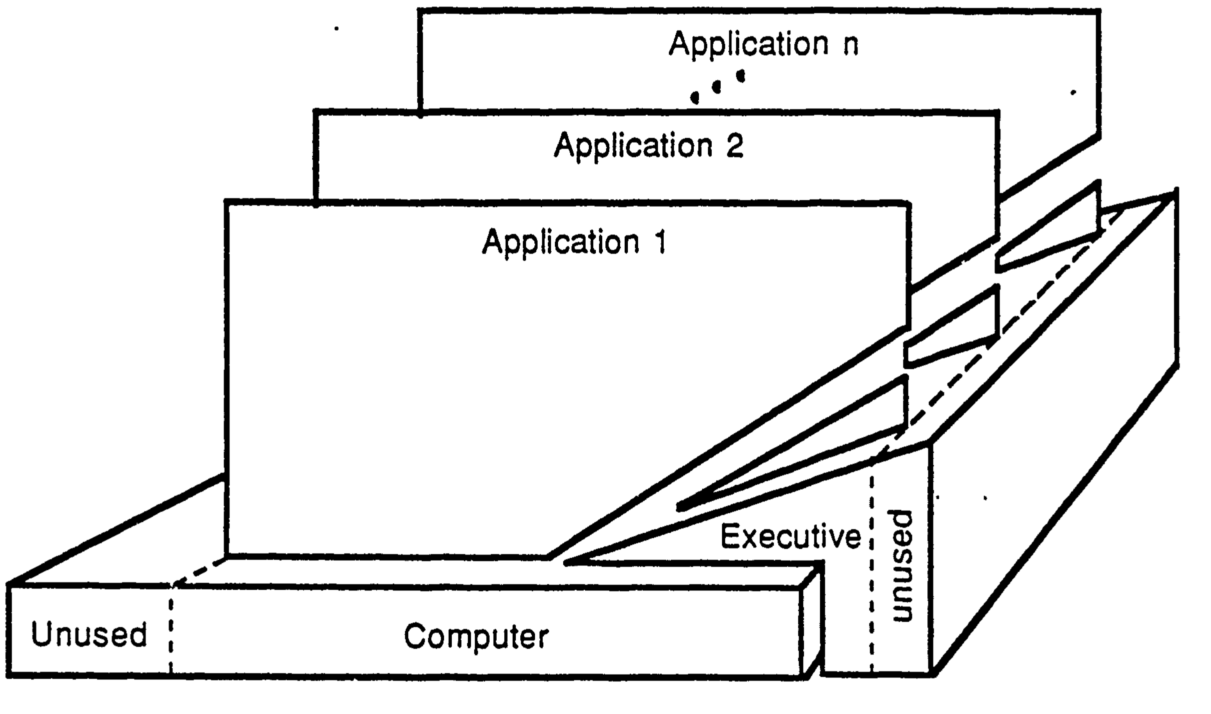

Figure 4. Introduction of Resource/Concurrency Management in an Executive

The development of executives followed a similar track, although it was slower to develop. Because of performance and size constraints, executives were usually unique to both the computer system and the application. The phenomenon of UNIX has been slow to occur in the embedded real-time application area. Yet transportable and adaptable executives for embedded real-time applications have appeared and have been used. The appeal of these executives is similar to that of UNIX, namely, capabilities, ease-of-use, and sufficient performance.

Some configurability was obtainable with both operating systems and executives. All operating systems were configured to provide different performance and size characteristics for each computer system. Likewise the executives available for real-time systems were configured both in their size and the amount of resources they manage. Both operations were performed manually.

1.2.2 Impact of Programming Languages

Complementing the development of executives was the development of programming language compilers. Programming language compilers translated source programs by selecting the appropriate representations of the abstract data structures in the source program and by generating the code sequences to manipulate the representations of those structures as directed by the statements of the source programs. Conventions for the selection of data structure representations and for code generation were established by the writers of the translator. This resulted in the same type of pattern of code sequences and abstract structure representations as shown in Figure 5.

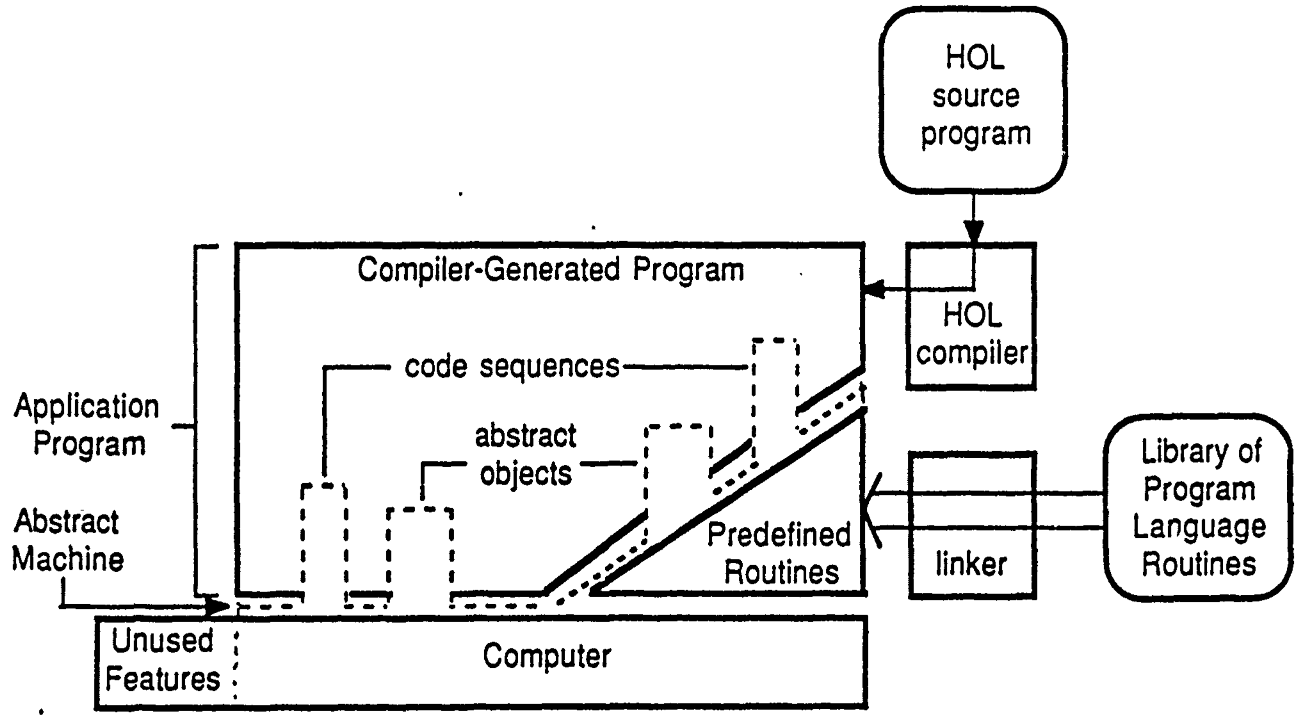

Figure 5. Introduction of Compiled Programming Languages

Additionally there were predefined subroutines that were needed to provide functionality not directly represented in the data and code generated by the translator. These subroutines, which are the predefined subroutines in Figure 3, were usually unique to the program language and were available from the library of predefined subroutines supporting that language. The size and shape of this "wedge" of supporting subroutines varied with each source program, the capabilities of the translator, and the capabilities of the underlying computing resource.

