The Impact of Machine Architecture on the Implementation of the SAS® System
Larry J. Noe, SAS Institute Inc., Cary, NC

INTRODUCTION

The responsibility of each of the host development groups at SAS Institute is to implement a host-dependent layer of code that provides low-level services to a portable supervisor. Each host group is generally responsible for one architecture and one operating system. The UNIX® host development group is in the unique position of having one operating environment running on many different architectures. Differences in machine architecture can have a major impact on certain areas of the host layer. The primary areas affected by these differences are dynamic loading and linking. Other areas that are also affected are tasking, arithmetic, and code generation. This paper discusses in detail each of the affected areas of the host layer and the impact the architectural differences have on their implementation.

PORTABILITY OF THE HOST LAYER

The SAS System is divided into three layers: the applications layer, the portable supervisor, and the host supervisor. The applications and portable supervisor layers compose approximately 95% of the SAS System. That percentage is growing as more applications are written. This leaves a small percentage of host-specific, and therefore, nonportable, code. The UNIX host development group has structured its host supervisor so as to achieve a high degree of portability even within this historically nonportable layer. This level of portability can only be achieved under the UNIX operating system and its derivatives and allows us to move SAS software to virtually any UNIX platform. The initial step in porting the SAS System to a new architecture requires a period of investigation ranging from a few days to a few weeks. During this time, all instruction-set-related documents and information describing compiler calling conventions and register usage conventions are studied. This is usually accompanied with disassembly of some small C programs to verify the documentation and to determine if there are any undocumented aspects of code generation that will affect the port. Object file format details are needed to implement a dynamic loader and are often not documented. Perusal of system include files usually gives us enough information to understand the format and write the code. The kind of documentation needed to assist in the development of the host layer is typically difficult to obtain since most software development projects do not require the level of technical detail needed to port the SAS System to a new machine. This problem tends to slow down the porting process.

Dynamic loading is the process of adding separately linked, relocatable modules to the address space of a running process. Differences in compiler conventions and object file format prevent us from accomplishing an abbreviated period of time. Differences in compiler conventions and object file format prevent us from achieving complete portability even when the chip is identical between two machines.

Portability of the host supervisor is accomplished using a combination of techniques. First, we try to use only system calls that are common and consistent across both System V™ and BSD UNIX, the two most common UNIX variations. This is not always possible. For example, signal handling is vastly different between SYS V and BSD, and tape I/O varies between manufacturers. In cases such as this, we use compile time switches to select the appropriate code, and layering to isolate nonportable portions. This gives us the ability to write most of the host supervisor with almost complete disregard of the underlying architecture. The problems for us stem from the portions of code that cannot ignore the architectural details of the machine. These portions are dynamic loading, dynamic linking, tasking, numeric formatting, and code generation. Many of these sections require total rewrites when moving to new architectures.

The UNIX host development group is currently investigating a large number of architectures from a variety of manufacturers for release 6.07 of the SAS System. These architectures include Motorola MC68000® and MC68000™, Sun Microsystems's SPARC®, MIPS Computer System's MIPS, HP® Precision Architecture, Intel® and Intel486®, IBM® POWER architecture, IBM 370, and National Semiconductor 32532™. Release 6.03 of the SAS System is available on HP-PA, SPARC, and two MC68000-based systems: the HP9000/300 series and the Sun-3. In working with such a large number of vastly different architectures, we have come to recognize the architectural and C compiler code generation features that have the greatest effect on SAS System implementation.

DYNAMIC LOADING

The main complicating factor in porting SAS software to a new architecture is the requirement that it be dynamically loaded. Dynamic loading is the process of adding separately linked, relocatable modules to the address space of a running process. Very few applications in the UNIX environment use this technique, though it is common under other operating systems.

Dynamic loading provides an extensible architecture to the SAS System. New products can be written after the base product is shipped. In addition, the user can choose, with options, which windowing environment to use or which graphics driver to use. Additional windowing environments can be supported after the product is released as well as new graphics drivers and procedures. Licensing of any combination of products is easily accomplished by providing the appropriate set of dynamically loaded images. User written functions, formats, informats and procedures fit nicely into this model. Modifications can be made to individual modules without fear of affecting other parts of the system. Dynamic loading makes more efficient use of system resources by only loading those portions that are needed, and unloading when necessary. We also need the ability to dynamically generate and execute code as in the DATA step code generation facility. Dynamic loading is the only feasible way to support a product as diverse and growing as the SAS System.

