ADA* EVALUATION PROJECT

RUNTIME EXECUTION CONSIDERATIONS FOR ADA* SOFTWARE DEVELOPMENT

Prepared for

HEADQUARTERS UNITED STATES AIR FORCE
Assistant Chief of Staff of Systems for Command, Control, Communications, and Computers
Technology & Security Division

COMMAND & CONTROL SYSTEMS OFFICE

Prepared by
Standard Automated Remote to AUTODIN Host (SARAH) Branch
COMMAND AND CONTROL SYSTEMS OFFICE (CCSO)
Tinker Air Force Base
Oklahoma City, OK 73145-6340
COMMERCIAL (405) 734-2457 / 5152
AUTOVON 884-2457 / 5152

* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

27 March 1987

THIS REPORT IS THE SEVENTH OF A SERIES WHICH DOCUMENT THE LESSONS LEARNED IN THE USE OF ADA IN A COMMUNICATIONS ENVIRONMENT.

ABSTRACT

This paper discusses Ada run-time execution issues. Information is provided on execution speed, load handling ability, and static and dynamic memory considerations. The examples provided in the paper are based largely on run-time experiences gained through the development of the Standard Automated Remote to Automated Digital Network (AUTODIN) Host (SARAH) workstation.

The first section of the paper provides some background information of the Ada evaluation task and run-time execution. The scope and constraints of the paper are also addressed.

The second section of the paper deals with execution speed. The effect of the compiler implementation on execution speed is covered along with several methods that can be employed to reduce risk. In addition, the section focuses on design considerations that need to be addressed when developing real-time software.

The third section focuses on load handling ability. Several examples are provided to illustrate how Ada can be used to enhance load handling ability. In particular, the use of Ada tasking is addressed. Other issues that are covered in this section include the use of prototyping, experimentation, and the importance of effective device management on load handling.

The fourth section looks at static memory considerations. Several aspects of how static memory requirements can be reduced are covered. The question of compiler maturity is addressed as are Ada language issues.

Dynamic memory considerations are addressed in section five. Examples are provided showing how the SARAH designers dealt with the problem of memory fragmentation, memory allocation, and memory deallocation.

The final section provides a summary of the main points covered in the paper and provides specific recommendation on run-time execution.

Ada Evaluation Report Series by CCSO

Ada Training	March 13, 1986
Design Issues	May 21, 1986
Security	May 23, 1986
Micro Compilers	December 9, 1986
Ada Environments	December 9, 1986
Transportability	March 19, 1987
Runtime Execution	April 10, 1987
Modifiability	Spring 87
Project Management	Spring 87
Module Reuse	Fall 87
Testing	Fall 87
Summary	Fall 87

TABLE OF CONTENTS

Appendices

LIST OF FIGURES

1. INTRODUCTION

1.1. THE ADA EVALUATION TASK

This paper is one in a series which seeks to help potential Ada developers gain practical insight into what is required to successfully develop Ada software. With this goal in mind, Air Staff tasked the Command and Control Systems Office (CCSO) to evaluate the Ada language while developing real-time communications software. The task involves writing papers on various aspects of Ada development such as training, Ada design, environments, and security issues. This paper discusses the run-time execution issues.

CCSO chose the Standard Automated Remote to AUTODIN (Automatic Digital Network) Host (SARAH)[1] project as the vehicle basis for the Ada evaluation. SARAH is a small to medium size project (approx. 40,000 lines of executable source code) which will function as a standard intelligent terminal for AUTODIN users and will be used to help eliminate punched cards and paper tape as a transmit/receive medium. The development environment for SARAH consists of a number of IBM PC AT, Zenith Z-150, and Z-248 microcomputers. The source code produced is compiled on the PC ATs, and Z-248s using Alsys Ada compilers and the object code can be targeted to all three microcomputers. The SARAH software will run on a range of PC XT, PC AT, and compatible microcomputers under the MS-DOS operating system (version 2.0 or higher).

1.2. PURPOSE

The purpose of this paper is to:

- Discuss some of the language features provided by Ada to enhance run-time execution.
- Discuss some of the language characteristics that must be considered in the context of run-time execution.
- Discuss those run-time issues that had to be considered by the SARAH design team.
- Discuss some constraints caused by compiler immaturities.
- Provide design and coding recommendations to help the designer and coder enhance the run-time execution of their system given language features, language characteristics, and compiler constraints.

1.3. BACKGROUND

Efficient run-time execution is one of the major goals for the design and development of most software systems. Consequently, run-time execution requirements usually are, and should be, defined in the project requirements document.

The source of run-time execution requirements are the user's requirements and the constraints of the environment that the system will operate within. There may be time constraints, memory constraints, device constraints, and/or processor constraints.

The SARAH designers were confronted with all of these constraints to at least some degree. Since SARAH is a real-time on-line communications system, definite time constraints exist. The target environment is an IBM PC compatible (both XT and AT) and memory is limited to 640K. The system must manage a number of relatively slow devices (disk drives and printers), therefore numerous device constraints exist. Processor constraints result mainly from the 8088 target which runs at a slow 333,000 instructions per second.

In addition, as a communications terminal, SARAH terminals will be subjected to large variations in load. The system will often be online with no other activities; at other times there may be many simultaneous demands.

For these reasons, this paper is divided into chapters covering execution speed, load handling, program memory requirements, dynamic memory requirements, and device management.

Compiler maturity is a recurring topic throughout the paper. This is a consequence of the effect of less than completely mature compilers on the development effort and the related effect on the performance of the final product. The Reference Manual for the Ada Programming Language (commonly referred to as the LRM) does not specify the size or speed of the object code, or the relative execution speed of different language constructs. Therefore, purchase of a validated compiler does not, of itself, guarantee anything in the way of performance. Performance generally increases with the maturity of the compiler.

