



Migrating Solaris applications to the IBM AIX platform

Authors:

*A. Govindjee, V. Kumar, N. Keung, L. Cheng,
Bradford Cobb, W. Huang, Glen “Tex” Chalemin*

Update by Ruzhu Chen, Venkataravikumar Dadi and Kathleen Dowling

September 2010



Table of contents

Abstract	4
Helpful terms and definitions	4
Overview	4
Brief description of AIX	5
General differences between AIX 6.1 and Solaris	6
Defines for numerical values	6
LIBPATH environment variable	6
Language data types	6
Compiler options	7
Library options	9
Libraries and archives.....	10
Differences between AIX 6.1 and Solaris regarding shared objects and shared libraries	10
Differences between AIX 6.1 and Solaris for linking and loading	10
Differences between AIX 6.1 and Solaris for library search-path creation and usage....	11
A comparison between the Solaris and AIX 6.1 environments for library search paths .	13
Memory segments.....	13
32-bit address space	14
Runtime issues — 32- compared to 64-bit kernel.....	14
AIX 4.3 compiled 32-bit and 64-bit applications on AIX 6.1	15
32-bit and 64-bit applications on AIX 6.1	15
Performance tuning.....	16
Threads programming	16
Determining whether an application uses Solaris or POSIX threads.....	17
Comparing the POSIX thread library and the Solaris thread library	18
Signals	19
Twelve AIX 6.1 tools similar to the Solaris /proc tools.....	19
Porting issues	20
Solaris gcore and coreadm commands	20
Compiler errors	20
Namespace	20
Containers	20
Casting	20
fstream.....	20
fdopen.....	22
char constant reference.....	22
const reference.....	23
Storing returned object in a reference	24
Copy constructors, equals operator, and comparison operator	25
riend usage.....	25
Static friend usage.....	25



Class qualifiers	25
Text after #endif.....	25
Linked lists.....	26
Comments in C++ style	27
ENONET error	27
Macro MAXHOSTNAMELEN	27
System call differences.....	27
High resolution real time call	27
Socket on top of stream operation	27
econvert(), fconvert(), and gconvert().....	28
clock_gettime()	28
Unresolved seteuid() and setegid() calls.....	28
Access process information	28
Signals.....	29
SIG_PF signal	29
cftime().....	29
dladdr() — Calculating symbolic addresses.....	30
Programming issues.....	32
IPC limits.....	32
The shared memory limit and using EXTSHM	34
RPC.....	34
Stack frame issues.....	35
/proc file structure	36
Memory mapped files.....	36
Java issues	38
Application performance issues.....	38
AIX environment variables.....	38
AIX memory pages	39
AIX memory affinity.....	39
AIX application performance analysis tools.....	39
AIX application optimization tools.....	40
Service and support.....	40
Summary.....	41
Resources	42
About the authors	45
Acknowledgements.....	46
Trademarks and special notices.....	47



Abstract

This paper serves as a technical guide to migrate applications from the Sun Solaris platform to the IBM AIX platform. In many cases, migration requires nothing more than a simple recompilation, but there are exceptions. Throughout this paper, various migration scenarios are discussed, with a focus on the instances that require changes to the application source or to the way the application is built for an easy move from Solaris to AIX.

Helpful terms and definitions

Table 1 shows terms and definitions that are referenced in this paper.

ILP32	A 32-bit application source. In this model, the size of int, long and pointer for C and C++ is 32 bits.
LP64	A 64-bit application source. In this model, the size of int for C and C++ is 32 bits; the size of long and pointer is 64 bits.
AIX 6	IBM® AIX® Version 6 operating system that runs on IBM POWER™ and IBM PowerPC™ processor-based systems. IBM AIX Version 6.1 and IBM AIX Version 5.3 are currently available.
Little Endian (LE)	The right-to-left assignment of addresses to bytes in a data word.
Big Endian (BE)	The left-to-right assignment of addresses to bytes in a data word. Both AIX 5.3 and Solaris are Big Endian.

Table 1. Terms and definitions that this paper uses

Overview

Most applications are written in one or more high-level languages. Because applications that are written in assembly or macro assembly need complete rewrites, this paper discusses porting issues mostly in terms of C and C++. Variations of the problems occur with other high-level languages.

Most well-written C and C++ programs compile and run on AIX 6.1 without change, where *well-written* means the use of good programming practices, including:

- Conformance to the ANSI/ISO C/C++ standard
- Portability considerations high in the design, implementation and maintenance phases of the software cycle
- Use of prototyped function declarations throughout the code

Migrating applications from one platform to the other involves recompiling, relinking, debugging, testing and, if necessary, performance tuning. Well-written programs often port easily with few code changes when they adhere to language standards and POSIX interfaces and when they are not overly dependent on the platform architecture.



Brief description of AIX

AIX is the strategic IBM UNIX[®] based operating system for mission-critical, core-business applications. The most currently available release of AIX is version 6.1. One previous release of AIX, version 5.3 is still supported. The current IBM C/C++ compiler is IBM XL C/C++ Enterprise Edition Version 10.1 for AIX.

The industrial-strength features and functions of AIX have been well proven over the years in a wide variety of server environments, from relatively small, single-processor systems through massively parallel Scalable IBM POWER parallel (SP) servers.

Features in the latest version, AIX 6.1, include a state-of-the-art 64-bit kernel with both 32- and 64-bit application programming interface (API) support. AIX 6.1 also includes the following:

- Dynamic logical partitioning (LPAR) for processors, memory and I/O
- IBM Micro-Partitioning™ (LPARs)
- Workload Partitioning (WPARs)
- Active Memory Sharing
- System Planning Tool
- Live Partition Mobility (LPM)
- Live Application Mobility (LAM)
- Virtual I/O server (VIO server) — disk, tape, Ethernet
- N Port ID Virtualization (NPIV) — management of fibre-channel SAN environments
- System Planning Tool — planning and installation of POWER processor-based servers with IBM PowerVM™ processors
- IBM Systems Director Console for AIX
- Dynamic Capacity Upgrade on Demand
- Cluster Systems Management — monitoring and administering multiple machines (both AIX and Linux[®] operating systems) from a single point of control
- Enhancements to Enterprise Storage Management
- Advanced reliability, availability and scalability (RAS) features
- Additional security features and enhancements
- Network enhancements — including Mobile IPv6, Simple Network Management Protocol (SNMP) V3 and upgrade to BIND V9
- APIs from the latest C language and single UNIX specification standards
- Twelve utilities similar to the `/proc` tools provided on Solaris
- Supported on selected IBM POWER6 processor-based systems



General differences between AIX 6.1 and Solaris

The following sections describe some of the differences between AIX 6.1 and Solaris.

Defines for numerical values

All defines for numeric values on AIX 6.1 are in `/usr/include/sys/limits.h`. On Solaris, the defines are in `/usr/include/limits.h`.

Because of differences in the system architecture, there are some differences between the maximum and minimum values of standard defines between the two operating systems. See the *include* files just listed.

LIBPATH environment variable

The environment variables for indicating the library paths are different on Solaris and AIX. On AIX, use `LIBPATH` in place of the Solaris variable `LD_LIBRARY_PATH`.