Operating systems such as MVS, CMS, and PRIMOS® have system-level support for dynamic loading. Under MS-DOS®, dynamic loading and overlays were the only way a product such as the SAS System could run in such a memory restricted environment. It was decided early on that dynamic loading must be supported and that the potential, additional run-time cost was justified. The fact that dynamic loading was foreign to the vast majority of UNIX systems meant that there would be little or no operating system support. The only versions of the UNIX operating system that directly support dynamic loading are IBM's AIX® and Apollo's SR10. Without uniform support, we are forced to write machine-specific dynamic loaders. Since object file formats often differ between versions of the UNIX operating system and different manufacturer's implementations, even machines based on the same architecture can require different dynamic loaders.
Simple Dynamic Loading

The most simple and portable way (in terms of a general algorithm) to dynamically load a module is to allocate memory sufficient to hold the module, read the module into the allocated memory, and relocate the image in memory. Relocation involves adjusting position-dependent portions of an object module. When a linker produces a relocatable module, it creates data structures called relocation records. Each relocation record describes an item in the object module (usually an address) that was generated assuming that the module would be loaded at some fixed address (usually zero). Relocation records must be produced by the linker since its job is to combine separately compiled object modules into a final, nonrelocatable module. The linker (if it is producing a relocatable module) and the compiler or assembler cannot know, at the time an object module is produced, what its position will be in the final executable. Relocation records are typically generated for such things as global data references and function calls. Relocation records allow us to position an object module anywhere in memory. The host supervisor is typically the only portion of the SAS System not kept in relocatable, dynamically loadable modules.

The run-time cost of dynamically loading a module using the method just described is significant. The amount of disk I/O performed is substantial, although the penalty is only paid when the module is first loaded (although unloading can occur). The amount of overhead involved in relocating the image is usually quite small and probably negligible compared to the I/O. The only real impact on performance in the relocation phase is that practically every page in memory where the file was loaded will be modified, causing a very large number of page faults. This has an impact on the entire system, not just the SAS session running. This is especially a problem in a diskless workstation environment, where paging is being done across the network, and the performance impact can be felt on far more than just one workstation and one server. Reducing the cost of dynamic loading is an important goal, and substantial performance gains can be realized.

Most of the techniques used in this regard are specific to a particular architecture, a particular version of the UNIX operating system, or a specific C compiler implementation, but some techniques can be applied to more than one platform.

File Mapping

The number one source of overhead in dynamic loading is the amount of I/O being done when a module is read into memory. This is the reason for the pause when a procedure is first executed. One method used to eliminate this pause and much of the I/O is to map the module into the host supervisor’s address space. Mapping is a process that involves having the operating system associate an area of memory with a file on disk. We simulate mapping when we allocate memory and read in the object module in the simple dynamic loading model. The overhead of mapping in a file is very small. The operating system does not actually perform any I/O but simply sets up page tables that refer to the file specified. A pointer is returned to the host supervisor which tells it where the file is located in memory. When relocation begins, the dynamic loader starts to modify areas of memory which correspond to the file. At this point, pages of the file are actually read in by the operating system and local, modified copies of each modified page are made. Mapping in this manner is referred to as a private mapping since the changes being made to the mapped-in file are private to the process making the changes; that is they do not actually modify the file. In memory management terminology, the file is mapped in copy-on-write. An important aspect of mapping in object modules besides the I/O overhead savings is the fact that all unmodified pages can be shared between processes mapping in the same file.

Relocation

The file mapping method of loading has the potential to share the modules mapped in but immediately loses its main advantage over other methods on many architectures. The reason for this is the relocation required after the image is loaded. On many architectures, the amount of relocation in a module causes practically every page of the file to be modified. The actual amount of I/O required is not greatly reduced over the original method (allocate space, read in the file, relocate it). The original method is guaranteed to hit every page in the file simply because every page in the file must be read. Mapping at least has the chance of not hitting every page if relocation does not force every page to be modified. We have reduced some of the overhead by using mapping since the pages will no longer be hit two times during loading (once when read in, once when relocated).