Compilers provided the generated code direct access to the underlying computing resource, as Figure 5 shows. By contrast, interpreters that support interpretation, as shown in Figure 6, produced generated programs which rarely had direct access to the underlying computer; the generated code contained pseudo-instructions and used abstract data structures that could only be executed by the runtime interpreter. The runtime interpreter was an integrated set of subroutines that sequenced through the generated program and executed the instructions of the generated program by invoking appropriate subroutines that supported that instruction.

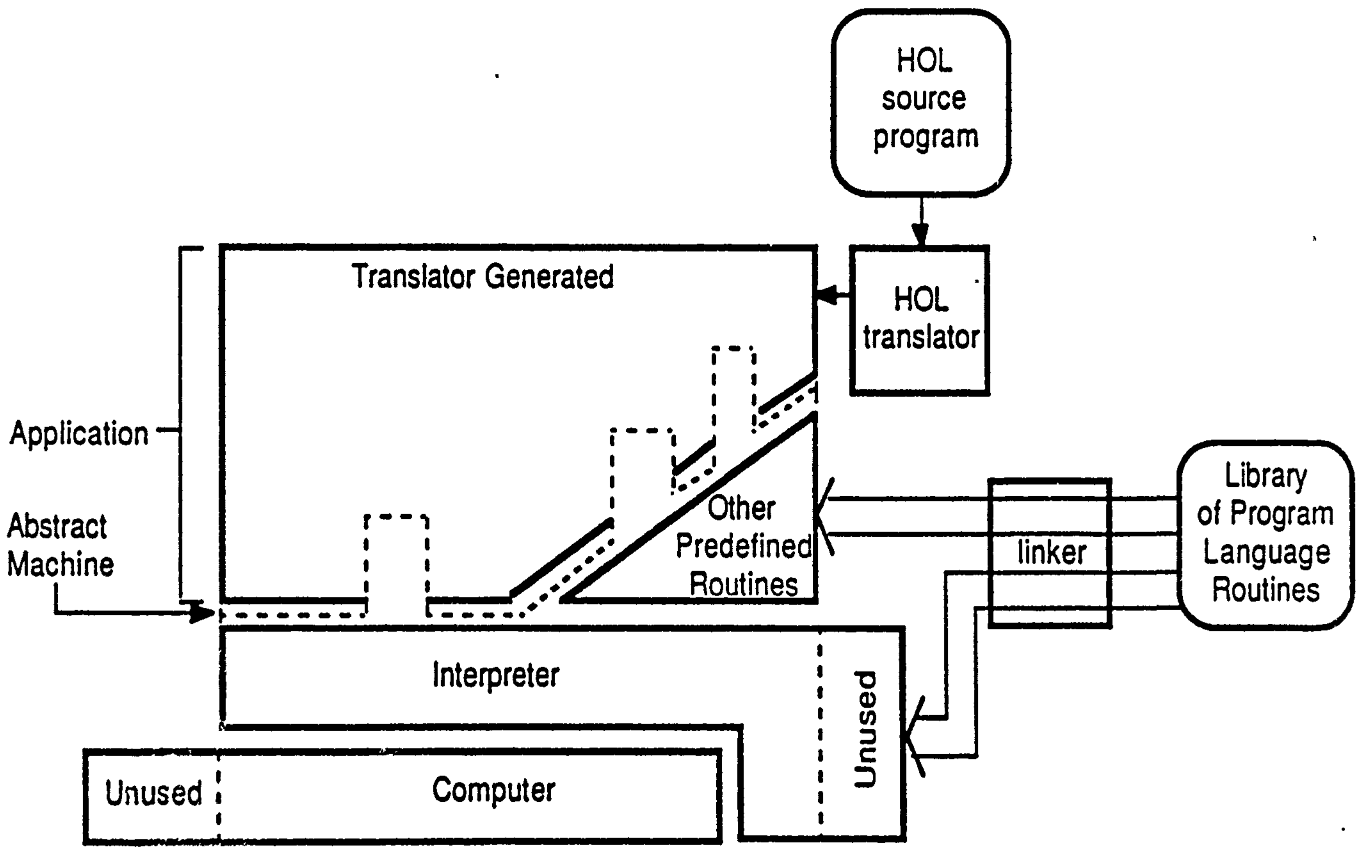

Figure 6. Introduction of Interpreted Program Languages

For programming languages, the runtime environment contained the code conventions and abstract data structures to represent the high-level features and constructs of the language, and a set of predefined subroutines that supported those language features which were not represented entirely in generated code. Each programming language implementation has its own unique runtime environment, which meant that the same application program may have exhibited different performance characteristics when it was translated by different programming language implementations.

The application builder could customize or tailor the performance of the application by the judicious use of language features in the application. That selection process, in effect, was tailoring the specific instance of the runtime environment to improve the performance of the application. In fact, many software engineers quickly learned which language features in a programming language implementation were the most effective for their applications.

While the runtime environments for applications written in a programming language were rarely as effective as ones that were hand-built, they were sufficient to accomplish the application. The power, convenience, and cost savings of using a programming language were sufficient to offset the loss of performance in the runtime environment.

1.2.3 Combination of Executives and Programming Languages

Executives or operating systems and programming language compilers complemented each other in providing the runtime environment for application programs. The executives or operating systems extended the underlying computing resources available to the programming languages by providing many functions to the programming languages and by enabling the application programs to share the underlying computing resource among their own internal threads of control as well as those of other application programs. The programming language compilers provided the implementation of the abstract data structures and operations found in the application program. Implementations of functions which the compiler could not represent in generated code were represented as calls to either predefined subroutines unique to the programming language or to the executive or operating system.

The elements of the runtime environment were jointly provided by both the executive and the programming language implementation. The runtime environment always included the executive, any additional subroutines from the library supporting the programming language, and the code conventions and abstract data structures selected from the language features in the application and required by the executive. The builders of the executives defined the functionality and interface of the executive. In turn, the builders of the compilers had the responsibility to generate object programs which adhered to the conventions established by the executive. The elements of the runtime environment provided by the programming language implementation had to conform to the runtime environment conventions of the executive. This meant that some runtime environment elements from the programming language were either replaced or modified to match the elements provided by the runtime environment of the executive. Figure 7 illustrates how this combined runtime environment supported an application running on top of a general purpose operating system , such as UNIX.