Language data types

The `double` data type on AIX (32- and 64-bit) has a size of eight bytes, but is aligned at a 4-byte boundary. See Table 2 (32-bit data types) and Table 3 (64-bit data types) for more information.

Data type	AIX		Solaris
	32-bit source	64-bit source	
			64-bit
char	8	8	8
short	16	16	16
wchar_t	16	16	32
int	32	32	32
float	32	32	32
pointer	32	64	64
long	32	64	64
long long	64	64	64
double	64	64	64
long double	64	64	128
		128 (when compiled with LONGDOUBLE128 define)	

Table 2. C Language data types (data-type sizes are indicated in bits)



Data type	AIX (64-bit)	Solaris (64-bit)
char	1	1
wchar_t	4	4
short	1/2	1/2
int	4	4
float	4	4
pointer	8	8
long	8	8
long long	8	8
double	8	8

Table 3. Natural Data Alignment (data-type sizes are indicated in bytes)

Many RISC systems are sensitive to the alignment of data within memory and give the best performance when data is aligned on particular boundaries. However, there is no silver bullet. Aligning data on certain boundaries might be an excellent choice when dealing with one data structure, but not so good when dealing with an array of thousands of data structures. The reason is that memory space might be used less efficiently, or there might be more frequent cache misses.

Four alignment options are supported by the IBM XL C/C++ compiler, as detailed in Table 4.

-qalign	Effect
Power (or full)	Uses PowerPC alignment rules
Twobyte	Uses the Apple Macintosh alignment rules
Packed	Uses the packed-alignment rules
Natural	Structure members are mapped on their natural boundaries (as reflected in Table 3)

Table 4. Alignment options that are supported by the compiler

The default alignment is `Power`. However, `Natural` is actually the same as `Power`, except that it also applies alignment rules to doubles and long doubles that are not the first member of a struct or union. Thus, `long`, `double` and `long double` are word aligned, unless `-qalign` is set to `Natural`: doubles are doubleword (8 bytes) aligned and long doubles are long doubleword (16 bytes) aligned.

Compiler options

For details about features of the latest version of the native compiler for AIX, IBM XL C/C++ for AIX V11.1 (XL C/C++), see *Guide to porting to the IBM XL C/C++ for AIX compiler*, go to ibm.com/support/docview.wss?uid=swg24028013. This version of the compiler exploits the POWER7 hardware and supports the new AIX 7.1 operating system.

AIX provides full support for the ANSI specifications of the C and C++ languages. XL C/C++ adheres to the latest approved definitions of these languages. Additionally, XL C/C++ supports a subset of the gcc and g++ extensions to ease the migration of applications that have been developed using gcc and g++. The XL C/C++ V11.1 compiler also offers the capability of being invoked by using `gclx` and `gxc++`. These invocations allow the user to use the GNU options and have the XL C/C++ compiler automatically translate them. Not all GNU C/C++ flags have a corresponding XL C/C++ flag. A warning is given for



untranslated flags. For the full list of supported GNU option compatibility, see *Guide to porting to the IBM XL C/C++ for AIX compiler* at ibm.com/support/docview.wss?uid=swg24028013.

By default, the compiler binaries are installed in `/usr/vac/bin` (only install C compiler) and `/usr/vacpp/bin` for C/C++. To find the latest version of the XL C/C++ V11.1 documentation, see the information center at <http://publib.boulder.ibm.com/infocenter/comphelp/v111v131/index..>

The XL C/C++ V11.1 compiler product includes a number of default configurations in the `/etc/vac.cfg` file, which are activated depending on which exact driver program was called. For example, to compile an ANSI C program by using the UNIX98 threads libraries, use the `/usr/vacpp/bin/xlc_r` command. To compile an ANSI C program using the POSIX Draft 7 threads library, use the `/usr/vacpp/bin/xlc_r7` command. For threaded C++ applications, use the `/usr/vacpp/bin/xlc_r` command.

By default, the compiler creates 32-bit object code on AIX and Solaris. If the application is compiled with the `-m64` or `-xarch=v9|v9a|v9b` flag on Solaris, then the code is written for a 64-bit environment and compiled in 64-bit mode. The option to enable compiling an application in 64-bit mode on AIX is `-q64`. Another way to obtain 64-bit object code is by setting the environment variable, `OBJECT_MODE=64`. With the XL C/C++ compiler, `-qwarn64` warns of potential 64-bit problems that are related to 64-bit migration.

Note that any third party library being linked into a 64-bit application must also be 64-bit. To determine the object mode of any 32-bit module or library on AIX 6.1, run the command:

```
nm -X32 <file_name>
```

...where `file_name` is the name of any object module, such as `libxyz.a`. In the output, search for the string `XCOFF`. For 64-bit modules, use:

```
nm -X64 <file_name>
```

For a detailed suggested mapping (or cross-reference) of Solaris C/C++ compiler options to the options available on AIX, see *Guide to porting to the IBM XL C/C++ for AIX compiler* at ibm.com/support/docview.wss?uid=swg24028013. This guide also describes some language features that have been implemented to make porting easier and a new option combination tool to help you select optimization options for your application.

You can also find more information Appendix C of the IBM Redbook, *AIX 5L Porting Guide* at ibm.com/redbooks/abstracts/sg246034.html.

Also see the man pages for the AIX `ld` command. Table 5 compares the Solaris and AIX library and linking options.



Library options

Table 5 compares the Solaris and AIX 6.1 compiler library and linking options.

Solaris compiler flag	AIX 6.1 XL compiler flag	Action
-Bbinding	-bbinding	Requests symbolic, dynamic or static library linking
-d{y n}	-d{y n} only valid when used with -bsvr4	Allows or disallows dynamic libraries for the entire executable file
-G	-G, but some differences -qmkshrobj also	Builds a dynamic shared library instead of an executable file
-hname	n/a	Assigns a name to the generated dynamic shared library
-i	n/a	Tells ld to ignore any LD_LIBRARY_PATH setting
-Ldir	-Ldir	Adds dir to the list of directories to be searched for libraries
-llib	-llib	Adds liblib.a or liblib.so to the linker's library search list
-library=llist	n/a	Forces inclusion of specific libraries and associated files into compilation and linking
-mt	xIC_r invocation	Compiles and links for multithreaded code
-norunpath	n/a	Does not build path for libraries into executable
-Rplst	-bllibpath	Builds dynamic library search paths into the executable file
-staticlib=llst	-bstatic -llib -bdynamic	Indicates which C++ libraries are to be linked statically
-xar	n/a	Creates archive libraries
-xbuiltin[=opt]	n/a	Enables or disables better optimization of standard library calls
-xia	n/a	Links the appropriate interval arithmetic libraries and sets a suitable floating-point environment
-xlibmieee	n/a	Causes libm to return IEEE 754 values for math routines in exceptional cases
-xlibmil	n/a	Inlines selected libm library routines for optimization
-xlibmopt	n/a	Uses library of optimized math routines
-xlic_lib=sunperf	n/a	SPARC: links in the Sun Performance Library (preferred)
-xnolib	n/a	Disables linking with default system libraries
-xnolibmil	n/a	Cancel -xlibmil on the command line
-xnolibmopt	n/a	Does not use the math routine library
-xlang=[,l]	n/a	Includes the appropriate runtime libraries and ensures the proper runtime environment for the specified language

Table 5. Comparison of compiler's library and linking options

Libraries and archives

On AIX, an archive can contain objects of different types. For example, `libc.a` contains both 32- and 64-bit `.o` files, 32-bit shared libraries, and 64-bit shared libraries. Applications (both 32- and 64-bit) can link dynamically, statically, or both against this single archive (`libc.a`). The AIX linker works somewhat differently than the Solaris linker.