It should be noted, the advent of Ada has pushed the science of compiler development forward quickly. Ada compilers are much larger and more complex than most traditional compilers (300,000 lines of source code to implement an Ada compiler is not uncommon). Consequently, most compilers progress through a maturing process after their first release.

1.4. SCOPE AND CONSTRAINTS

The SARAH design team is developing a system to run under the MS-DOS (MicroSoft Disk Operating System). The Ada compiler that is being used for the project is the Alsys, Inc., AlsyCOMP_003 which is hosted on the IBM PC AT and selected compatible microcomputers. Much of our experience with Ada, especially in the area of run-time topics, is limited to this application. Some experience was also gained through early experimentation and training on a Burroughs XE550 (hosting the Telesoft Version 1.4 and 2.1 compilers), a Digital Equipment Corporation VAX 11/780 (hosting the SOFTECH Ada Language System), and an Intellimac IN7000 (hosting the Verdix Ada Development System).

....

2. EXECUTION SPEED

Execution speed is an important run-time characteristic that must be considered when designing and developing software. There are many factors that affect execution speed. For example, the maturity and quality of the compiler implementation, the design approach, the target hardware, and language features will all have a bearing on how fast the code will execute.

2.1. SARAH SPEED REQUIREMENTS

Execution speed is an important requirement for the SARAH system. The system must provide full duplex synchronous communications at 2400 baud, yet still allow the user to simultaneously perform message preparation/editing functions. One of the major execution speed criteria is that the host must not have to wait on SARAH when it is trying to transmit data. Execution speed should be sufficient to satisfy this basic requirement, yet still allow the user to prepare and edit messages without noticing the effect of the background communications task.

A major factor that must be considered when addressing speed requirements is the target hardware. SARAH is designed to run on a range of IBM PC XT and PC AT compatible microcomputers. More specifically, the requirements identify the Zenith Z-150 (a PC XT compatible computer) and the Z-248 (a PC AT compatible computer) as the initial targets. Execution speed that is acceptable on a Z-248 may appear very slow and sluggish on the less powerful Z-150 system.

Many projects use a 'software first' approach, where the hardware is selected after the software requirements have been established. This is a preferable approach because the software design need not account for any hardware limitations. However, for the SARAH project, the establishment of a standard microcomputer contract ensured that there would be a large community of Z-150 users. As such, the SARAH developers were tasked with ensuring that the SARAH application would exhibit satisfactory execution speeds on the Z-150 system. This is a major challenge.

2.2. COMPILER ISSUES

The quality of code produced by the compiler will have a large bearing on execution speed. As such, the project team needs to have a thorough knowledge of the characteristics of the compiler to be used. There have been many questions as to how Ada code compares with code produced by compilers of other languages. Another common question has been: "How do we determine whether the code will execute fast enough for our application?" These issues will be discussed in this section.

2.2.1. Language Comparisons

Code produced by Ada compilers can be as fast as that produced by compilers of other modern High Order Languages (HOLs). Benchmark tests comparing Ada against 'C' and Turbo Pascal on a IBM PC AT show that Ada code can be at least as fast as the other languages, and in many cases much faster.[3] The results of these tests were obtained using a non-optimizing Ada compiler; therefore, further improvement can be expected. A great deal of research and development effort is being applied to the question of Ada code optimization. As Ada compilers mature, the quality of code produced should be as good as, if not better than, the code produced by compilers of other languages.

2.2.2. Speed Optimization

The complexity and size of the Ada language has created problems for compiler developers and so the Ada community has had to wait patiently until suitable compilers were made available. Most of the early compilers did not employ code optimization and many compilers produced code that showed poor execution speeds. Developers are now beginning to 'fine tune' their compilers and provide code optimization. For example, Alsys Inc. has been continually refining their PC AT compiler since its release. Low level optimization will be included in their Version 3 release, due in fourth quarter of 1987.

The Ada language itself provides some problems for vendors intending to provide code optimization. Several papers[12,13] have been written on Ada optimization problems. Some of the Ada features that cause problems for optimizers are exception handling, separate compilation, tasking and generics. Restrictions imposed by the language on limiting the bounds for reordering to the innermost enclosing frame can hinder the developer in providing some very profitable optimizations. To illustrate, consider the following code segment:

```
begin

   for i in 1 .. 10 loop
      begin
         Total := Total + i * Factor(Int_Value);
      end;
   end loop;

   --some other code

end;
```

The calculation for Total is within the inner frame. A great deal of execution time could be saved if, during optimization, the function Factor(Int_Value) could be moved outside the loop and calculated just once. Ada forbids this and so, in this case, a very beneficial optimization is lost. If in-line inclusion of subprograms is used to increase execution speed, there would be many situations where inner frames (such as the one described in the example) would be used. As such, many of the speed benefits would be lost because of the restrictions imposed on optimization. The future will tell how well these problems can be overcome and whether Ada can produce superior code for real time applications.

2.2.3. Benchmarking

Benchmark testing can establish whether or not a particular compiler will produce code that will allow the application to execute at an acceptable speed. Benchmarking is important for a number of reasons:

- **Compiler Selection.** Benchmarking allows developers to select a compiler that best meets the needs of the project.

- **Identifying Compiler Strengths and Weaknesses.** Benchmarking gives the developer a measure of code performance which can be used for making well founded design and development decisions concerning execution speed.

The information provided by benchmark tests can have a large affect on the overall design and development approach. For example, the Ackermann's function[8] gives a good indication on how efficiently procedure calls are implemented. If the results of this test are poor, then consideration needs to be given to designing a system where procedure calling will not adversely affect execution speed. Benchmark test programs can be obtained through several sources.[10]

Although benchmark tests can be valuable for determining compiler characteristics, performing the tests 'in house' may take a considerable amount of time and expertise. As such, developers may be better advised to look at published benchmark reports.[2,3,5] Organizations such as the Special Interest Group on Ada (SIGAda) Performance Issues Working Group are compiling benchmark test results that may prove valuable.