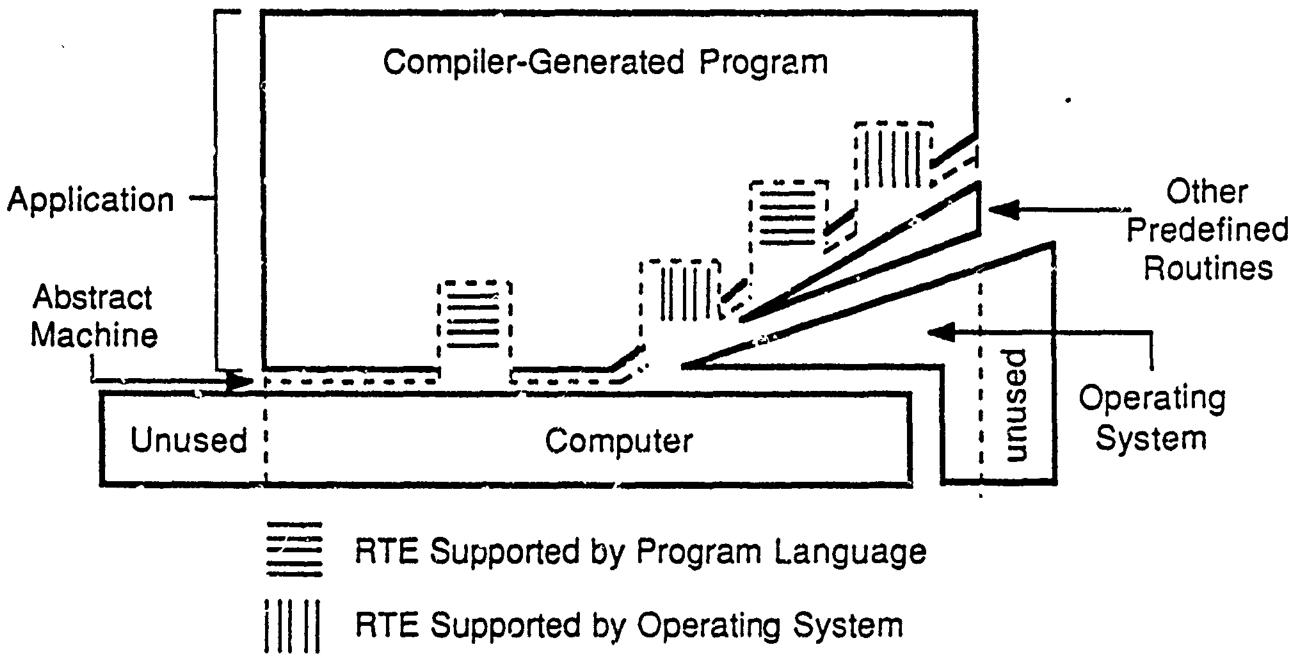
Compiler-Generated Program
Application
Abstract Machine
Other Predefined Routines
Operating System
unused
Unused
Computer
RTE Supported by Program Language
RTE Supported by Operating System

Figure 7. Combination of Program Language and Operating System to Support RTE

The development of embedded system applications paralleled this separation of responsibility between executives and compilers. An application was divided into various sections of sequential code which were encoded in a programming language, such as JOVIAL or CMS-2. A separate complementary executive was developed to handle the concurrency and communication among those various sections of sequential code of the application. Many times the executive was unique to the application, due to the stringent performance requirements of the application. The runtime environments provided by the executive and compiler combination very often exhibited different performance characteristics that reflected the needs of different application areas. Thus, applications written in the same programming language exhibited different performance characteristics due to the different performance needs of the application. This tailoring of the runtime environments to specific applications required the unique development of the compiler and executives for those environments. This pattern has been repeated many times in the course of developing support systems for embedded application systems.

The runtime environments of the same application could be different depending on the capabilities of the underlying computer resource and the complexities of the programming language features. The lack of support from the underlying computer resource implies that the missing capabilities must be supplied by the runtime environment of the programming language and associated executive. The result is a larger and more complex runtime environment with possible performance penalties.

2. Ada Runtime Environments

Ada blurs this separation of the responsibilities of runtime environment support between executives and programming language compilers because Ada includes features for concurrent programming and storage management, and Ada demands no specific supporting executive. Ada programs for embedded applications are expected to directly execute on bare computer systems. Consequently, Ada compilation systems are responsible for providing all the elements of the runtime environment to support applications written in Ada.

2.1 The Elements of Ada Runtime Environments

The **runtime environment (RTE)** for Ada consists of the same three elements, abstract data structures, code sequences, and predefined subroutines, that other languages and executives provide. The **compilation system** for Ada selects the appropriate elements as directed by the **source Ada program** and as dictated by the **underlying computing resource**. The result, as Figure 8 shows, is a **generated Ada program**, which is similar to the generated program produced by other language compilers. The code of the generated Ada program adheres to conventions for data structures and code that the Ada implementors have selected for representing abstract data types and program structures in the Ada language. In addition, the generated Ada program may use predefined subroutines to support features of the Ada language that the Ada implementor has chosen not to directly represent in generated code. The set of predefined subroutines for any generated Ada program is called the **runtime system (RTS)** for that program. These predefined subroutines are chosen from the **runtime library (RTL)** for that Ada compilation system. Altogether, the data structures, the code conventions, and the runtime system selected by the Ada implementation for a generated Ada program provide an Ada **virtual machine** on which the generated Ada program executes.