The address space of a process doing a large amount of dynamic loading will be extremely large, and practically all of it will be unshared. This puts a large demand on system swap space and can adversely affect system performance. Obviously we need to reduce or eliminate relocation records to achieve the best performance. The only way to eliminate relocation for the potentially shareable code portions of dynamically loaded modules is for the compiler to produce position-independent-code (PIC). This is accomplished through a number of techniques, all involving performing relocation only in the already unshareable data section (each running process must have its own data and stack). Rather than generating direct references to data or function addresses in the code section, indirect references through the data section or off of a base register pointing to the data section are generated. When a PIC object module is dynamically loaded by mapping, the only relocation that takes place is in the data section. It is often grouped on a small number of pages in a table format that is easy to relocate. The code segment can be mapped in read-only since it should now never be modified. This way, the entire code segment of an object module is shared and does not consume swap space since it will be paged directly out of the file on disk. When other SAS sessions map in the same module, they will simply share the kernel data structures that refer to these files.

On one workstation with 32 megabytes of swap space, before PIC and mapping, we were able to bring up only 12 SAS sessions before all of the swap space was consumed. After PIC and mapping, we could bring up 29 SAS sessions. Each SAS session consumed between 2 and 2.5 Meg with the unshared version and about 900K with the shared version. The extra space consumed by each invocation of the SAS System is the private data and stack space. In the case of the SAS System running on a diskless workstation, the amount of network traffic is reduced. This is because most procedures will not end up hitting every page in the file that is mapped in. The network page requests will also be spread out because of time spent executing on the current page, before the next page fault. Using these methods we can reduce the cost of dynamic loading to almost negligible amounts. The added flexibility this method provides justifies the cost. File mapping is unfortunately not available on all UNIX systems. On systems that do not support it, dynamic loading is performed by simulating mapping and is much less efficient.

Not all compilers have the capability of producing position-independent code, though often the type of code generated is only partially position dependent. The MIPS architecture is an example of this. Function calls are not position independent and some data references in the code space will require relocation. However, most data references use an addressing mode which does not require relocation. The chip provides a global data pointer register (gp register) as a base register from which the first 64K of global data can be referenced in a position-independent manner. To reduce relocation on this architecture, we have done two things. We have written a utility
that post-processes a relocatable module and replaces non-PIC function calls with a PIC version if the function can be reached with this more limited instruction. We then can remove the associated relocation record. We also force all data to be referenced using relocation process, so swap space is consumed (reserved) for each module. We still find this a much better alternative to the straightforward method of dynamic loading.

**DYNAMIC LINKING**

Static linking is the function performed by a linkage editor, where external references get resolved from the list of object files used to build a module or from libraries of object files. Dynamic linking is the process that resolves external references at run-time. The mechanism used to perform dynamic linking is called bridging. Different architectures require substantially different bridging techniques. On some architectures, there is considerable overhead for each intermodule call, while on others the overhead amounts to only a few instructions. In all cases, calls between dynamically loaded modules are more expensive than static calls within a module. The overhead of the bridging mechanism has been studied in depth to determine its run-time effect on performance, and the bridge code actually has little impact. If the bridge could be totally removed, it would certainly help performance, but we do not believe the difference would be dramatic. Bridge calls are typically made for expensive services such as reading or writing data sets or formatting variables. Once a procedure is entered, processing for the most part, remains local to that procedure. On architectures that require extensive bridge support, a bridge call can take up to 20 times longer than a local function call. The number of bridge calls though is not nearly enough to significantly affect performance. On the MIPS architecture, for example, the bridge code is quite extensive, but 1 million bridge calls amount to only one or two seconds of CPU time. Testing has shown that it can take several minutes of CPU time, running some typical SAS procedures, to generate a million bridge calls. The bridge therefore has approximately a 1% to 2% performance penalty in CPU time, and an even smaller percentage when compared with real time.