2.3. DESIGN CONSIDERATIONS

Software design can have a major effect on execution speed. Designers need to look at various trade-offs and these decisions need to be based on solid facts and good judgment. Sound design decisions can only be made if the designers are aware of the problems and limitations they may face; this information can be obtained through experimentation, prototyping, and benchmarking. Some of the design considerations that must be made when developing Ada applications where execution speed is important are: the effect on transportability, the use of tasks, and the effect of the design structure.

2.3.1. Transportability

A design trade-off that needs to be considered is transportability versus execution speed. To enhance transportability, the SARAH designers attempted to use the standard predefined packages wherever possible. However, in some cases, their use would have seriously affected execution speed. For example, early experiments showed that Text_IO would not be satisfactory for the SARAH user interface. The user interface was designed to consist of a number of windows and pull down menus (see Fig 2.1). The Text_IO routines proved to be too slow and lacked the functionality needed to implement this type of interface. As such, an independent VDT_Manager which directly accesses screen memory was designed. This approach is not as transportable as would have been the case had Text_IO been used, but was a necessity because of the execution speed requirements.

2.3.2. Use of Tasking

Most current implementations of the Ada tasking model are inefficient. As such, when designing Ada applications where execution speed is important, tasking effects need to be considered. For example, Burger and Nielsen[9] show that using DEC Ada (version 1.2) on a VAX 8600, a simple rendezvous between producer and consumer tasks takes 503 microseconds, whereas a simple procedure call takes 11 microseconds. Use of procedures instead of tasks (in cases where possible) may be a trade off that needs to be considered when performance issues are being addressed. Two design methodologies that specifically look at this issue are the Process Abstraction Methodology for Embedded Large Applications (PAMELA)[14] and the Modular Approach for Software Construction, Operation, and Testing (MASCOT3).[15] PAMELA provides a number of rules for identifying whether a task or procedure should be used for a particular process.

The first design of the SARAH Communications subsystem used a number of tasks. During an early design walk-through tasking effects were addressed. The designers used tasks to interface with the low-level communications driver and intended to pass one character at a time to the driver by way of task rendezvous. The interface was to operate at 2400 baud and so a character would have to be received (and perhaps transmitted) at intervals of just over 4000 microseconds. At the time, Alsys Inc. indicated that a simple rendezvous consisting of two context switches would take about 3000 microseconds. When the overall system overheads were considered, a quick calculation showed that SARAH would not even be able to receive data at the required rate, much less do any of the other functions that were required. Armed with this information, the designers reconsidered their approach to using tasks. The number of tasks and rendezvous were reduced.

2.3.3. Design Structure

Ada has many language features that complement modern software engineering practices. For example, Ada packages are useful for developing systems that make good use of abstraction and information hiding. The use of these features can produce well structured designs which aid software understandability, modifiability, and hence, maintainability. Design methodologies such as Object Oriented Design (OOD)[7] provide a 'cook book' approach to establishing this type of design structure. Although these designs may be theoretically correct, the overall design structure should be analyzed thoroughly to determine how the structure might impact execution speed.

Once the structure is defined and the concurrent modules are identified, a design walkthrough should be conducted to estimate execution requirements and to set task priorities. The results of previous experiments, benchmark tests, and simulations can be important for estimating how much processor time should be allocated to the various modules. For identifying concurrent modules, the SARAH designers made use of the concurrency view of multi-view design approach.[11] By performing design walk-throughs, potential execution problems can then be determined early in the development cycle.

2.4. ADA LANGUAGE FEATURES

The language itself plays a large part in how fast the code will execute. Language features can be used to advantage to improve execution speed; however, there are also those features that tend to slow execution speed.

One Ada language feature that can be used to improve speed is the access type. If a compiler implementation used a 'pass by value' scheme for parameter passing in procedures and functions, a significant speed overhead can be imposed when passing large data objects. Rather than passing the complete object, an access object can be created as an access type. Reference to the object is then made by way of an access value (a pointer). For example, if we had a large amount of data to be passed as a parameter in a procedure, we could define the data object as an access type. Instead of passing all the data each time the procedure was called, a pointer to the data object could be passed. This approach can significantly reduce the overhead associated with passing large amounts of data as parameters in functions and procedures where the compiler uses a 'pass by value' scheme for parameter passing.

Run-time checks in Ada have a negative effect on execution speed. Ada has a number of predefined exceptions which may be raised implicitly. The run-time checking associated with these exceptions imposes a run-time overhead. The effect of run-time checks on execution speed varies with each compiler implementation. If there is some compelling reason to do so, the run-time checks can be suppressed using pragma SUPPRESS. Benchmark tests performed with the Alsys PC AT compiler[3] show that there is a significant increase in execution speed when the programs are run with 'checks off.'

In addition to the predefined exceptions, Ada allows for user-defined exceptions; however, ideally, these should have little effect on execution speed unless the exception is raised. The Ada designers were careful in defining user defined exceptions so that no overhead would be imposed on execution speed.[12] They even provided a scheme that could be used to efficiently implement exceptions.[17] The Language Reference Manual does not specifv how exceptions will be implemented and so there is no guarantee that the compiler will not impose a speed overhead. Developers need to check on the efficiency of exception handling and the effect that exceptions may have on execution speed.

Another Ada feature that can cause problems if not properly used is generics. Execution problems can arise if care is not taken in how and where generic units are instantiated. For example, if a generic is defined inside one of the procedures, the generic will be instantiated each time the procedure is called. If the procedure is called many times, then severe performance problems will result.

....