On AIX, symbol resolution is performed at link time. If a symbol is defined in a shared object (or module) referenced on the command line, the linker records the symbol name and defining module in the loader section of the program. When a program runs, imported symbols must be resolved by finding them in the defining module, as recorded at link time.

Differences between AIX 6.1 and Solaris regarding shared objects and shared libraries

AIX 6.1 and Solaris have different views of shared objects. A shared module in AIX 6.1 can be an object file or an archive file member. A single shared archive library, as just described, can contain multiple members, some of which are shared modules and some of which are ordinary object files. In Solaris, shared libraries are always ordinary files, which are built from source files that are compiled with special options and linked in a particular way. Typically in AIX 6.1, system-shared libraries are archive files, which makes it easier to provide updates to the system libraries with minimum impact to the applications.

Differences between AIX 6.1 and Solaris for linking and loading

In AIX, all of the linker's input shared objects are listed as dependents of the output file only if there is a reference to their symbols. In Solaris, the names of all shared libraries listed on the command line are saved in the output file for possible use at load time. However, in AIX, the `-brtl` option causes all shared objects (except archive members) on the command line to be listed as dependent objects in the output file. In Solaris, when a shared object is loaded, all references are resolved by the runtime linker, which searches all the modules currently loaded in the process. In many cases, resolution of function symbols is deferred until the function is first called (references to variables must be resolved at load time).

Deferring resolution allows the definition for a function to be loaded after the module referring to that symbol is loaded. In AIX 6.1, even when the runtime linker is used, all symbols (except the deferred symbols) must be resolved at load time. The runtime linker allows certain symbols, marked as deferred, to pass through at load time in an unresolved state. These symbols then become the responsibility of the application programmer to resolve. In any case, a definition must exist in the process at the time the referencing module is loaded. Although this implementation of the linker in AIX 6.1 puts more pressure on the application programmer for maintaining a well-defined interface, this up-front work consistently yields better runtime results. Starting in AIX 5.3, these deferred-resolution symbols can be forced to be resolved at load-time by exporting the environment variable `LDR_CNTRL=RESOLVEALL` without having to use run-time linking.

The inability to associate a symbol to a specific library on platforms other than AIX makes the order in which the libraries are specified very important. For instance, with respect to `rpc`, if the streams



behavior is required, the library order must be **libnsl** and then **libc**. If sockets behavior is required, it must be **libc** followed by **libnsl**.

For example:

- Module A contains a definition for function X.
- Library foo also has a definition for a different function X.
- Library foo has a function Y which calls function X.

If an application using both of these functions is linked such that A is specified on the command line before foo, and then the following symbol references occur:

- On AIX 6.1, function Y gets the X present in library foo.
- On Solaris, function Y gets the X present in Module A.

If the application is linked with foo listed before A on the command line, then the AIX 5L behavior remains the same. On Solaris, function Y now calls the instance of X found in foo.

An additional feature on AIX 6.1 is the ability to re-ink executable code. It is possible to use the output of the **ld** command as an input to it also. Thus, it is possible to relink new object files (.o's) with an existing application and to replace code in the application without having to completely rebuild the application. This capability can save a considerable amount of time during the application-build cycle.

For a more detailed description of the AIX 6.1 linker and loader mechanisms, various linker options, examples and FAQs, see the paper *AIX Linking and Loading Mechanisms*, which you can find at ibm.com/developerworks/eserver/pdfs/aix_ll.pdf. The paper mainly addresses AIX 4.3.3 and AIX 5L 5.1, but is applicable for the most part to AIX 6.1, as well.

Differences between AIX 6.1 and Solaris for library search-path creation and usage

Consider how the Solaris environment handles the library search path at link time and run time (see Table 6):

Flag	Effect
-L	The -L linker flag gives the path used for searching the libraries specified with the -l flag. The linker uses this.
LD_LIBRARY_PATH #1 and #2	The environment variable in the form of libpath#1[:libpath#2]. Both the linker and loader use this information.
-R	This is a linker command line flag. The linker stores the associated path value in the dynamic section of an ELF object.
LD_RUN_PATH	The linker stores this environment variable path value in the dynamic section of an ELF object.

Table 6. Solaris library search path at link time and run time

Following are the rules that the Solaris environment uses:

- There is only one default search path, /usr/lib, and this is always the last element.
- You must add all other directories to be searched to the search path explicitly.
- -L and LD_LIBRARY_PATH both change the search paths.

Example:

`-L path3`

(`LD_LIBRARY_PATH` has the value of `path1;path2`).

Then, the effective search path for resolving a `-l` library is:

`path1:path3:path2:/usr/lib`

`-R` path name is stored as the `RPATH` information in the dynamic section of an ELF object.

- Note that `-L` and `LD_LIBRARY_PATH` give search paths for searching the `-l` libraries; the `-R` pathname is the `RPATH` that is stored in the ELF for the loader to use at run time.
- If the `RPATH` information is not sufficient for loading the runtime objects, the `LD_LIBRARY_PATH` is used.

Now, take a look at how the AIX 6.1 environment handles the library search-path information (see Table 7):

Flag	Effect
<code>-L</code>	The <code>-L</code> linker flag gives the path used for searching the libraries specified with the <code>-l</code> flag. The linker uses this.
<code>-bllibpath</code>	The path that is to be inserted into the default library search path field (Index 0 path) in the loader section. The linker does this.
<code>-bnolibpath</code>	This does not provide path information and is only a flag for the linker.
<code>#!path</code>	The path that is associated with a shared library or object in an import list. It is to be recorded as a fixed path in the loader section.
<code>LIBPATH</code>	Not for searching for libraries at link time, but it might get recorded in the loader section under certain circumstances.

Table 7. AIX library search path

Here are the rules that the AIX 6.1 environment uses:

- In a link command line, the `-l` library name is searched through the `-L` paths.
- Each `-L` path that contains libraries found is recorded in the Index 0 path line in the loader section, appended with `/usr/lib` and `/lib` at the end. The `LIBPATH` environment-variable value is not stored in this case.
- If an Import List is presented, the paired path and library information on the `#!` line of the import file is recorded in the loader section. The path becomes fixed for the specific library referenced. The specified path is placed in the loader header section and cannot be altered at run time.
- If `-bllibpath:<pathname[:pathname][...]>` is specified on an `ld` command line, the `pathname` information is recorded *verbatim* in the Index 0 line. This is the only library path information that is recorded in the loader section, and the default paths `/usr/lib:/lib` are *not* automatically appended. In this case, no `-L` path is recorded, although locations that are specified with `-L` are still used for searching and choosing `-l` libraries. If neither `-L` nor `-bllibpath` is used in the command line, the `LIBPATH` value is then recorded in the loader section as the Index 0 path. If `-bnolibpath` is specified, only `LIBPATH` is recorded.