Dynamic linking in the SAS System is accomplished through the use of unique identifiers (called subsystem IDs) for each module that contains entry points that can be called externally. Each of these modules form a table of entry points, and each function has a unique index into this table that identifies the function. The host supervisor maintains a data structure called the subsystem vector, which is an array of pointers to the function tables of each of the modules and is indexed by the subsystem ID. Each array slot is filled in at run time when a module is loaded. We end up with a unique (i,j) pair for each externally callable function. The i value is the subsystem id and the j value is the index into the function pointer table of the subsystem.

Another vital element is the transfer library. The transfer library is a collection of bridge stubs. One bridge stub is associated with each externally callable function in a module, and each module containing such entry points has a transfer library. Each bridge stub is built with the unique subsystem ID and function pointer index hard-coded into it. A bridge stub is just a small piece of code used as a stepping stone to the real function whose address is unknown until run time. The system linkage editor uses the transfer libraries to resolve references to external functions. For example, the GLM procedure needs to call a certain vector math routine that is contained in another module. Since this function, as well as many others, will not reside within PROC GLM, it is necessary to link PROC GLM with the transfer libraries of modules it may make calls to. One of the transfer libraries contains the bridge stub for the vector math routine. This bridge stub will be brought into PROC GLM to resolve the external reference to this math routine and will provide the necessary linkage to allow PROC GLM to make this external function call.

The simplest type of bridge stubs are the type found on the MC68000, Intel 8086/88, SPARC, and MC68000 based systems. The overhead of dynamic linking is the smallest on these systems. When a call is made by a module to a bridge stub, the bridge stub simply indexes the host supervisor's subsystem vector with the subsystem ID that was hard-coded into it. It then indexes the resulting function table pointer to obtain the real function address and jumps to the function. When the called function returns, it ends up back where the bridge stub was called from, as if nothing out of the ordinary had happened. As you can see, the overhead involved in making such calls is only a few instructions, and typically this overhead is negligible compared to the time required to perform the function.

In the example just given, it is assumed that the module being called is already loaded in memory. The first call through a bridge stub to a module requires special handling. Dynamic loading in the SAS System supports the concept of demand loading. A module is not loaded until it is needed. Two things are required to perform this operation. Another data structure in the host supervisor holds the names of modules that can be demand loaded, and is indexed by the subsystem ID. This structure is called the subsystem name table. When the portable supervisor is first loaded, a number of commonly used module names are placed in this table. When a procedure is loaded and run, and it requires access to less commonly used modules, it may either load these directly or install them in the name table for demand loading. The second requirement is that the subsystem vector be initialized to function pointers that point to a demand loading routine in the host supervisor. The first time a module is called into from a bridge stub, rather than ending up at the desired function, you end up at the demand load routine. The subsystem ID and the function table index are in some fixed location (the bridge stub typically loads them into two registers). This routine then indexes the subsystem name table and dynamically loads the correct module. The modules entry point is called after loading. The job of the entry point routine is to export its function pointer table by calling a host routine that replaces the default subsystem vector table entry with the one given. The demand load routine then completes the call to the originally called function. From this point on, all calls into this module will go directly through the subsystem vector as described in the example. When a module is unloaded, its entry point table is simply replaced with the default entry point table pointing to the demand load routine, so it can be reloaded if needed.

The other assumption made in the example is that each of the dynamically loaded modules has access to the host supervisor's subsystem vector data structure. The address of the subsystem vector is made available to each dynamically loaded module by passing this information to the entry point of the module. Aphony entry point is generated for each module, that saves this information along with other useful information in global variables within the module and passes back the real entry point. The subsystem vector is referenced by using a global variable that gets filled in by the entry point routine, so before the real entry point is called, the variable contains the correct address of the subsystem vector in the host supervisor.

Unfortunately, not all architectures are as simple as the ones just described. Architectures such as the IBM POWER architecture and HP-PA require slightly more work to be done in the bridge stub. Other architectures such as the NS32X32 and the MIPS require substantially more work. The reason for the extra work
is that certain registers must be saved and initialized before the call, then restored after the call. For a detailed example, I will describe the MIPS architecture.