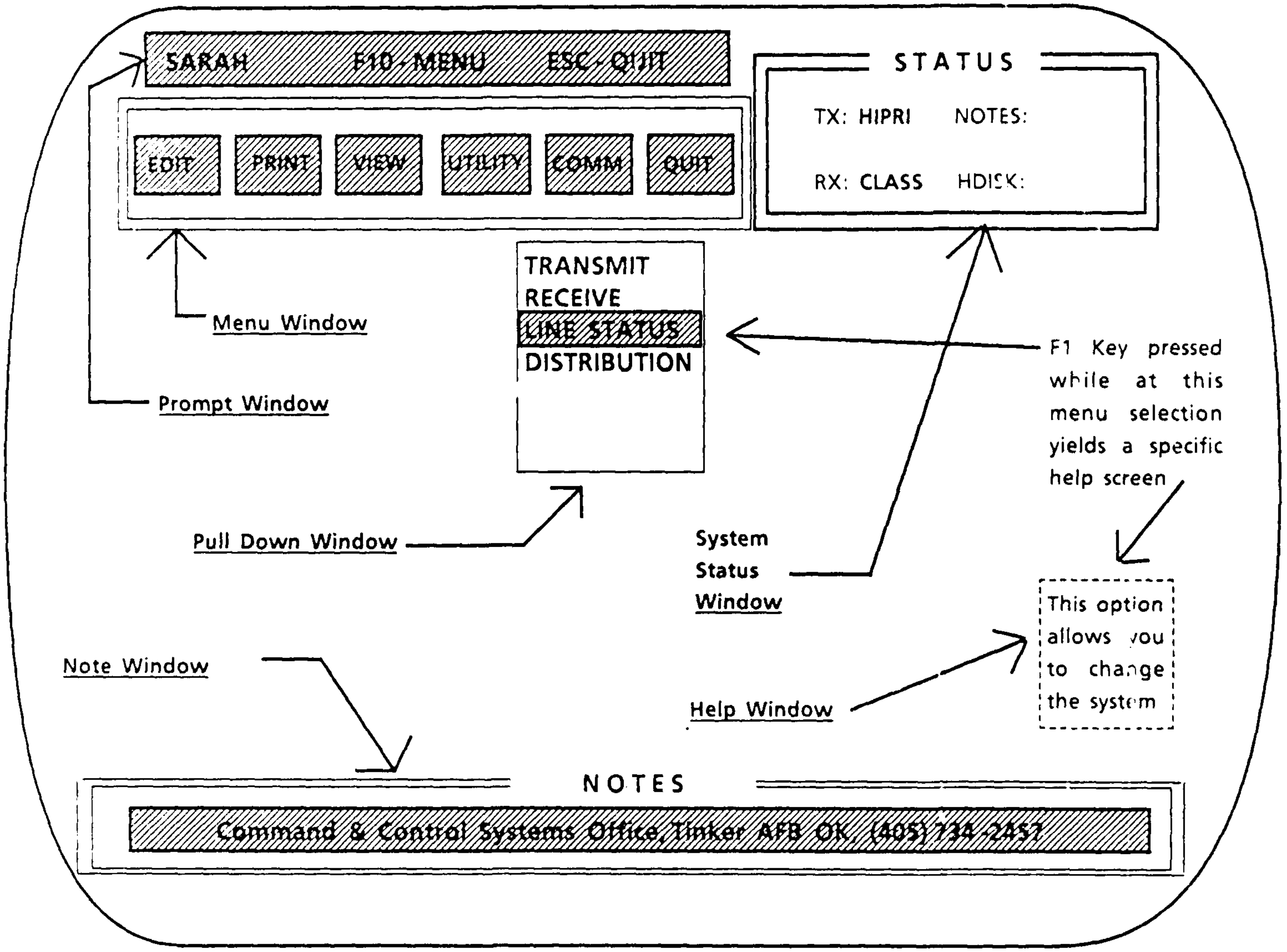

Figure 2-1. User View. Window Naming Conventions

3. LOAD HANDLING ABILITY

Load handling ability refers to the ability of a system to handle a number of active tasks simultaneously. These tasks are not necessarily Ada tasks and need not be implemented in software. For example, the printer could perhaps be seen as a task. The application software communicates with the printer, loads the print buffer, and then continues to execute in parallel with printer operation. When considering load handling, all aspects of the system need to be considered, not just the software. Indeed, device management plays a large part in load handling. Load handling ability is an important consideration for digital communications. These systems are required to simultaneously transmit and receive data while providing other functions such as queuing, print operations, data storage, and data retrieval.

3.1. SARAH LOAD HANDLING REQUIREMENTS

For the SARAH application, effective load handling is crucial. SARAH employs multitasking to allow for concurrent message preparation/editing, communications functions, and printing. The loading imposed by the number of active tasks and the load changes that occur during the day must be considered. In addition, the loading effects imposed by the SARAH peripherals must be considered. The communications version of SARAH must manage the resources of two printers, two floppy disk drives, a winchester drive, and a full duplex synchronous communications interface (see Fig 3-1).

3.2. HARDWARE EFFECTS

The target hardware has a major effect on load handling ability. The initial requirement for SARAH specified the Zenith Z-150 as the target microcomputer. Since then, the requirements have been changed to include the more powerful Z-200 and Z-248 microcomputers. In future, SARAH may be required to run on the Z-386 which uses the very powerful 80386 microprocessor. The ability of the SARAH application to handle load will improve with each generation of hardware. However, designers must be careful to ensure that the software will operate effectively across the full range of target hardware.

3.3. DEVICE MANAGEMENT

3.3.1. Language Features That Support Device Management

Ada tasking can be used to advantage in device management for improved load handling. There are several features of the Ada tasking model that can be used to advantage. For example, timed entry calls can be used to communicate with device driver tasks so that information exchange only need occur when the device is ready to accept or provide data. If this technique is employed, the system does not have to wait on the slower devices. Also, task priorities can be applied to help fine tune the system for optimum throughput. Effective use of task constructs can aid in device management and hence load handling.