A comparison between the Solaris and AIX 6.1 environments for library search paths

- Both Solaris and AIX 6.1 use the linker to store runtime search-path information in the objects that are created.
- Solaris uses LD_LIBRARY_PATH #1, -L path, and LD_LIBRARY_PATH #2 for the linker to search for -l libraries.
- AIX 6.1 uses -L for searching the -l libraries. The AIX 6.1 LIBPATH is not used here.
- Solaris stores the -R (RPATH) and LD_RUN_PATH environment-variable information in the dynamic section of an ELF file.
- AIX 6.1 stores the runtime path information in the loader section of an XCOFF file:
 - Fixed-path information is paired with library names that are obtained from where the Import List is stored.
 - Effective -L paths are stored if -bllibpath is not present.
 - If -bllibpath is present, the associated path is stored with no questions asked.
 - This AIX 6.1 feature is equivalent to the Solaris -R flag but AIX 6.1 does not have the exact equivalent of LD_RUN_PATH.
 - If -bnolibpath is in the command line, the LIBPATH is stored and the -L paths are dropped. This is somewhat similar to the LD_RUN_PATH on Solaris.
- At run time, if the runtime path information contained within the runtime object is not sufficient for resolving the path:
 - Solaris uses LD_LIBRARY_PATH to supplement the path.
 - AIX uses LIBPATH before using the built-in path information.

Memory segments

On AIX, the hardware provides a continuous range of virtual memory addresses, from 0x00000000000000000000 to 0xFFFFFFFFFFFFFFFFFFFFFFFF, for accessing data. The total addressable space is more than 1 trillion terabytes. Memory access instructions generate an address of 64 bits: 36 bits to select a segment and 28 bits to give an offset within the segment.

This addressing scheme provides access to more than 64 million segments of up to 256 MB each. Each segment register contains a 52-bit segment ID that becomes a prefix to the 28-bit offset, which together form the virtual-memory address. The resulting 80-bit virtual address refers to a single, large, system-wide virtual memory space. The process space is a 64-bit address space; that is, programs use 64-bit pointers. However, each process or interrupt handler can address only the systemwide virtual memory space (segment) whose segment IDs are in the segment register.

A 64-bit application program that is running on the system has an address space that is divided into segments, as follows (see Table 8):

Memory address range	Description
0x0000 0000 0000 0000 - 0x0000 0000 0fff ffff	Contains the kernel
0x0000 0000 d000 0000 - 0x0000 0000 dfff ffff	Contains 32-bit shared library text (inaccessible to 64-bit apps)
0x0000 0000 e000 0000 - 0x0000 0000 efff ffff	Shared memory segment available to 32-bit applications



0x0000 0000 f000 0000 - 0x0000 0000 0fff ffff	Contains 32-bit shared library data (inaccessible to 32-bit applications)
0x0000 0001 0000 0000 - 0x07ff ffff ffff ffff	Contains the application program text and data, plus segments for shared memory or mmap services
0x0800 0000 0000 0000 - 0x08ff ffff ffff ffff	Privately loaded 64-bit modules
0x0900 0000 0000 0000 - 0x09ff ffff ffff ffff	64-bit shared library text and data
0x0f00 0000 0000 0000 - 0x0fff ffff ffff ffff	64-bit application stack

Table 8. Program address space

On AIX, a 64-bit application is always loaded at 0x0000000100000000. Note that the kernel is mapped into *low* memory, starting at address 0x0. Therefore, although 32-bit pointers cannot be used to access any portion of the 64-bit application address space, it is possible to read regions of the kernel segment (in low memory) using a 32-bit pointer.

32-bit address space

In Table 9, note how the 32-bit address space is portioned out. A 32-bit application program has an address space that is divided into seven regions, as follows:

Memory Address Range	Description
0x00000000 to 0x0fffffff	Contains the kernel
0x10000000 to 0x1fffffff	Contains the application program text
0x20000000 to 0x2fffffff	Contains the application program data and the application stack
0x30000000 to 0xcfffffff	Available for use by shared memory or mmap services
0xd0000000 to 0xdfffffff	Contains shared library text
0xe0000000 to 0xefffffff	Contains miscellaneous kernel data; also available for use by shared memory or mmap services
0xf0000000 to 0xffffffff	Contains the application shared library data

Table 9. 32-bit address space in seven segments

For more information on the AIX memory model, see the AIX online documentation at:

<http://publibn.boulder.ibm.com/doc link/Ja JP/a doc lib/aixprggd/genprogc/address space.htm#HD RA9CF64F8555SYLV>.

Runtime issues — 32- compared to 64-bit kernel

AIX 6.1 can be started with either the 32- or 64-bit kernel. Beginning with AIX 6.1, all 64-bit applications will run on either kernel (provided you are on 64-bit hardware), as will all 32-bit applications. However, if the most critical applications running on the system are 64-bit, it is advisable to boot the server with the 64-bit kernel to take full advantage of the hardware capabilities.



The 64-bit kernel is packaged with the `bos.mp64` file set. Installation of this file set is automatic if you select the option `Enable 64-bit Kernel` and `JFS2` under `Advanced Options` on the `Installation and Settings` screen. This selection will also cause the 64-bit kernel to boot by default. Note that when you install on 64-bit hardware these options are enabled by default.

During the normal installation process, the 32-bit kernel is enabled by default on 32-bit hardware. On 64-bit hardware, after the system is installed, either kernel can be selected for boot use with the following procedure:

```
> ln -sf /usr/lib/boot/unix_xx /unix
> ln -sf /usr/lib/boot/unix_xx /usr/lib/boot/unix
> bosboot -ad /dev/ipldevice
> shutdown -r
```

where `unix_xx` is:

```
unix_mp      - 32 bit mp kernel
unix_64      - 64 bit mp kernel
```

AIX 4.3 compiled 32-bit and 64-bit applications on AIX 6.1

32-bit applications that have been compiled on AIX 4.3 and have followed the binary compatibility rules are binary compatible with AIX 6.1. These applications will run *as is* and do not need to be recompiled to run on AIX 6.1. But 64-bit applications that originated on AIX 4.3 need to be recompiled on AIX 6.1 to run on AIX 6.1.

32-bit and 64-bit applications on AIX 6.1

32-bit applications are compiled such that memory addresses are 32-bit (four bytes) in size. These applications can directly exploit up to four gigabytes of virtual memory — the memory potentially available for use on a computer. This virtual memory constraint is present regardless of the amount of real memory (RAM) available on the system to be shared between the operating system and other applications. 64-bit applications on the other hand are compiled such that memory addresses are 64-bits (eight bytes) in size and can use more than four gigabytes of virtual memory without restriction. Operating systems typically also impose additional virtual memory restrictions on applications, and the theoretical maximum virtual memory per application might be as little as one to two gigabytes, even though it can have 32-bit addressing capability.

When you compile an application on a 32- or 64-bit platform, by default it is generally compiled to run on that particular platform. You can create 64-bit applications on 32-bit operating systems and 32-bit applications on 64-bit operating systems with some compilers by using special compiler-specific compile options and by appropriately linking to 32-bit or 64-bit libraries where appropriate.