As previously mentioned, the MIPS architecture has a register called the global data pointer (the gp register) that contains the address of the first 64K of global data for a module. No individual module in the SAS System has over 64K of static data, so every module is compiled to use the gp register for all data accesses. This reduces the code size and speeds up execution by allowing all global data to be accessed in one instruction using gp-relative addressing. The problem is that each separately compiled and loaded module has its own data and therefore its own unique gp register value. When one module calls another, the caller’s gp register must be saved and the callee’s gp register loaded. On return, the caller’s gp register is restored.

In order to accomplish this, the bridge stubs on the MIPS load the subsystem ID and function table index into registers and then call a stub handler function. This function saves the callee’s gp register and return address onto a bridge stack, which is separate from the task’s real stack. It then loads up the callee’s gp register, sets the return address to come back to the stub handler, and jumps to the correct function in the same way as the simpler bridge stubs of other architectures. A subsystem gp table is maintained by the host supervisor. This table is indexed by the subsystem ID and holds the correct gp value for each module. The gp values are set when a module is loaded.

The MIPS compiler actually gives us a choice of how we want it to generate global data accesses. The code can be compiled to never use the gp register, which would make the MIPS look like the simpler architectures and allow us to remove the vast majority of bridge overhead. Our initial implementation actually used this method (there turned out to be some other problems with this choice that make it difficult, but not impossible). Timing tests indicated that there was no noticeable difference between the different implementations. This must be due to a combination of few bridge calls and few global data accesses, one in favor of each implementation. The high bridge call overhead version was chosen for two reasons: many C library functions are used by both the SAS System and the X Window System™. Many of these functions already contain gp-relative addressing, and cannot be recompiled. Secondly, the gp-relative versions contain substantially fewer relocation records providing a greater opportunity for sharing modules when using memory mapping for dynamic loading.

The bridge stack required for the MIPS is also needed on the National Semiconductor based machines. It can be the source of some complexity in the bridge handler. The bridge stack must be checked for overflow, each time through, adding to the overhead. When the bridge stack overflows, it can be grown and moved, or task can be killed, depending on the implementation. Architectures such as the HP-PA and IBM’s POWER architecture have registers that must be saved and restored, but the compilers on these machines provide a fixed stack frame for each function call that contains space for saving the necessary registers. Using this space allows us to avoid the overhead of the bridge stack and lets us use the real stack for the purpose it was intended. The bridge stubs on these architectures tend to be more complex since they perform the necessary register manipulations before calling the function, but they do not use a bridge stub handler routine.

The only routines that need to be written in assembler for all architectures are the bridge stubs, the demand load routine (at least a portion of it), and the bridge stub handler routine. These routines must not alter the stack frame and may have to preserve registers. They must also jump to entry points in a manner not supported in high level languages. The bridge stubs are automatically generated as are the function tables exported into the subsystem vector.

Another required feature of dynamic linking that can be the source of substantial overhead is the use of function pointers passed between dynamically loaded modules. This is quite common in the SAS System and is also heavily used in the X Window system. The obvious problem with passing around function pointers is that a call can be made from one module to another without going through the bridge and therefore not setting up the required registers. Once again, on architectures requiring simple bridging, this is not a problem. Function pointers can be passed around without regard to the function’s location. Complicated bridging though must have control over the function pointers being passed. The SAS System has the concept of global function pointers. A global function pointer is one that can be passed to any function and called from any function. A function pointer is converted to a global function pointer by calling a local-to-global conversion routine, which takes a local function pointer and returns a global function pointer. On simple bridging systems this function is a simple assignment, but on other systems, the function must generate a stub at run time to call through which acts like a bridge stub.

**TASKING**

In the SAS System, each procedure and each window is a separate pseudo-task that is not a real task as seen by the operating system, but one controlled within the address space of the SAS session being run. This form of tasking is referred to as lightweight processes or threads. The current implementation of the SAS System allows for only one active task, but future implementations will have multiple running tasks that will require scheduling. This portion of the host supervisor has remained surprisingly portable. A small section of code that performs the actual task switch is machine dependent, and some machine dependencies exist in the creation, destruction of tasks. Task switching in this environment uses the C library routines setjmp() and longjmp(). Before jumping into the environment of a new task, we provide the task with a stack and data space. The stacks are typically just allocated space, and the stack pointer is manipulated by altering the jump buffer filled in by the setjmp() call. This simple mechanism works correctly on systems that allow the stack pointer to be easily manipulated, but the SPARC architecture forces us to use a different mechanism for tasking.