Ada tasks can be used for device allocation and deallocation. A simplified example of using a selective accept in a task to control resource pool allocation[5] follows:

```
select
        when Resource_Available =>
                accept Allocate do
                        -- grant resource to calling task
                end Allocate;
                -- if resources are exhausted set
                -- resource_available to false
or
        accept Free do
                -- return resource to pool
        end Free;

        -- set Resource_Available to true
end select;
```

The selective accept allows rendezvous to either the "Allocate" entry point or the "Free" entry point when the boolean object "Resource_Available" is true. If Resource_Available is false, only rendezvous to Free are allowed (Allocate is a "guarded accept"). The above construct can be particularly useful when working with operating system device drivers that are not re-entrant (like MS DOS). In these cases, calls to access devices may not be queued against the device drivers, they must be limited to one at a time. Other, more complex, examples of how tasks can be used for controlling resource allocation can be found in various other publications.[18,9]

3.3.2. Device Allocation Priorities

As demonstrated, Ada tasks can be used for the allocation and deallocation of resources; however, many applications need to be able to prioritize the device management. For example, as discussed, one of the major speed and load handling requirements for SARAH is that the host computer must not have to wait on SARAH when the host is transmitting data. SARAH must therefore place the highest priority on receiving the data and storing it to the winchester disk. This means that when device allocation is considered, the writes to the winchester disk must take precedence. Moreover, multiple requests for disk drive access may be pending at any time. The SARAH Disk_Manager subsystem utilizes a disk resource control task which controls the priority and disk resource allocation for the floppy drives and the winchester drive.

Ada does not provide an effective mechanism for allocating priorities to task entry points. However, task entry families[9] can be used to construct a prioritizing scheme. Many of these techniques have been used in the development of the device management routines for SARAH. Designers can meet load handling requirements (particularly those associated with devices) by using tasks for device management and assigning appropriate priorities to task entry points.

3.3.3. Task Switching

The method employed by the compiler for task switching can play a large part in how well the system will handle load requirements. Many compilers do not currently implement time-slicing for switching control between tasks. Instead, these compilers employ a scheme where the tasks switch only at synchronization points (i.e. at rendezvous points, delays, and task activation). This can severely affect device allocation and load handling because, if tasks are used to manage the resources, some devices may get a much higher proportion of the total execution time than was intended by the designers. Delay statements can be strategically placed in the code to simulate a true time slicing scheme. To illustrate the problem, consider the device allocation problems in SARAH. The SARAH workstation is required to provide background printing as well as providing communications. However, the communications task is a much higher priority than the printer task. The system must be tuned to comply with the communications throughput requirements but must still allow sufficient resources to provide printing and message preparation/editing facilities.

3.4. <u>THE IMPORTANCE OF DESIGN</u>

Design is important for load handling. If the application is to handle the load requirements, this must be considered during the design phase. One of the more important characteristics of load handling is throughput. Throughput can be defined as the number of transactions that can be accomplished per second.[21] Careful consideration to module throughput during the design phase can provide significant load handling benefits.

Load handling should be considered early in the design phase. When considering throughput, all aspects of total system design should be considered, not just the software elements. For example, load handling ability could be improved by the use of additional processors or by using different peripheral devices. A system may be required to provides full duplex synchronous communications and so must continually output data or synchronization characters. This puts additional load on the main processor. Early in the design phase, consideration should be given to using an external hardware device which would provide the synchronous communications interface and so reduce these loading effects. This approach was used for SARAH, where a communication board with its own processor was used to provide synchronous communications.

3.5. PROTOTYPING AND EXPERIMENTATION

Prototyping and experimentation can aid in the development of real time systems where load handling is an important consideration. Without knowledge of how well the compiler implementation supports various language features and how the hardware will perform in different situations, designers and programmers will have difficulty in developing a system which will effectively handle load requirements.

Prototypes can be used to test loading effects. Once the basic design structure has been established, prototypes should be developed to test for throughput and load handling ability. This is particularly important if tasks are used in the system. Task priorities need to be established and the tasking implementation checked to see if it will support the load requirements. Compiler problems and implementation specific details need to be identified. These problems can provide developers with some rather challenging and expensive surprises if they are not identified early in the development cycle. Many of the problems that are found during the coding and testing phases can be identified during the design phase if prototyping and experimentation are employed.

3.6. DEVELOPMENT AND CODING ISSUES

The low-level design and coding phases can have a big effect on load handling. Even though good design techniques may be used and the macro-structure of the software fully supports the load handling requirements, decisions made by the programmer who is coding the design could have an enormous impact on throughput and load handling ability. Programmers and designers need to be well trained in the use of language features and structures to gain maximum performance from an application. Indeed, the development team needs to know which language features support load handling and how these should be used.

The way in which a programmer uses language features to implement the design is important. Consider the example of a module that provides disk read and write facilities for other parts of an application. The designer may have considered all the problems that could impede load handling ability and correctly established read and write priorities for the different devices. When the programmer codes the design, instead of writing blocks of data to the disk, the system is set up to update the disk one character at a time. The large amount of additional processing and the additional disk accesses can place a considerable overhead on the system. There are many decisions that must be left until the low-level design and coding phases. These decisions can have a major effect on load handling.

....