32-bit applications can generally be run on both 32- and 64-bit operating systems, although many operating systems cannot run 64-bit applications if they employ a 32-bit kernel.

The max amount of data segment and stack segment for 32-bit applications are 2 GB (default is 256 MB) and 256 MB, respectively. To fully use this resource, compile or link the applications with — `bmaxdata:0x70000000` and `-bmaxstack:0x10000000` options. 64-bit applications do not have these limitations.



Performance tuning

A significant difference between AIX 6.1 and Solaris (as well as other UNIX systems) is that the end user or system administrator does not modify kernel configuration files and, therefore, does not need to rebuild the kernel. All tunable parameters in AIX 6.1 can be adjusted through the `smit tuning` command (“`smitty tuning`”), as well as the `vmo` and `schedo` commands.

A large body of online documentation is available on the topic of performance tuning. See the *Performance Management Guide* for more information on application- and system-tuning parameters available on AIX 6.1.

Threads programming

Applications that use the Solaris threads programming interface, for example, `thr_create()`, `mutex_lock()`, and `cond_signal()` function calls, are generally not portable. To port an application that uses this interface to AIX 6.1, you need to do some work. The main areas of Solaris thread functions that are not supported in POSIX threads are:

- Daemon threads (`thr_create`).
- Joining any thread (Solaris allows you join any thread with a `tid 0`).
- Thread suspend and thread continue.

Solaris supports the POSIX threads API as well as the Solaris threads API. If you want to make a Solaris threaded application portable, we recommend that you re-code the threaded part of the application to use POSIX threads. However, the Solaris threads API has some unique functions over the POSIX threads API, which can make the task of converting a Solaris threaded application to use POSIX threads very time-consuming and difficult. Depending upon the application, the rework can vary from a few editing substitutions to re-engineering the application.



Determining whether an application uses Solaris or POSIX threads

To confirm whether the application on Solaris is using the POSIX threads or Solaris threads, check the name of the threads library. On Solaris, the user has to include declarations to link to Solaris Threads Library (STL). If it is linked with “-lpthreads”, it is using the POSIX thread library.

Table 10 compares the Solaris Thread API to the POSIX threads API.

POSIX Threads API	Solaris Threads API	Operation
pthread_create()	thr_create()	Create a new thread of control
pthread_exit()	thr_exit()	Terminate the execution of the calling thread
pthread_getschedparam()	thr_getprio()	Retrieve a thread's priority parameters
pthread_getspecific()	thr_getspecific()	Bind a new thread specific value to the key
pthread_join()	thr_join()	Suspend calling thread until target thread completes
pthread_key_create()	thr_keycreate()	Create a key that locates data specific to a thread
pthread_kill()	thr_kill()	Send signal to another thread
pthread_self()	thr_self()	Return the thread ID of the calling process
pthread_setschedparam()	thr_setprio()	Modify a thread's priority parameters
pthread_setspecific()	thr_setspecific()	Bind a new thread specific value to the key
pthread_sigmask()	thr_sigsetmask()	Change/examine calling thread's signal mask
sched_yield()	thr_yield()	Make the current thread yield to another thread
pthread_setconcurrency()	thr_setconcurrency()	Set thread concurrency level
pthread_getconcurrency()	thr_getconcurrency()	Get thread concurrency level
pthread_setspecific()	thr_setspecific()	Set the thread specific data key
pthread_getspecific()	thr_getspecific()	Get the thread specific data key
Not Supported	thr_suspend	Suspend the execution of the specified thread
Not Supported	thr_continue	Resume the execution of a suspended thread

Table 10. Comparison of threads APIs

The POSIX standard provides no mechanism by which thread A can suspend the execution of another (thread B) without cooperation from thread B. The only way to implement a suspend/restart mechanism is to have thread B check periodically some global variable for a suspend request and then suspend itself on a condition variable, which another thread can signal later to restart thread B.

Comparing the POSIX thread library and the Solaris thread library

Solaris supports both the POSIX 1003.1c and the proprietary semaphore API. Table 11 maps the POSIX 1003.1c API to the proprietary Sun API. Solaris defines these functions in `libthread`: AIX 6.1 defines them in `libpthreads`.

POSIX Library (libpthreads)	Solaris Threads Library (libthread)	Operation
<code>pthread_mutex_destroy()</code>	<code>mutex_destroy()</code>	Destroy or disable the state associated with the mutex object
<code>pthread_mutex_init()</code>	<code>mutex_init()</code>	Initialize the mutex variable
<code>pthread_mutex_lock()</code>	<code>mutex_lock()</code>	Lock the mutex object, block until mutex object is free
<code>pthread_mutex_unlock()</code>	<code>mutex_unlock()</code>	Release the mutex object
<code>pthread_cond_broadcast()</code>	<code>cond_broadcast()</code>	Unblock all threads waiting on the condition variable
<code>pthread_cond_destroy()</code>	<code>cond_destroy()</code>	Destroy any state associated with the condition variable
<code>pthread_cond_init()</code>	<code>cond_init()</code>	Initialize a condition variable
<code>pthread_cond_signal()</code>	<code>cond_signal()</code>	Unblock the next thread waiting on the condition variable
<code>pthread_cond_wait()</code>	<code>cond_wait()</code>	Block on a condition variable, release it finally
<code>pthread_rwlock_init()</code>	<code>rwlock_init()</code>	Initialize a read-write lock
<code>pthread_rwlock_destroy()</code>	<code>rwlock_destroy()</code>	Lock a read-write lock
<code>pthread_rwlock_rdlock()</code>	<code>rw_rdlock()</code>	Read lock on a read-write lock
<code>pthread_rwlock_wrlock()</code>	<code>rw_wrlock()</code>	Write lock on a read-write lock
<code>pthread_rwlock_unlock()</code>	<code>rw_unlock()</code>	Unlock a read-write lock
<code>pthread_rwlock_tryrdlock()</code>	<code>rw_tryrdlock()</code>	Read lock with a nonblocking read-write lock
<code>pthread_rwlock_trywrlock()</code>	<code>rw_trywrlock()</code>	Write lock with a nonblocking read-write lock

Table 11. Mapping the POSIX 1003.1c API to the proprietary Sun API

In Solaris applications that use the proprietary threads API rather than the POSIX threads, the `fork()` function duplicates in the child process all the threads and Light Weight Processes (LWPs) of the parent process. The `fork1()` function duplicates only the calling thread in the child process.

In applications that use the POSIX threads API, however, a call to `fork()` is like a call to `fork1()`, which replicates only the calling thread. There is no call that forks a child with all threads and LWPs duplicated in the child.

Note that if a program was linked with both libraries (`-lthread` and `-lpthread`) on Solaris, the POSIX semantic of `fork()` probably prevailed at run-time.



Signals

AIX 6.1 provides the POSIX , BSD and System V style subroutine functions, so that you can easily handle most signal code encountered during ports. For a detailed listing of the signals supported by AIX 6.1 and other UNIX implementations; see Chapter 7, section 7.6 of the IBM Redbooks title, *AIX 5L Porting Guide*.

Twelve AIX 6.1 tools similar to the Solaris /proc tools

AIX V5.2 and later provides twelve simple utilities that are similar to tools provided on Solaris. The 12 utilities are very useful for debugging and analyzing process behavior. The tools (see Table 12) display information on a process using `/proc` data.