On all architectures we have worked with other than SPARC, the stack is completely under our control. The SPARC architecture has the concept of a register window. Each time a function is called, the window is rotated over a new set of registers for the current function to use. Parameters and return address information is passed by allowing eight registers to overlap in the register window. This mechanism lets the system take care of saving and restoring register contents rather than leaving register management up to the compiler. The good and bad points of this implementation have been debated for some time. SPARC systems typically have a large number of registers to work with, but eventually there will not be enough registers to give the current function a new set. At this point, the registers must be saved on the stack by the OS. It was feared that altering the stack pointer and using longjmp() to start a new task would confuse things terribly. Other programs that performed similar stack manipulations did not work on the SPARC. We decided to use a lightweight process library provided by Sun Microsystems to perform tasking. This library provides all the functionality we need and stack manipulation is left up to this library. This approach can also be used on the other Sun systems using the Motorola and Intel architectures providing us with a portable tasking subsystem for any systems implementing the lightweight process library. The NeXT™ machine for example provides a similar library. Without the lightweight process library, or substantial help from Sun, it is
doubtful we could have ported the SAS System to the SPARC architecture.

**ARITHMETIC**

The arithmetic formatting routines used by the SAS System can potentially be the source of substantial work when porting. We have been lucky so far, in that every system we have ported to has chosen to implement the ANSI/IEEE 754 standard for binary floating-point arithmetic. The only problems encountered have involved byte-swapping on some architectures, which is not a difficult problem to deal with. Compile time switches are used to select byte-swapped/non-byte-swapped versions of the routines.

Some routines have been written in assembly language to enhance performance since a small set of routines are used in format practically every number the SAS System produces. RISC architectures tend to be difficult to write assembly language for, so more performance analysis needs to be done to see if it is justified. CISC architectures often have complex formatting instructions that make it natural for these types of routines to be written in assembler.

The IEEE format provides an ideal representation for missing values. This numeric format has the concept of Not-a-Number (NaN), which is an invalid numeric representation used to hold special, nonnumeric values in a numeric format. This is exactly what a missing value is. The only difficulty lies in the fact that NaNs do not collate, but missing values do have a specified sorting order. The choice of missing value encoding can have a substantial impact on floating point performance. Since math operations cannot be performed on missing values, all SAS procedures must filter out missing values. It is important that this can be done quickly. The NaN representation is easily detected (one instruction on CISC) and the actual missing value collating index is easily extracted (two instructions on a CISC).

There are two different types of NaNs, signaling and non-signaling. Signaling NaNs cause floating-point exceptions when used in operations, non-signaling NaNs merely propagate the NaN. We chose a non-signaling NaN as our representation because the simple operation of moving a signaling NaN through a floating point register can cause an exception on some machines. The non-signaling version is also easier to detect because of its encoding. The obvious advantage of signaling NaNs is that they can be used to find bugs in procedures, where missing values are being used in operations. The representation of missing values on some non-IEEE format machines can allow a missing value to be erroneously used in a calculation. This is most often harmless, but can lead to bugs. An interesting architectural difference showed up between the HP300 and HP800 series machines: the representation of signaling and non-signaling NaNs is reversed. On discovering this, we initially thought we would have to alter the missing value representation on the HP800 series. Fortunately, the use of signaling NaNs as missing values on the HP800 only caused exceptions when the missing value was used incorrectly. The HP800 became an excellent tool for tracking down bugs, not only because missing values were represented as signaling NaNs, but also because it was the first RISC machine we had ported to. RISC chips tend to have alignment requirements which greatly simplify hardware and speed up access to memory. Data items must be aligned on a boundary that is a multiple of the size of the item. Alignment restrictions must be taken into account when writing portable code. Quite a few nonportable alignment and size assumptions were discovered in the applications layer when porting to the HP-PA that worked perfectly on other architectures.