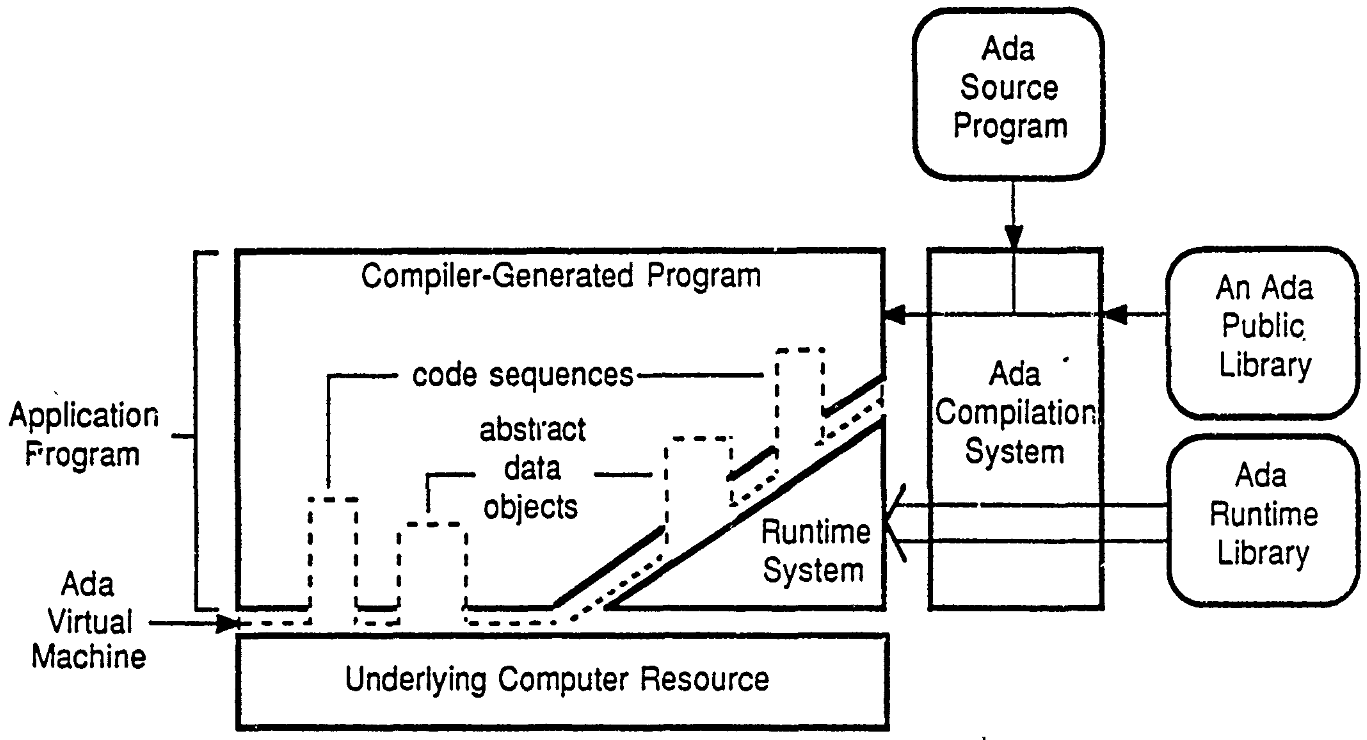

Figure 8. Ada Compilation System

2.2 Ada Runtime Environments with an Existing Executive

Unlike many other languages, Ada includes high level abstractions for concurrent programming, exception handling, and resource allocation. The Ada compilation system is expected to provide the runtime environment that supports these sophisticated features. An Ada compilation system can not assume that generated Ada programs will be supported by a specific executive or operating system. This means that the Ada implementation must be tailored directly to the capabilities of the underlying computing resource. If the underlying computing resource includes an existing executive or operating system as shown in Figure 9, the Ada implementation may choose to have the runtime environment take advantage of some subset of the executive so that the runtime system of subroutines that is created by the Ada compilation system shares the support of the generated Ada program with the executive. This means that the capability needed to support a feature of the generated Ada program, such as tasking, may be provided directly by the underlying computing resource (some combination of the bare computer and the executive) or by the runtime system selected for the generated Ada program. In turn, the runtime system may either provide all the requested capability itself or it may require assistance from the existing executive of the underlying computing resource. Of course, some features of the executive of the underlying computing resource, like some features of the bare computer, may never be used by the runtime environment, as they may be inconsistent or unnecessary to the execution of any generated Ada program.