AIX VERSION 5.2 and later tools	Description
proctree	Displays process tree containing a process
procstack	Dumps current stack of a process
procmmap	Displays a process address map
procldd	Displays list of libraries loaded by a process
procflags	Displays a process tracing flags and pending signals
procsig	Lists the signal actions for a process
procfiles	Prints list of open file descriptors
proccred	Prints a process credentials
procwdx	Prints current working directory for a process
procstop	Stops a process
procrun	Restarts a process
procwait	Waits for a process to terminate

Table 12. Twelve AIX 6.1 utilities



Porting issues

This section describes some of the most commonly encountered issues when you port applications from Solaris to AIX 6.1.

Solaris `gcore` and `coreadm` commands

The Solaris `gcore` command provides a snapshot (core) of a running process so that developer can take a look at what the different threads were doing. On AIX 5.2 and later, the `gencore` command provides a similar function.

The `coreadm` command provides the ability to custom name a core file so that

The core files do not overwrite each other. On AIX 6.1, export the environment variable `CORE_NAMING=true`. The naming convention is

`core.<pid>.<timestamp>`. AIX version v6.1 uses the `chcore` command to specify a default location for core files on a per user basis or system-wide. For complete details, see <http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.cmds/doc/aixcmds1/chcore.htm>.

Compiler errors

The Sun Forte compiler is more forgiving of nonstandard constructs than is the IBM XL C/C++ v10.1 compiler. Unexpected compiler errors can occur. Some known errors are described here, along with recommended solutions.

Namespace

Namespace is very strict on AIX 6.1, especially within the Standard Template Library (STL).

Expressions like `use std`, `use std::string`, or `use std::vector` are required after `#include` header files to avoid compiler errors.

Containers

Some containers such as `map` and `vector` do not allow `const` type by the XL C++ v10.1 compiler. Specifically, `std::map< const std::string, Foo>` will not compile with the XL C++ v10.1 compiler. The `const` is unnecessary because `map` keys are immutable anyway. The key type is made `const` when a (key, value) pair is returned, but it must not be declared `const` in the `map` template.

Casting

Casting is strictly enforced by the XL C++ v10.1 compiler. For example, `const void *` cannot be cast into non-const pointers.

`fstream`

On Solaris, the `fstream::rdbuf().fd()` method is used to get a file descriptor from the `fstream` object, as in the following example:

```
tout = new ofstream(_filename, ios::app, 0666);
```



```
if(tout->good()) {
    dup2(tout->rdbuf()->fd(), 1);
}
```

The method `fd()` is not available in `basic_filebuf` class on AIX 6.1.

However, you can create a `filebuf` object and call it `mybuf`, with `mybuf(myfile)`. You can then associate that buffer with an `ifstream` or `ofstream` object `str` by calling its member function `str.rdbuf(mybuf)`, as in the following example:

```
#ifdef SUN5
tout = new ofstream(dup(1)); //attach stdout to it...
#else // defined AIX 6
tout = new ofstream();
FILE *fp = fdopen(dup(1), "a+");
filebuf *mybuf = new filebuf(fp);
((basic_ios<char, char_traits<char> > *)tout)->rdbuf(mybuf);
#endif

if(tout->good()) {
#ifdef SUN5
dup2(tout->rdbuf()->fd(), 1);
#else // defined AIX 6
dup2(fd, 1);
#endif
}
```

fdopen

The `fdopen` subroutine associates a stream with a file descriptor obtained from an `openx` subroutine, `dup` subroutine, `creat` subroutine, or `pipe` subroutine. These subroutines (such as `dup`) open files, but they do not return pointers to FILE structures. Many of the standard I/O package subroutines require pointers to FILE structures.

On AIX 6.1, when you call `fdopen` with a file descriptor obtained from the `dup` subroutine, the duplicated file descriptor must open the file with the same privileges. For example, if the file was originally created with read-only permission, then the stream must also have read-only permission. This is specified in the `Type` parameter for the `fdopen` subroutine call:

```
FILE *fdopen (FileDescriptor, Type)
int FileDescriptor;
const char *Type;
```

The `Type` parameter for the `fdopen` subroutine specifies the mode of the stream, such as `r` to open a file for reading or `a` to open a file for appending (writing at the end of the file). The mode value of the `Type` parameter specified with the `fdopen` subroutine must agree with the mode of the file specified when the file was originally opened or created.

char constant reference

The following code compiles on Sun Forte compiler but fails on AIX 6.1:

```
#include <iostream.h>
class test
{
public:
// Using a constant ref. to a variable whose value can be changed
void test1 ( char & const newintrlEvent) {cout << newintrlEvent <<
endl;}
};

int main()
{
test t;
char * const ptr="ABC";
t.test1(*ptr);
return 0;
}
```

Here's the code that compiles on AIX 6.1:

```
#include <iostream>
class test
{
public:
// Using a constant ref. to a variable whose value can be changed
void test1 ( char const & newintrlEvent) {cout << newintrlEvent <<
endl;}
};

int main()
```

```

{
test t;
char * const ptr="ABC";
t.test1(*ptr);
return 0;
}

```

Table 13 shows different code samples side-by-side. Take a minute to review them.

Sun Forte	AIX 6.1 XL Compiler
<pre> #include <iostream.h> class test{ public: // Using a constant ref. to a // variable whose value can be // changed </pre>	<pre> #include <iostream.h> class test{ public: // Using a constant ref. to a // variable whose value can be // changed </pre>
<pre> void test1(char& const nEvt){ </pre>	<pre> void test1(char const & nEvt){ </pre>
<pre> cout << nEvt << endl; } }; int main() { test t; char * const ptr="ABC"; t.test1(*ptr); return 0; } </pre>	<pre> cout << nEvt << endl; } }; int main() { test t; char * const ptr="ABC"; t.test1(*ptr); return 0; } </pre>

Table 13. Comparison of code samples

The `char& const` is different from `char const &`. Note that `char& const` is a constant reference to a char; the `char const &` is a reference to a constant char.

const reference

The following code fragment compiles on Solaris but gives error messages on AIX 6.1.

```

class T {
public:
T(int);
};

class U {
public:
U(T& k=T(0));
};

```

Here are the error messages that result on AIX 6.1:

```

"test.cpp", line 10.16: 1540-1280 (S) An rvalue of type "T" cannot be
converted to "T &".

```



"test.cpp", line 10.16: 1540-1290 (I) An rvalue cannot be converted to a reference to a non-const type.

To get this code fragment to compile clean on AIX 6.1, we need to add `const` to the parameter. Table 14 shows the code fragment as it appears on Solaris and AIX 6.1:

Solaris	AIX 6.1
<pre>class T { public: T(int); }; class U { public: U(T& k=T(0)); };</pre>	<pre>class T { public: T(int); }; class U { public: U(const T&k=T(0)) };</pre>

Table 14. Comparison of code fragments

The problem with making the parameter non-const is that the constructor can try to update the reference. Since the reference points to a temporary value, the update is then be lost. The way to make this code fragment compile on AIX 6.1 is to use `U(const T&k=T(0))`. This use also makes the code conform to C++ standards.