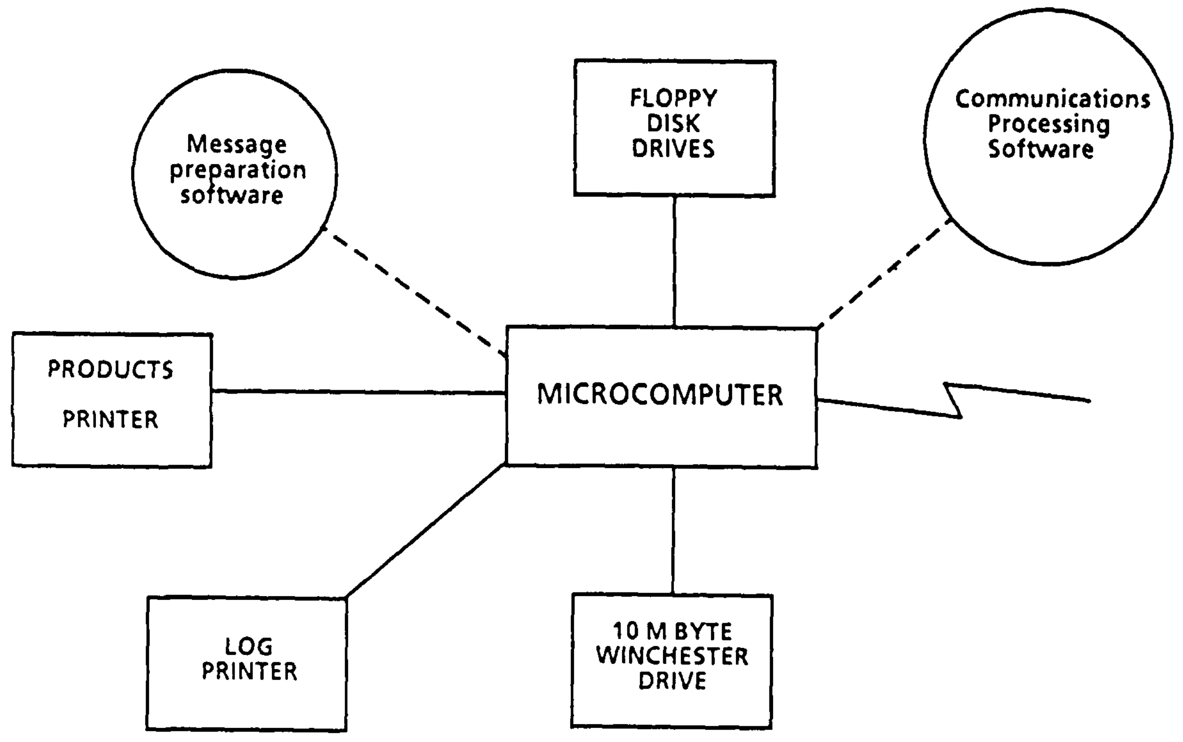
Message preparation software
FLOPPY DISK DRIVES
Communications Processing Software
PRODUCTS PRINTER
MICROCOMPUTER
LOG PRINTER
10 M BYTE WINCHESTER DRIVE

Figure 3-1. SARAH COMMUNICATIONS WORKSTATION

(BASIC WORKSTATION)

4. STATIC MEMORY CONSIDERATIONS

4.1. SARAH's MEMORY CONSTRAINTS

SARAH is currently targeted to the IBM PC XT and compatible microcomputers and is therefore restricted to 640 Kilobytes of total memory. This must be sufficient to hold the entire SARAH executable file (about 300 Kilobytes, resulting from about 17,000 lines of source code for the first prototype) and provide sufficient dynamic storage to allow the system to function. Memory usage is a definite concern within the SARAH project; the first prototype is already pushing this memory limit.

4.2. LANGUAGE CHARACTERISTICS

Ada executable files require more memory than other traditional languages. A size difference of 20%-30% can be expected and is attributable to characteristics of the language.

Ada is a large language. Included in the language are numerous predefined library units (mostly packages) and many of them are large.[7] In addition, the Ada run-time environment has additional work to do in the way of cross checks, type checks, and error checks. For these reasons, the Ada run-time environment is larger than in other languages and the size of the run time is usually the reason for the size difference of the executable files.

4.3. COMPILER ISSUES

"Immature" Ada compilers may generate executable files 100% larger than the same application in FORTRAN or other traditional languages. Excessive size of the compiler's run-time environment and/or excessive source-to-object code expansion are usually the culprits. For example, during Ada Expo'86 (held at Charleston WV in November 1986), Northrop Wilcox provided information on problems that they had encountered during the development of the Manoeuver Control System. Some of the system software was to be targeted to a embedded computer system which could support one megabyte of ROM memory. The compiler that they were using produced 54 bytes of code for each Ada source line. Since the application consisted of some 34,000 lines of source code, the 1,836,000 bytes of object code produced by the compiler was well above the 1,000,000 bytes that the target computer could support. The company bought a more mature compiler which provided a much improved expansion ratio. The code was then able to fit within the available memory constraints and the system was delivered to the customer.

Ada compilers are beginning to produce more efficient code. However, there are many areas that compiler implementors should concentrate on when tuning and enhancing their compilers for more efficient memory utilization. Some of these are covered in the following sections.

4.3.1. Smart Binders

Smart binders are capable of excluding those parts of the run-time library not needed for a particular application. For example, the part of the run-time library that supports tasking is large. If tasking is not used in the application it should be left out. The Alsys compiler, used by the SARAH team, does not make this determination; but, it provides a similar effect by allowing the programmer to choose a non-tasking library when the development library is created.

4.3.2. Smart Linkers

Smart linkers link only those parts of library packages that are actually used by the application. Some compiler developers are already working at providing this feature. Some of the reasons for this requirement are that both the standard predefined packages and packages reused from repositories contain program modules and types which may not be used by an application. An example is the pre-defined package "Text_IO". Text_IO is an extremely large and commonly used package. On most current implementations, when Text_IO is made visible, the entire package is linked into the run time regardless of how many (or how few) Text_IO functions and procedures are called. On the Alsys compiler, the use of Text_IO adds about 26,640 Kilobytes bytes to the size of the executable file.[10]

To reduce system memory requirements, the SARAH designers elected to do without Text_IO. Instead, for disk accesses we used the predefined package Sequential_IO (which requires 9600 Kilobytes plus 1056 each time it is instantiated), and for accessing the screen we wrote our own packages (we call it VDT_Manager). This approach appears to work well; but, because of bugs in the current implementation the Alsys Sequential_IO package (version 1.3) and the memory constraints of our target, the use of Sequential_IO was eliminated for the first SARAH prototype. The vendor-supplied predefined package "DOS" (non-transportable) is currently being used for disk accesses.