**CODE GENERATION**

Code generation, used by the DATA step, some procedures, and for variable movement during I/O operations is obviously the most architecture-specific portion of the host supervisor. It also turns out to be one of the most time-consuming and difficult portions. Many different UNIX platforms share the same chips, the MC68000 and MIPS. In particular, Code generation for these chips can be ported between different UNIX platforms with little or no change. Occasional differences occur in parameter passing and return value location on the MC68000. The main difference in the MIPS code generator lies in the fact that the MIPS chip can be either byte-swapped or non-byte-swapped, but this presents little difficulty. Taking a layered approach to the design of this subsystem has allowed us to restrict the truly machine-dependent portion of the code generator to a small set of functions. This frees the developer to concentrate on the actual code generation rather than the scaffolding needed to prepare for code generation and execution of the generated code. This approach has allowed us to complete a large number of code generators in a very short period of time.

The code generators under the SAS System do not have the same requirements as C compiler code generators. The code must be generated quickly and, in the current release, is always thrown away after being run. Extensive optimization does not have much impact since the vast majority of code generated involves I/O and is run-time intensive. Optimization is also time consuming. On I/O bound applications, code generation tends not to do much better than an interpreter. However, on compute bound applications, code generation is substantially faster. Many applications tend to have a good mixture of I/O and computation and the difference between the code generator and interpreter is very noticeable. Code generators for SAS applications typically use only a small set of machine instructions. This makes the difference between RISC and CISC code generators, in terms of difficulty, quite small. Some complex operations for which it is feasible to generate in-line code on CISC machines have to be delegated to run times on the RISC machines. Function call overhead though is typically quite small on RISC architectures, so the penalty for calling a run time can be ignored.

**CONCLUSION**

The UNIX host group has strived to minimize the impact of machine architecture on our porting of the SAS System to a wide variety of platforms. We have achieved almost complete portability of the host supervisor between UNIX systems, and the portable supervisor and applications code is completely portable between such diverse systems as MVS, CMS, VMS, MS-DOS, and OS/2®. With over 3 million lines of C code, we think this is quite an achievement.

An important point that needs to be made is that in spite of any implementation difficulties or architectural strangeness, the SAS System runs on all the machines discussed. The performance of the system seems to bear no relation to the amount of effort required on the part of the UNIX host group in porting to a particular machine. Fast machines run the SAS System very fast which should not surprise anyone. The group is always concerned about performance, and we are always looking for ways to improve it. When the dynamic linking overhead became large on some machines, we worried about its performance impact, but this has proven not to be a factor. We hope that whatever a user's preference in machines or operating systems, we can provide SAS software on that platform.

Just a few short years ago, the battle of the architectures was being fought primarily by two companies, Intel and Motorola, with two CISC chips. Today the field has widened considerably. The
current battle is the battle of RISC versus CISC, even from within Motorola and Intel. Using the SAS System as a real world benchmark, it is clear to us that RISC performance is substantially better than any of the current CISC offerings. Low end RISC machines can often outperform the top of the line CISC systems. For developers, the big question is, does development of software for RISC-based machines mean more work than development for CISC machines? Is the amount of work required to support the simplicity of RISC so much more than CISC that it becomes difficult and time consuming? No, not at all. The factors that make a port difficult for us are not inherent in RISC or CISC design. The sources of difficulty come from decisions made at the architectural, compiler and OS level that can be made in either design.

RISC chips will not totally replace CISC chips for quite some time, if ever, for a number of reasons. The most compelling reason is the amount of existing software. Modern CISC designs like the MC68040™ and Intel486 are integrating some RISC design techniques into their own designs. The idea is to make the most common CISC instructions execute in as close to one clock cycle as possible, thereby matching RISC performance. The more complex instructions and addressing modes will still be available, for the rare times they are used. This should increase performance of CISC chips while still offering the instructions that assembly language programmers like. However, the reduction in chip size needed to obtain higher clock speeds and the use of faster materials which are currently difficult to work with, make the design of a 1.2 million transistor CISC chip a formidable task. Some RISC designs have one tenth the number of transistors. RISC chips will continue to have much shorter design cycles and will therefore be able to take advantage of new technologies more rapidly than CISC chips. The architecture battle seems to be narrowing down to simply which RISC design is best.