Storing returned object in a reference

The code fragment in Table 15 compiles on Solaris (in the left column) but gives error messages on AIX 6.1. In the right column, two solutions are shown that compile correctly on AIX 6.1.

Solaris	AIX 6.1
<pre>class foo { public: foo(); }; foo getFoo() { foo myFoo; return myFoo; } void myFunc() { foo& myFoo = getFoo(); }</pre>	<pre>class foo { public: foo(); }; foo getFoo() { foo myFoo; return myFoo; } void myFunc() { <i>Solution 1:</i> const foo & myFoo = getFoo(); <i>Solution 2:</i> foo tmpFoo =getFoo(); foo& myFoo = tmpFoo; }</pre>

Table 15. Comparison of code samples



A temporary value, such as produced by the expression `getFoo()`, cannot bind to a non-const reference, according to the C++ standard. The AIX 6.1 XL C++ compiler adheres to that requirement.

Copy constructors, equals operator, and comparison operator

The IBM XL C++ v10.1 compiler requires copy constructors, equals operator, and comparison operator parameters to sometimes be `const`. Therefore, the general rule is to make all the parameters `const` if possible to avoid errors. Table 16 shows an example showing this rule:

Solaris	AIX 6.1
<pre>Foo(Foo&) Foo & operator = (Foo &) Boolean operator == (Foo &, Foo &)</pre>	<pre>Foo(const Foo&) Foo & operator = (const Foo &) Boolean operator == (const Foo &, const Foo &)</pre>

Table 16. Comparison of parameters

friend usage

Using `friend` does not make it a class. You must forward declare `class` prior to making it `friend`. See the code examples in Table 3.

Solaris	AIX 6.1
<pre>class Foo{ friend class FooFriend; };</pre>	<pre>class FooFriend; class Foo { friend class FooFriend; };</pre>

Table 23. Comparison of declaring friend

Static friend usage

The Solaris Forte C++ compiler allows the `friend` function to be static, but the IBM XL C++ v10.1 compiler does not allow it. Do not specify the friend function as static.

Class qualifiers

The AIX 6.1 XL C++ v10.1 compiler does not allow usage of class qualifier `::` within a class definition, as illustrated in Table 24174:

Solaris Forte	AIX 6.1 XL C++ compiler
<pre>class FooNext { void FooNext::getFoo(); };</pre>	<pre>class FooNext { void getFoo(); };</pre>

Table 2417. Comparison of class qualifiers

Text after #endif

The IBM XL C++ v10.1 compiler does not allow extra non-comment text following `#endif` statements, as Table 185 shows:

Solaris Forte	AIX 6.1 XL C++ compiler
#endif AIX 6	#endif //AIX 6

Table 185. Comparison of #endif statements

Linked lists

The following code fragments compile correctly on Solaris but causes an error message when compiled with the IBM XL C++ v10.1 compiler:

```

/* Type definition for a linked list and linked list pointer. */
class LinkList : public Node {
    int    _listlength;    /* no. of elements in linked list */
    int    _itemsize;     /* size of the data _item (see _NODE) */
public :
    Node   *_head;        /* pointer to _head of linked list */
    Node   *_tail;        /* pointer to _tail of linked list */
    Node   *_current;     /* pointer to _current node */
    -
    -
}

LinkList operator++(){
    if( _current->_next )
        _current = _current->_next;
    else
        _current = NULL;
    return *this;
}

-
-
LinkList LinkList::operator=(LinkList &from_list)
{
    char *from_item,
        *to_item;
    LinkList _list(sizeof(this->_head));
    -
    -
    from_list++;
    -
}

```

Following is the error message for the `from_list++` line:

```

1540-0218 (S) The call does not match any parameter list for
"operator++".
1540-0215 (I) The wrong number of arguments have been specified for
"LinkList::operator++()".
1540-1283 (I) "LinkList::operator++()" is not a viable candidate.

```

To make this code work with the IBM XL C++ v10.1 compiler, change the declaration from

```
LinkList operator++() {
```

to

```
LinkList operator++(int ){
```

The reason is that if the compiler sees `++b`, for example, it generates a call to `B::operator++()`. If the compiler sees `b++(int)`, it calls `B::operator++(int)`.

Comments in C++ style

To enable comments in the C++ style in a C source file, use the flag.

```
-qcplusplus.
```

ENONET error

AIX 6.1 does not support the ENONET error symbol (“Machine is not on the network”) which is available on Solaris. A comparable error symbol might be ENETUNREACH (“Network is unreachable”).

Macro MAXHOSTNAMELEN

The macro MAXHOSTNAMELEN is defined in `netdb.h` on Solaris; this macro is defined in `sys/param.h` on AIX 6.1.

System call differences

Several system calls used by an application running on Solaris might not be available on AIX 6.1. Here are some known ones and the recommended solution to make the application work on AIX 6.1.

High resolution real time call

AIX 6.1 does not support the `gethrtime(void)` call. To read the processor real time clock, add the following:

```
#include <sys/time.h>
#include <sys/systemcfg.h>
int read_real_time(timebasestruct_t *t, size_t
Size_of_timebasestruct_t);
```

To obtain time base register high resolution elapsed time, add the following:

```
int time_base_to_time(timebasestruct_t *t,
size_t size_of_timebasestruct_t);
```

Socket on top of stream operation

Solaris implements sockets on top of streams. Because AIX 6.1 sockets are not streams-based and there is no stream-head read queue, the `ioctl` call with `I_NREAD` operation does not make sense on an AIX 6.1 socket. The call can be implemented as follows on AIX 6.1:

```
#ifdef _AIX
#include <sys/ioctl.h>
#include <sys/types.h>
#include <unistd.h>
#else
#include <stropts.h>
#endif
#ifdef _AIX
```

```

if (ioctl ( sock, FIONREAD, length) == SOCKET_ERROR)
#else
if (ioctl ( sock, I_NREAD, length) == SOCKET_ERROR)
#endif
{
IPC_ERR_TRACE("failed to retrieve pending data size")
return (WAP_FAIL);
};

```

econvert(), fconvert(), and gconvert()

Equivalent routines on AIX 6.1 are:

```
ecvt(), fcvt(), and gcvt()
```

clock_gettime()

In Solaris, `clock_gettime` links with `-lrt`. On AIX 6.1, `clock_gettime` links with `libc.a`.

Unresolved seteuid() and setegid() calls

To resolve undefined symbols `setegid()` and `seteuid()` when compiling/linking, add the following:

```

#include <stdio.h>
#include <unistd.h>

extern "C" {
int seteuid(uid_t);
int setegid(gid_t);
}

```

Access process information

In Solaris, there is a structure `prpsinfo`, which contains process information, and `ioctl(fd, request, .../*arg*/)`, which populates this structure. AIX 6.1 does not support the Solaris process file system in the same manner; therefore, this `open()` call fails. AIX 6.1 adds the `/proc` file system but the `/proc` details are different. For more information, see Programming issues section of this paper.

```

char path[64];
sprintf(path, "/proc/%lu", (unsigned long) getpid());
int pfd = open(path, O_RDONLY);
struct prpsinfo processinfo;

```

The preceding example can be implemented like this on AIX 6.1:

```

int pfd = (unsigned long) getpid();
long psize = sysconf(_SC_PAGESIZE);

struct procinfo processinfo;
memset(&processinfo, 0, sizeof(processinfo));

while(1){
    if(getprocs(&processinfo, sizeof(processinfo), (size_t)0, 0, &pfd, 0) !=
-1){

```

```
.....
}
```

Signals

In Solaris, the `sig2str()` and `str2sig()` are specific calls used for signal name conversion back and forth from a “signal” to a “str”.