Some compiler implementors have attempted to circumvent the Text_IO problem by supplying their own subset of Text_IO. For example, the Meridian's compiler comes with the package "Ada_IO."[19,22] Alsys[20,23] also includes some extra environment packages not defined in the Ada Language Reference Manual, they are: DOS, DOSE, and Unsigned (these can be used for Input/Output, MS DOS error control, and low-level work).

In the interest of transportability (One of the major goals of the language), use of these packages should be avoided when possible. When vendor supplied packages are used in an application, the application is then tied to the vendor's compiler. The system will not compile on other compilers without rewriting those parts of the system which utilize the vendor packages or writing a package which emulates the vendor package.

4.3.3. Efficient Generic Instantiations

A new copy of generic package bodies is linked into the run time each time they are instantiated with some current implementations. This problem should diminish as new compiler versions are released. More sophisticated versions will be smart enough to utilize most of a single body of a generic, even if it's instantiated with various different types.

4.3.4. Virtual Memory Techniques

The use of virtual memory techniques, such as overlays and paging, is a traditional method used to reduce memory requirements of the executable portions of larger software systems. Their use, however, is not defined in the Language Reference Manual [6] and, to our knowledge, none of the currently released compilers employ virtual memory techniques. Some compiler developers may eventually provide virtual memory capabilities.

Arguments against using virtual memory methods include:

- o System slow-down if the virtual memory area is a device (e.g. a disk drive).
- o They are no longer required as they were a few years ago since memory is now relatively cheap.

Unfortunately, cheap memory does not help the SARAH team with the problem of trying to fit the system into a Z-150. MS DOS on the Z-150 generally limits addressable memory to 640 Kilobytes.

....

5. DYNAMIC MEMORY CONSIDERATIONS

5.1. SARAH's DYNAMIC MEMORY CONSTRAINTS

Available dynamic (or working) memory in SARAH is limited to what's left over after the operating system loads (MS DOS), the executable file loads (about 300 Kilobytes for the SARAH prototype), and the run time initializes during elaboration. Since the total memory available in a PC XT compatible is 640 Kilobytes, memory availability is a definite constraint with SARAH.

Besides the dynamic memory requirements of the Alsy 'a run time, the SARAH application requires memory to read message files from disk, accept message files from the communications line, create messages in the editor, maintain the printer queue, etc.

5.2. LANGUAGE FEATURES FOR DYNAMIC MEMORY CONTROL

Ada provides features needed to conduct run-time allocation and deallocation of memory. Memory can be explicitly allocated for access objects with the "new" statement. Memory is explicitly deallocated by use of the generic procedure "unchecked deallocation." (Memory is implicitly allocated and deallocated through subprogram calls, subprogram recursion, and run-time creation and termination of task objects.)

5.3. GARBAGE COLLECTION

Garbage collection is not implemented on most of the currently available Ada compilers (garbage collection is the very resource intensive process of packing allocated memory to prevent loss of usable memory due to memory fragmentation caused by run-time allocation and deallocation). Garbage collection is not defined in the Language Reference Manual and most implementors have chosen to concentrate on those features which are. Unfortunately, because of the way the language is defined, operating systems do not have the information available to them required to conduct garbage collection. Lack of garbage collection can be a problem for many applications, particularly those that are on-line for extended periods of time when fragmentation problems tend to multiply.

A couple of compiler implementors do provide garbage collection. Others are expected to follow as the compiler maturing process continues. Many implementors choose not to employ garbage collection because of the high overheads (processor time) involved.

5.4. MEMORY MANAGEMENT WITHOUT GARBAGE COLLECTION

The SARAH designers implemented a memory management package to work around the lack of garbage collection (called

Buffer_Manager). Since a requirement of SARAH workstations is to remain on-line for extended periods of time, it's especially important that available dynamic memory not be diminished by fragmentation. One way to avoid memory fragmentation when allocating and deallocating access types is to use a common buffer size. The SARAH Buffer_Manager accomplishes this. Buffer_Manager is the central point for allocation and deallocation of buffers in the system; all buffers allocated are of the same type and consequently the same size (To handle variations in buffer requirements, buffer types are defined as a variant record.).

There are other ways to get around the problem of not having garbage collection. Garbage collection can be conducted using Ada constructs. One way of doing this is to load all the memory (available for dynamic allocation to access objects) into a huge array at boot time and then allocate and deallocate from the array. If the application packs the array to eliminate empty holes, memory buffers can be provided in any size increments. A warning: this is no small undertaking, besides requiring a complex set of algorithms, a large portion of processor time will be required. A second method is to employ a free-list approach[2].

5.5. COMMON TOOLS PACKAGES

A common tools package is a functional grouping of commonly used tools into a library package to reduce memory requirements. The package Buffer_Manager referred to in the above section is also good example of a common tools package. Tools are provided in Buffer_Manager to build and manipulate linked-lists of buffers. Use of this common tools package reduces the memory requirements over what would be required if system components conducted their own linked-list manipulations. This also helps to ensure the integrity of internal lists by using consistent list control algorithms. This central buffer control arrangement also provides an ideal environment for policing buffers. If memory runs short due to one of the system components failing to turn in its buffers, the culprit should be easy to identify if Buffer Manager maintains a record of who checked out the buffers.

Common tools packages also support the software engineering goals of modifiability, modularity, abstraction, information hiding, and uniformity.

....

6. SUMMARY AND RECOMMENDATIONS

6.1. SUMMARY

Compiler issues are a major consideration for run-time execution. The maturity of Ada compilers remains a major issue for developers. Most current compilers do not provide an efficient task model implementation. As such, many developers are careful in their use of tasks. There are several guidelines that should be established for the use of tasking. For example, because of the inefficiencies associated with task communications, task rendezvous should be kept to a minimum. In addition to problems with tasking, many compilers produce large executable object files and provide little support for dynamic memory management.