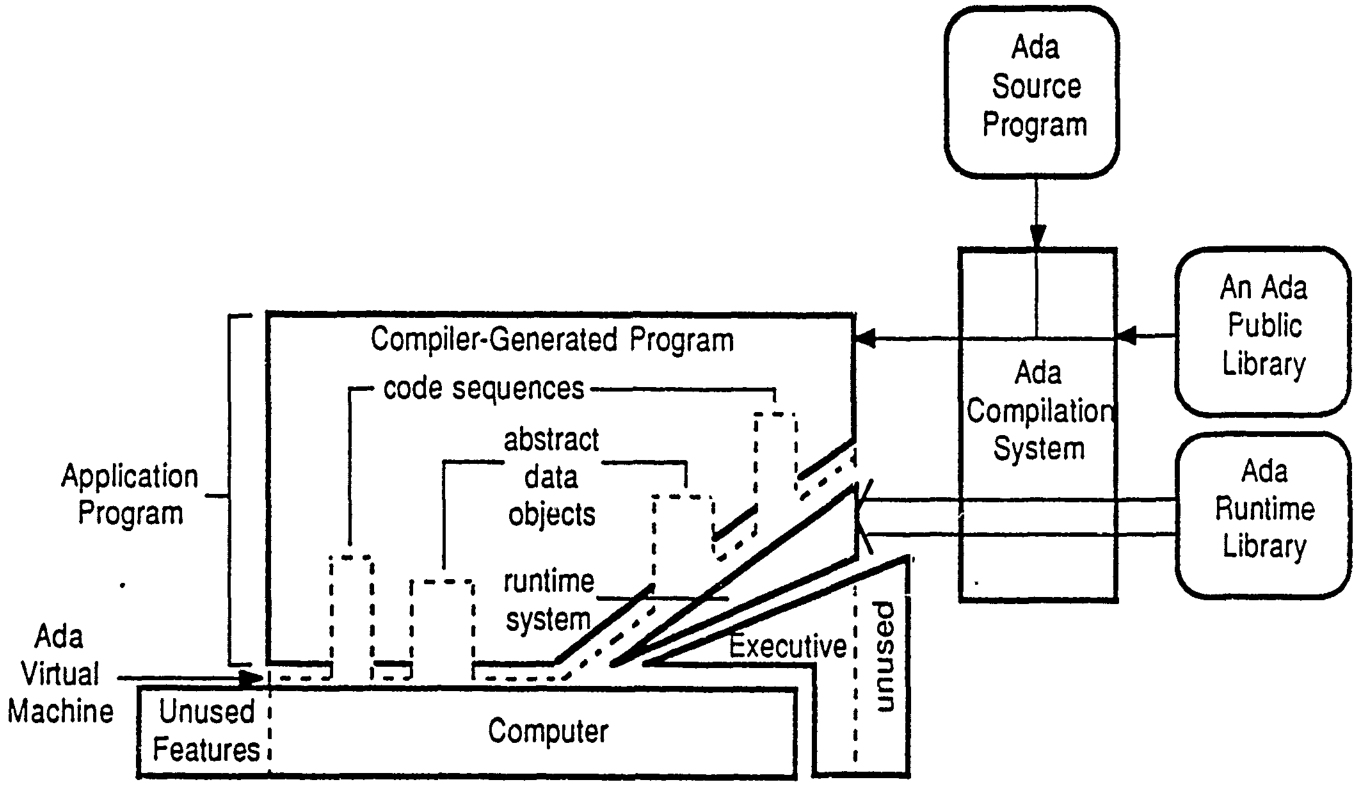

Figure 9. Ada Compilation System Targeted to an Executive

2.3 Ada Runtime Environments on Bare Machines

The challenging case for Ada compilation systems is the development of runtime environments for underlying computing resources with no existing executive. All capabilities required by a generated Ada program that are not directly supported by the bare computer must be supplied by the runtime system for that generated Ada program, as shown in Figure 10. It is apparent that the runtime system has all the aspects of an executive, which has caught the attention of those who are concerned about performance. It would be straightforward to build an Ada runtime system executive that supports all generated Ada programs, as Figure 10 shows. But it does not need to be that way. Ideally, the Ada compilation system can configure the runtime system of subroutines from the runtime library to exactly fit the needs of the application written in Ada, just as software engineers have custom built executives for applications in the past. The result would be the smallest runtime system for that generated Ada program. In addition, this configuration process could potentially generate a unique runtime system for each application written in Ada.

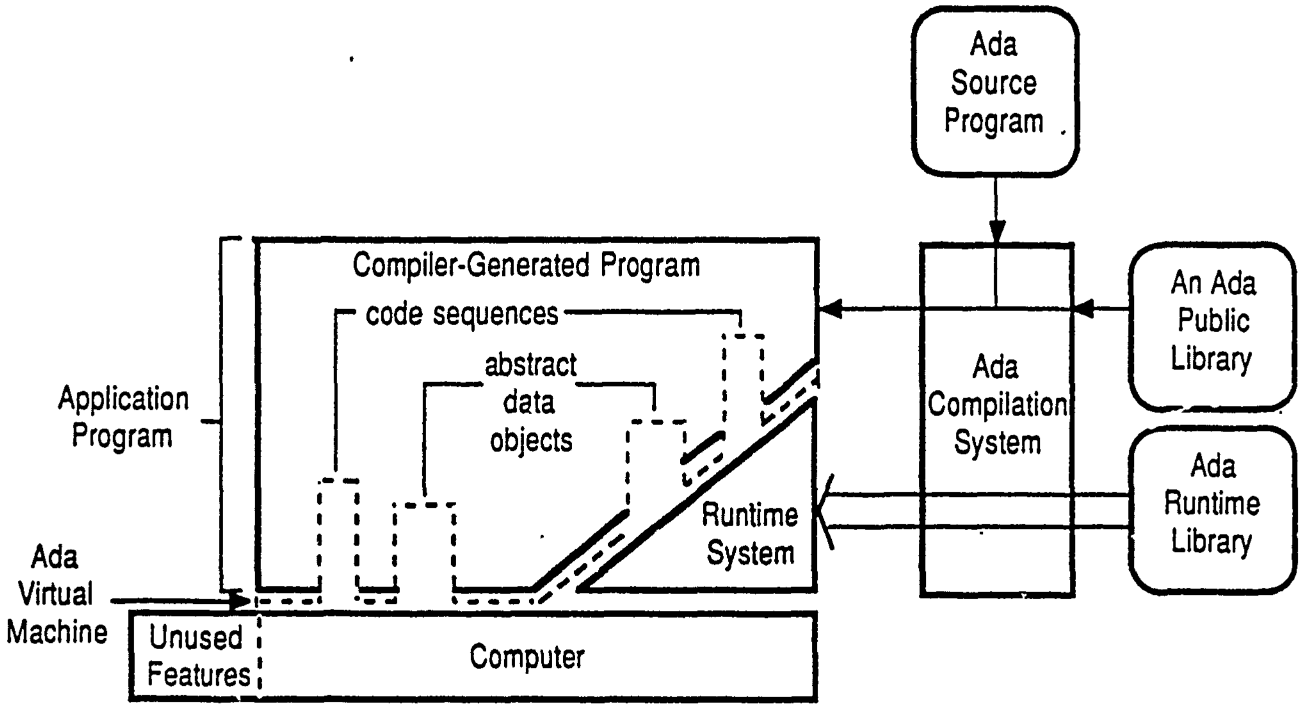

Figure 10. Ada Compilation System for Bare Machine

The runtime environment of the Ada compilation system must always comply with the rules of the Ada language as defined by the Ada Reference Manual. Yet the Ada Reference Manual provides significant flexibility in how the runtime environments support the language definition. The runtime environment is thus allowed to exhibit different performance characteristics (that may reflect the needs of the application) for the same features or combination of features of Ada. In fact, the Ada Reference Manual provides the pragma construct as one method to help the Ada compilation system determine the performance characteristics that the runtime environment should provide for an application. Thus, the runtime environment of an Ada compilation system may be able to accommodate an arbitrary number of interpretations of an application in Ada that comply with the Ada language standard. These interpretations can be guided by the pragma construct or by other mechanisms provided by the Ada compilation system.