Prototypes, experimentation, and the use of benchmark test results can help reduce the risks associated with run-time execution. Developers need to be aware of compiler and language limitations, and the characteristics of the target hardware and peripheral devices. Prototypes can help establish whether a particular implementation will be able to handle the load requirements. Benchmark test results are becoming available for various compiler implementations and can be very beneficial for determining the best approach for developing real-time software where run-time execution requirements are important. If prototypes and test results are used early in the development process, risks associated with run-time execution can be eliminated and so save expensive software re-works during the coding and testing phases.

Good design practices, together with a thorough knowledge of the target environment and development tools, help to eliminate potential problems early in the development cycle. There are several design methods that can be used to enhance run-time characteristics. For example, the PAMELA methodology addresses task utilization and object-oriented methods can be useful for developing flexible systems where module interfaces are minimized. The software design should be flexible enough to allow for 'fine tuning' during later stages of development. However, the manner in which the 'fine tuning' will occur needs to be planned during the design. There are many trade offs that must be considered during the design phase. For example, a trade off between transportability and execution speed may need to be made. These decisions need to be made early and be based on firm requirements.

Ada has many language features that aid in enhancing run-time characteristics. For example Ada tasks can be used for device management and to enhance load handling ability. Use of access types can enhance both storage and speed characteristics of the system. The Ada language is a very rich and complex language, and although programmers can write code after a short training period, considerably more time and experience is needed to design and develop systems that will execute efficiently in a real-time environment.

Several features of the Ada language are an impediment to run-time execution. Ada has many features that have not been available in the more traditional languages such as FORTRAN, COBOL, and Pascal. Features such as exception handling, tasking, and generics have caused problems for compiler optimization. Some of the problem lies in the complexity of the language and the lack of experience in providing optimizations for these more advanced features. Future Ada compilers should produce more efficient code than is currently being produced. However, some constraints on optimization are directly attributable to the limitations imposed by the language designers.

....

6.2. RECOMMENDATIONS

Based on our experiences in developing the SARAH system, there are several recommendations that should be made regarding run-time execution. These are:

- Use good design practices.
- Experiment and develop fast prototypes before deciding on the final design structure.
- Identify compiler strengths and weaknesses to facilitate 'fine tuning' for run-time performance.
- Ensure that the run-time execution characteristics are considered during the design phase.
- Use common buffer sizes to work around the lack of garbage collection if your compiler does not have it.
- Employ common tools packages to reduce memory requirements by reducing redundant routines.
- Avoid instantiating generics inside packages, subprograms, or tasks.
- Avoid using tasks when subprograms can be used as effectively.
- Minimize the number of task rendezvous.
- Know your operating system and/or hardware limitations (device drivers re-entrant?, etc.).

....

A. REFERENCES

[1] "SARAH Operational Concept Document," Command and Control Systems Office, US Air Force, 5 September 1986.

[2] " Ada Portability Guidelines," National Technical Information Service, No. AD A160 390, March 1985.

[3] BROGSOL B., AVAKIAN A.S., GART M.B., "Alsys Ada Compiler for the IBM PC AT," Proceedings of First International Conference on Ada Language Applications for the NASA Space Station, June 1986.

[4] BARBACCI M.R., HABERMANN N., SHAW M., "The Software Engineering Institute: Bridging Practice and Potential," IEEE Software, Nov 1986, pp 4-21.

[5] WEIDERMAN N., HABERMANN N., et al, "Evaluation of Ada Environments: Executive Summary Chapter 1 Chapter 2," Software Engineering Institute, August 1986.

[6] U.S. Department of Defense, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD 1815A, Jan 1983.

[7] BOOCH G., "Software Engineering with Ada," Benjamin/Cummings, Menlo Park, California, 1983.

[8] WICHMANN B.A., "Ackermann's Function in Ada," ACM Ada Letters Vol VI No 3 (May/June 1986), pp 65-67.

[9] BURGER T.M., NIELSEN K.W., "An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada," ACM Ada Letters Vol VII No 1 (Jan, Feb 1987), pp. 49-58.

[10] "Usage and Selection of Ada Microcomputer Compilers," Command and Control Systems Office, Tinker Air Force Base, Oklahoma, 9 December 1986.

[11] "Architectural Approach to the Design and Development of Ada Software," Command and Control Systems Office, Tinker Air Force Base, Oklahoma, 21 May 1986.

[12] BAKER T.P., RICCARDI G.A., "Implementing Ada Exceptions," IEEE Software, September 1986, pp.42 -51

[13] KIRCHGASSNER W. et al, "Optimization in Ada," Ada Letters, Vol.3, No.3, Nov - Dec 1983, pp. 45-50.

[14] CHERRY G.W., "The PAMELA Designer's Handbook," Thought Tools, Reston VA.

[15] "Special Issue on MASCOT," Software Engineering Journal, IEE Savoy Place London, Vol 1 No 3, May 1986.

[16] NISSEN J.C.D., WALLIS P.J.L., WICHMANN B.A., et al, "Ada-Europe Guidelines for the Selection and Specification of Ada Compilers," ACM Ada Letters Vol III No 1 (July-Aug 1983) pp 27-50.

[17] ICHBIAH J.D., et al, "Rational for the Design of the Ada Programming Language," SIGPLAN Notices, 14.6B, June 1979.

[18]. A. J. Wellings, D. Keeffe, G.M. Tomlinson, "A Problem with Ada and Resource Allocation," 25 October 1983, Ada Letters, Jan/Feb, 1984.

[19] Meridian Compiler User's Manual, January 1987.

[20] Alsys Compiler User's Manual, July 1986.

[21] Boris Beizer, "Software Performance," Handbook of Software Engineering, Van Nostrand Reinhold Company, 1984.

[22] Bruce A. Bergman, William H. Murray, & Chris H. Pappas, "Ada Compilers: Mission-Critical Software for the PC - Part 1," Computer Language, Dec 1986,

[23] Bruce A. Bergman, William H. Murray, & Chris H. Pappas, "Ada Compilers: Mission-Critical Software for the PC - Part 2," Computer Language, Jan 1987,