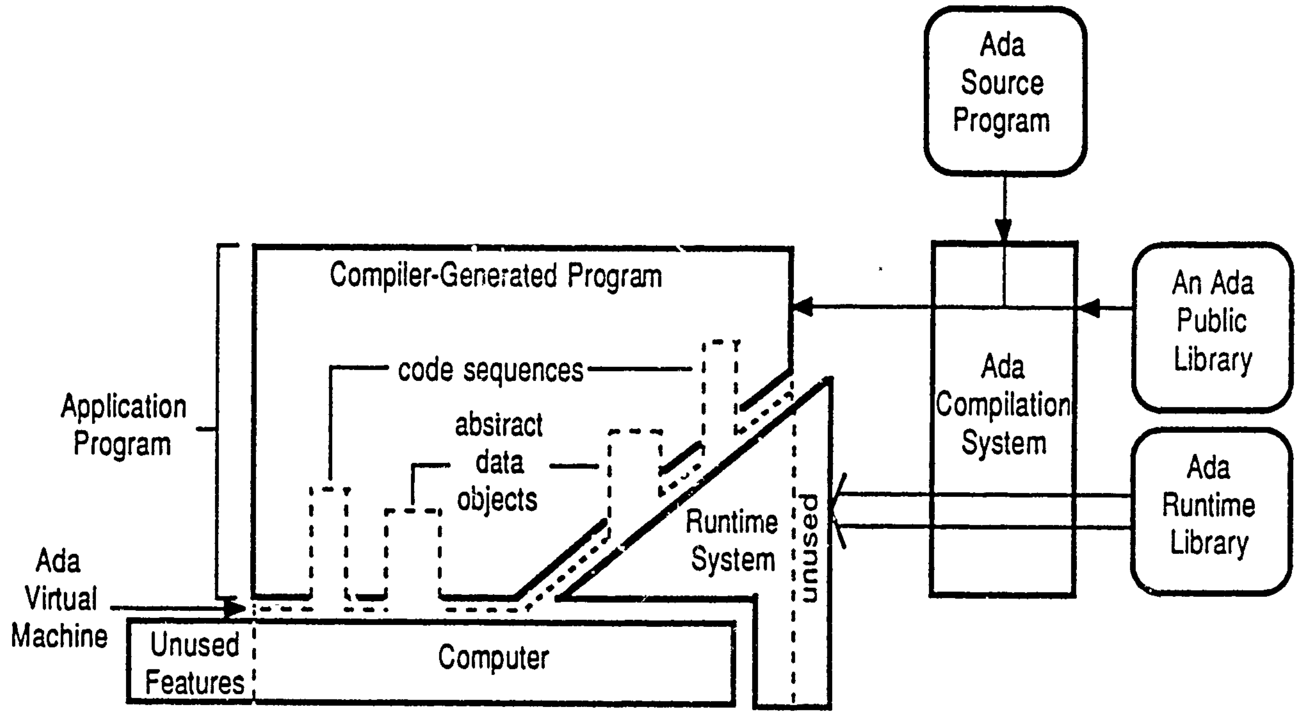

Figure 11. Ada Compilation System with Fixed Ada Runtime System

It should be clear from the preceding discussion that the more direct support the underlying computing resource provides the generated Ada program, the smaller the needed "wedge" of runtime system support for that program. There may be great potential rewards in developing computers that provide significant direct support of Ada features. Conversely, the less direct support by the underlying computing resource for the generated Ada program, the greater will be the size of the "wedge" of runtime system. This may explain the concern of application developers as they begin to realize the limitations of some existing computers to support the execution of Ada programs.

3. Taxonomy

If a runtime environment for an Ada program is composed of a set of data structures, a set of conventions for the executable code, and a collection of predefined routines, then the question arises. what are examples of these elements, and moreover,. what is the complete set from which such elements are taken when a particular runtime environment is built? At the same time, the question of well-defined terminology presents itself. This taxonomy describes a list of functions that can be expected in the runtime libraries for Ada implementations.

It should be noted that the dividing line between the predefined runtime support library on one hand, and the conventions and data structures of a compiler on the other hand, is not always obvious. One Ada implementation may use a predefined routine to implement a particular language feature, while another implementation may realize the same feature through conventions for the executable code. An example is the entry sequence for a subprogram.

This taxonomy concerns itself primarily with those aspects of the runtime execution architecture which are embodied as routines in the runtime library. It does not treat issues of code and data conventions, nor issues related to particular hardware functionalities, in any great depth.

3.1 Runtime Execution Model.

Behind any implementation of an Ada compiler targeted to a hardware/software configuration is a model of program execution on that configuration. This model is developed by compiler designers in order to guide the development process, so that performance and semantic requirements are met by the Ada programs generated for that configuration.

Ada features such as multitasking and dynamic memory management define a set of design constraints for the runtime model which must be made to work with an often conflicting set of constraints derived from the target architecture and operating system.

While the elements of the runtime model affect most parts of the generated code and the accompanying runtime system, it is convenient here to classify runtime model issues into two main categories: those affecting the generated code, and those affecting the division of runtime functionalities between code and runtime routines.

Code sequences are heavily affected by the selected definitions of predefined types and of representations used for addressing objects. Related issues include whether there is one or multiple areas for package (i.e., global) data, what the mechanism for uplevel referencing of objects is (e.g., static link or display), and what the subprogram call sequences and parameter passing mechanisms are determined to be. A related issue is the set of conventions for register usage and preservation of registers across subprogram calls.

Decisions made about the general strategies for dynamic memory management, exception management and tasking management also impact code sequences, often at a very fundamental level. Furthermore, the addressing models of a processor may have ramifications throughout the runtime

execution model, affecting such areas as representation of pointers, implementations of runtime type checks, and even the selection of data structures used in the runtime environment.

The split of functionalities between code sequences and runtime routines is another set of issues which must be resolved in the runtime model. Decisions regarding this partitioning are also very influenced by the capabilities and limitations of the target configuration.

In this area, decisions are made regarding how much of the tasking constructs are handled by inline code versus via calls to runtime routines. Similarly, tmemory management functions are divided among runtime routines and code sequences. Analogous decisions are made regarding the treatment of exception management functions, Ada attributes and commonly invoked routines such as multi-word arithmetic operations.

The main point of note regarding this category is that, while it is possible to define abstract interfaces describing the necessary functionalities, the best decisions regarding allocation of these functionalities to either of code or runtime routines are highly situational, and must be determined based upon the particular target architecture and the performance goals in various areas for the compilation system.

Additional aspects of the runtime model hinge upon appropriate use of the target instruction set architecture, and other target or operating system-dependent issues.

The runtime model resulting from the decisions broadly described here defines the constraints for the design of the runtime environment components listed in the remainder of this taxonomy.

3.2 Dynamic Memory Management

During the execution of an Ada program, it will usually become necessary to dynamically create objects. This may happen because a subprogram with locally defined data is called, because an Ada allocator is executed, or because the compilation system generates an anonymous temporary object for the purposes of computation.

When such objects are created, storage is allocated to represent them. Storage is also allocated at times for objects defined in the runtime environment, which have no obvious representation in a particular Ada program.

The **Dynamic Memory Management function** is that part of the runtime environment which concerns itself with the allocation and deallocation of storage at runtime. This function is also responsible for detecting when a request for storage cannot be fulfilled, and for raising the exception STORAGE_ERROR as appropriate.

Generally, there are two main protocols for dynamic memory management in an Ada runtime environment: stack structured and heap structured allocation schemes. There are a number of common variations of these.

The local variable sets of subprograms behave like a stack in the sense that their lifetimes are nested. It is therefore useful to associate a stack with each Ada task, as well as with the main program. Note that this does not imply any particular implementation of the stack. The **Stack Management function** for Storage Management allocates and deallocates space on the stack and checks for stack overflow. Note that an implementation will typically not only place local variables on the stack, but will also include various administrative variables such as return addresses, lexical parent pointers, dependent task counters, etc. in the "activation record" for a subprogram. The Stack Management function is usually implemented via generated code sequences, rather than via runtime routines.

The lifetimes of objects that are created by Ada allocators tend to be more difficult to predict than those of objects associated with a subprogram activation. While there is a relationship between the scope in which an access type is defined and the collection of objects associated with the type, the lifetime of each such object is generally not amenable to determination at compilation time.

Therefore, the allocation and deallocation of storage for these objects usually does not take advantage of predetermined lifetime characteristics of these objects. This type of memory management is the domain of the **Heap Management function**, which is usually implemented with runtime routines. (Note that pragma CONTROLLED dictates a Heap Management approach which takes advantage of the pattern mentioned above regarding access types and scopes defining them).

In general, Heap Management administers storage so that random patterns of allocations and deallocations can be handled. The function derives its name from the fact that a pool of memory managed in such a fashion is commonly called a "heap".

The Heap Management function implements the allocation of storage for objects that are created through Ada allocators, and it also may provide storage allocation for objects that are internal to the Ada runtime environment or temporaries generated by the compiler. Note that no assumptions are made about the organization of the heap storage. There may be a single system-wide heap, or there may be one heap for each Ada task (and for the main program). If length clauses for collections are implemented, there may be special arrangements for the storage set aside for collections.

Furthermore, there are a number of approaches to reclamation of unused storage. There may be no reclamation, only explicit reclamation (via UNCHECKED_DEALLOCATION), various kinds of garbage collection schemes, pragma CONTROLLED-style reclamation, or a combination of these.

3.3 Processor Management

If an Ada program utilizes Ada's multi-tasking facility (that is, if it contains Ada tasks), then its execution can be viewed as the parallel execution of a family of tasks. Each individual Ada task oscillates, throughout its lifetime, between being "logically executing" and "blocked". Tasks are blocked for a number of reasons: they may be waiting for a rendezvous that is currently not possible; they may be waiting for delays to expire which they have specified in a delay statement; they may be waiting for the termination of dependent tasks; or they may be waiting for the activation of newly created tasks. In this context, the only reason why a rendezvous should be considered impossible would be that no appropriate partner task has reached a point in its execution where it is prepared to engage in the desired rendezvous.

Tasks are "logically executing" if they could execute, provided enough processing resources were available. In a uniprocessor system, one of all the tasks that are "logically executing" will actually be executing on the physical processor.

The **Processor Management function** implements the assignment of physical processors to tasks that are "logically executing". The Processor Management function is invoked by other components of the runtime environment, in order to block and unblock tasks. It keeps a list of those tasks which are "logically executing" and uses this list, in conjunction with the priorities of tasks, to determine which task or tasks should be assigned to processors.

Note that there are a variety of approaches to selecting which tasks actually run at any given time.

3.4 Interrupt Management

If the underlying computing resource implements asynchronous events such as interrupts (in bare machines), signals (as in UNIX), or asynchronous system traps (as in VMS), the Ada runtime environment will usually contain an **Interrupt Management function** that reacts to these asynchronous events. In addition to truly asynchronous events such as timer interrupts, I/O interrupts, and hardware failures, the Interrupt Management function also reacts to events that are program synchronous (such as arithmetic overflow), but that are signalled through the same mechanism as truly asynchronous events.

In several contexts, the Interrupt Management function acts as an intermediary between the underlying computing resource and various other parts of the Ada runtime environment. For example, interrupts related to I/O devices are recognized and then passed on to the I/O Management function, by invoking the appropriate interrupt routine in the I/O Management function. Interrupts from hardware timers are passed on to the Time Management function. Those interrupts that are actually traps, signalling conditions that correspond to predefined Ada exceptions, are passed on to the Exception Management function. Spurious interrupts are handled locally in the Interrupt Management function.

If address clauses for task entries are implemented, the Interrupt Management function utilizes the Rendezvous Management function to realize interrupt rendezvous.

The Interrupt Management function is responsible for initialization of the interrupt mechanism of the underlying computing resource, and it is also responsible for resetting that mechanism after an interrupt has occurred, if the architecture of the underlying computing resource requires such resetting. (An example is the sending of an "end of interrupt" acknowledgment to an external interrupt controller.)

3.5 Time Management

The **Time Management function** consists of all those portions of the runtime environment that will support the predefined package CALENDAR and the implementation of delay statements. If the underlying computing resource offers enough functionality, the support of package CALENDAR is trivial. As in the case for predefined I/O packages, this document considers package CALENDAR, if it is included with a particular Ada program, as part of the runtime environment.

The support of delay statements depends on the characteristics of the underlying computing resource as well. It will usually include some form of bookkeeping of outstanding delays. This part of the Time Management function cooperates with the Rendezvous Management function in the implementation of select statements (in particular: timed entry calls and selective wait statements with delay alternatives).

3.6 Exception Management

Both predefined and user-defined Ada exceptions may be raised at any point during the execution of an Ada program. While user-defined exceptions can only be raised explicitly through a "raise" statement, predefined exceptions may also be raised because particular conditions are detected in the underlying computing resource.

Whenever an exception is to be raised, the **Exception Management function** is invoked. This function implements Ada semantics for exceptions: that is, it determines whether there is a matching handler for the exception at hand, and if there is one, it transfers control to the handler. If there is no matching handler, it invokes the Task Termination function to terminate the task at hand or the main program.

If the search for a matching handler involves propagating the exception out of the frame in which it was first raised, the Exception Management function simulates an "orderly return" of the frame that is thus completed. In order to do so, it may, for example, invoke the Dynamic Memory Management function and/or the Task Termination function.

If the exception is propagated out of an accept statement or out of the elaboration of the declarative part of a task body, the Exception Management function implements the raising of the appropriate exception in the rendezvous partner task or the creating task, respectively.

The Exception Management function may use static variables that are created at compile time, such as tables of exception handlers that are defined for a particular frame. It may also cooperate with the Call/Return function, in order to set up mechanisms for accessing such variables.

3.7 Rendezvous Management

The **Rendezvous Management function** implements the semantics of the Ada rendezvous concept. In order to do so, it utilizes variables that are internal to the runtime environment. These variables reflect, among other things, which tasks are blocked because they are waiting to rendezvous with other tasks, and what the exact circumstances of these wait states are. The Rendezvous Management function cooperates with the Interrupt Management function in the implementation of interrupt rendezvous, if interrupt rendezvous is supported by the runtime environment.

3.8 Task Activation

The Ada language allows for the dynamic creation of tasks. Whenever a task is created, two steps can be distinguished: First, one or several variables are created to represent a task object. These variables will be used during the lifetime of the task to reflect its current state, as well as its relationship with other tasks. At some point after the task object has been created, the execution of the new task has to be started. This is effected by the **Task Activation function**. The Task Activation function is invoked by the creator of a new task in order to start the new task's activation (which is defined as the execution of the declarative part of the task's body). The Task Activation function may also be invoked by the new task in order to signal the completion of that task's activation.

3.9 Task Termination

The Ada language includes a set of rules for the completion, termination, and abortion of tasks. The **Task Termination function** implements these rules. In order to do so, it utilizes variables reflecting task dependence that are maintained throughout the execution of the Ada program, and in particular by the Call/Return function and by the Task Activation function.

3.10 I/O Management

The **I/O Management function** consists of all those portions of the runtime environment that are provided for the support of input and output. This includes in particular all those functions that support predefined packages from Chapter 14 of the Ada Reference Manual. Whether these predefined packages themselves, as far as a particular runtime environment includes them, should also be considered part of the runtime environment, or as part of the Ada program that is being supported by the runtime environment, seems to be a philosophical issue. To have a well-defined terminology, and only for this reason, this document includes in the definition of the I/O Management function those predefined packages that are defined in Chapter 14 of the Ada Reference Manual.

On the other hand, the I/O Management function may, in one implementation, provide functions that are present in the underlying computational resource in a different implementation. The dividing line is again somewhat arbitrary. To have a well-defined terminology, and again only for this reason, this taxonomy includes in the definition of the I/O Management function only those components that are not already integral parts of the underlying computing resource.

The I/O Management function of a particular Ada runtime environment may include components that have no counterpart in Chapter 14 of the Ada Reference Manual. On the one hand, there may be components that are internal to the runtime environment, such as a page I/O handler if the runtime environment implements paging. On the other hand, an Ada implementation may offer the user additional I/O facilities which have no counterpart in the Ada Reference Manual.

3.11 Commonly Called Code Sequences

This category is somewhat of a "catch-all". It includes runtime routines in the classical sense: commonly called sequences of code. Typical examples are operations for multi-word arithmetic, block moves and string operations. Ada attribute calcuations also fall into this category.

3.12 Target Housekeeping Functions

Typically, there is a series of actions, called **Target Housekeeping functions**, associated with starting up and terminating the execution environment of an Ada program. Such actions include determination of the particular hardware and software execution environment, setting of variables identifying same, processor and interrupt initializations, and so on. Similarly, if a program terminates, control is typically returned to some surrounding software whose state must be reset upon program exit.

4. Summary

A runtime environment is defined by the common coding conventions, the data structure representations, and the predefined subroutines that support the execution of an application program. The runtime environment provides the abstract machine on which the application program executes. Traditionally, the runtime environment has been supplied jointly by the programming language compiler for the programming language of the application and by the executive on which the application executes. The executive provides the predefined subroutines that supply many of the functions and that manage many of the resources used by the application. The programming language translator provides the other elements of the runtime environments. It selects the data structure representations and enforces the common code conventions that support the remaining functionality of the application and that adhere to the interface provided by the executive.

Because Ada provides high level features for concurrent programming and for dynamic storage allocation, an Ada compilation system is responsible for providing all the elements of the Ada runtime environment. In addition to the compiler, which selects the data structure representations and enforces common code conventions, the Ada compilation system includes a runtime library of predefined subroutines that support all the features of Ada source programs which cannot be represented in code generated by the Ada translator. From this runtime library the Ada compilation system selects the appropriate subroutines to support the specific features of Ada used in an application. The set of selected subroutines constitutes the runtime system for that application. In theory, each application can have a unique runtime system configured for it. The size of the runtime system configured to support an application in Ada is influenced by the nature and functionality of the underlying computing resource. The runtime environment of an Ada compilation system may be able to accommodate an arbitrary number of interpretations of an application in Ada that comply with the Ada language standard. The more direct support that the underlying computing resource provides for the functionality of the features of Ada the smaller the configured runtime system needs to be.

Glossary of Terms

abstract machine - Set of capabilities provided by an instance of the runtime environment to an application program.

Ada compilation system - All the elements necessary to translate an Ada application program into an executable program; this usually includes the compiler, the linker, and the runtime library.

common code conventions - Set of rules to follow for implementing control and data structures in a software application.

Dynamic Memory Management function - that part of the Ada runtime environment which concerns itself with the allocation and deallocation of storage at runtime; it consists of a **Stack Management subfunction** that allocates and deallocates space on the stack and checks for stack overflow and a **Heap Management subfunction** that allocates and deallocates more random requests for storage and checks for adequate storage space in the heap.

Exception Management function - That part of the Ada runtime environment which implements Ada semantics for exceptions: that is, detection of an exception and selection of the appropriate handler if one exists; if there is no matching handler, it invokes the Task Termination function to terminate the task at hand or the main program.

executives - Set of capabilities, usually in software, to extend and share the bare embedded computing resources among one or more software applications.

generated program - The set of instructions and data in a machine executable representation that is directly produced by the compiler from the application program.

Interrupt Management function - That part of the Ada runtime environment which manages the response to asynchronous and synchronous events; examples of asynchronous events are timer interrupts, I/O interrupts, and hardware failures, while examples of synchronous events are arithmetic overflow and constraint error; acts as an intermediary between the underlying computing resource and various other parts of the Ada runtime environment.

I/O Management function - That part of the Ada runtime environment which supports input and output, including all of the functions that support predefined packages from Chapter 14 of the Ada Reference Manual.

operating systems - Set of capabilities, usually in software, to extend and share the bare general purpose computing resources among one or more software applications.

Processor Management function - That part of the Ada runtime environment which assigns physical processors of an underlying computer resource to tasks that are "logically executing"; it maintains a list of tasks which are "logically executing" and a list of tasks which are "blocked."

programming language compilation system - Set of programs that automatically translate an application program written in a high level representation into an application program in a machine executable representation; also known as a compiler.

Rendezvous Management function - That part of the Ada runtime environment which implements the semantics of the Ada rendezvous concept.

runtime environment - Set of all capabilities provided by three basic elements: predefined subroutines, abstract data conventions, and control structure code conventions.

runtime library (RTL) - Set of all the predefined routines in a machine executable representation that support all the functionality of an application program language that is not supported in code generated from application programs.

runtime system (RTS) - Set of predefined routines in a machine executable representation that is selected by the Ada compilation system from a runtime library to support functionality of the application program not supported in the generated program.

Target Housekeeping function - That part of the Ada runtime environment which is responsible for starting up and terminating the execution environment of an Ada program.

Task Creation and Activation function - That part of the Ada runtime environment which creates the storage and task management structures for a task, starts the execution of a task (task activation), and signals the completion of that task's activation.

Task Termination function - That part of the Ada runtime environment which supports the completion, termination, and abortion of tasks; it maintains task dependence for determine the extent of termination and abortion; it releases all storage to terminated and aborted tasks.

Time Management function - That part of the Ada runtime environment which supports the implementation of delay statements and the predefined package CALENDAR; if time slicing is provided by the Processor function, this function would provide the timing support.

underlying computing resource - The combination of bare computer and operating system/executive available to the runtime environment of a software application.

Distribution List for IDA Memorandum Report M-540

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John F. Kramer Program Manager STARS DARPA/ISTO 1400 Wilson Blvd. Arlington, VA 22209-2308	4
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
IDA	
General W.Y. Smith, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Mr. James Baldo, CSED	1
Dr. Richard J. Ivanetich, CSED	1
IDA Control & Distribution Vault	2