In AIX 6.1, use the `sys_siglist` array exported from `libc.a (shr.o)`:

```
extern char *sys_siglist[];

int main(int argc, char *argv[])
{
    char *str;
    int sig;

    sig = atoi(argv[1]);
    str = sys_siglist[sig];
    printf("Signal %d -> \"%s\"\n", sig, str);
}
```

SIG_PF signal

In Solaris, the `SIG_PF` is defined in `iso/signal_iso.h`. AIX 6.1 does not support `SIG_PF`. Therefore, use a macro:

```
#define SIG_PF (void*)(int)
```

cftime()

AIX 6.1 does not have the `cftime()` function in the C library:

```
int cftime(char *s, char *format, const timet *clock);
```

But AIX 6.1 has the `strftime()` function, which does roughly the same thing:

```
size_t strftime (string,length,*Format,*Tmdate)
char *String;
size_t Length;
const char *Format;
const struct tm *TmDate;
```

The `strftime()` subroutine converts the internal time and date specification of the `tm` structure, which is pointed to by the `TmDate` parameter, into a character string pointed to by the `String` parameter. The `String` parameter is under the direction of the format string pointed to by the `Format` parameter. The actual values for the format specifiers are dependent on the current settings for the LC TIME category. The `tm` structure values can be assigned by the user or generated by the `localtime()` or the `gmtime()` subroutine.

The resulting string is similar to the result of the `printf` `Format` parameter and is placed in the memory location addressed by the `String` parameter. The maximum length of the string is determined by the `Length` parameter and terminates with a null character.

dladdr() — Calculating symbolic addresses

The `dladdr()` function in Solaris queries the dynamic linker for information about the shared object containing the address `addr`. This API is not supported on AIX 6.1.

On AIX 6.1, the `dlopen()` function is used to open a shared object and dynamically map it into the address space of the running program. However, there is no easy way to get the nearest symbol name if the address is passed in it.

A workaround is to look at the trace back table at the end of each function (see `/usr/include/sys/debug.h`). This includes the name of the function. Using this, you can convert an instruction address into a function name.

You also can use the following example:

```
$ cat tryit.c
extern int malloc();
main() {
    int *x;
    x = (int *)&malloc;
    printf("Code for malloc is at %p\n", *x);
}

$ cc -o tryit tryit.c
$ ./tryit
```

The preceding code fragment produces the following output:

```
Code for malloc is at d016bc64 ---> Note this value.
```

Now, let's use the debugger to calculate these addresses. Note that we use `dbx` for simplicity. However, `loadquery` is more helpful in figuring out the starting of text for each shared library.

```
$ dbx tryit
Type 'help' for help.
reading symbolic information ...warning: no source compiled with -g

(dbx) map
Entry 1:
  Object name: tryit
  Text origin: 0x10000000
  Text length: 0x67f
  Data origin: 0x200003d8
  Data length: 0xf8
  File descriptor: 0x4

Entry 2:
  Object name: /usr/lib/libcrypt.a
  Member name: shr.o
  Text origin: 0xd00250f8
  Text length: 0x87a
  Data origin: 0xf0870528
  Data length: 0x13c
  File descriptor: 0x5

Entry 3:
```



```
Object name: /usr/lib/libc.a
Member name: shr.o
Text origin: 0xd015f720
Text length: 0x1bed86
Data origin: 0xf07f1340
Data length: 0x7ea10
File descriptor: 0x6
```

```
(dbx) q
```

```
libc.a(shr.o) starts at 0xd015f720
```

```
$ ar -x /usr/lib/libc.a shr.o
```

```
$ dump -hv shr.o
```

```
shr.o:
```

```
          ***Section Header Information***
          Section Header for .text
PHYaddr   VTRaddr   SCTsiz   RAWptr   RELptr
0x00000000 0x00000000 0x0014fab0 0x00000160 0x001bed9e
          ^^^^^^^^^^          ^^^^^^^^^^
.....
.....
```

Note the values of the virtual address (0x00000000) and RAWptr (0x00000160).

You can use the following command to get these values.

```
$ nm -x shr.o | grep ".malloc" | grep " T "
__malloc          T 0x00008000
__malloc_init     T 0x0000b26c
__malloc_postfork_unlock T 0x0000b500
__malloc_prefork_lock T 0x0000b61c
.malloc           T 0x0000c3e4
.malloc_s         T 0x0000d6e0
.malloc_y         T 0x0000ec6c
```

Note that nm for .malloc is 0x0000c3e4.

Now we can use the information that we have gathered so far to calculate the location of malloc with the following formula:

```
memory_address(malloc) = nm(.malloc) + text_origin - virtaddr(.text) + rawoffset(.text)
$ bc
ob=16
ib=16
C3E4 + D0015F720 - 0 + 160
D016BC64
```

Programming issues

Here are some programming issues that you might encounter.

IPC limits

IPC mechanisms (semaphore, shared memory, and message queues) are commonly used by UNIX programmers to provide communication among multiple processes. On Solaris, the application programmers communicate their IPC resource requirement to the system administrator who has to edit `/etc/system` and set the limits for IPC mechanisms (semaphore, shared memory segments, and message queues). Examples of entries in `/etc/system` are:

- **shmsys:shminfo_shmmax** — Maximum size of System V shared memory segment that can be created
- **shmsys:shminfo_shmmni** — System wide limit on number of shared memory segments that can be created
- **semsys:seminfo_semmni** — Maximum number of semaphore identifiers
- **semsys:seminfo_semmns** — Maximum number of System V semaphores on the system
- **msgsys:msginfo_msgtql** — Maximum number of messages that can be created

The problem with this method is that the higher the limits are set, the bigger the kernel gets, and performance can be adversely affected. On the other hand, if the limits were set too low and are exceeded at run time, the system administrator has to change the values in `/etc/system` and reboot the system to make the new values become effective.

AIX 6.1 uses a different method. On AIX 6.1, IPC limits are handled for users. The kernel has hard upper limits for IPC mechanisms, and the individual IPC types are dynamically allocated and deallocated up to these upper limits. The applications always get the maximum number of IPC resources allowed by the kernel. No editing of `/etc/system` is needed, and no rebooting is needed to raise the limits.

Therefore, the kernel grows and shrinks in size as IPC types are allocated, so any performance hit is only for the life of the IPC type. The structures containing IPC limits are defined in three files in `/usr/include/sys/`, `sem.h`, `msg.h`, and `shm.h`. The structures themselves are called `seminfo`, `msginfo`, and `shminfo`, respectively.

Table 19, Table 20 and Table 21 summarize the IPC limits on semaphore mechanisms.

Semaphores							
AIX 6.1 versions	4.3.0	4.3.1	4.3.2	5.1	5.2	5.3	6.1
Maximum number of semaphore IDs using 64-bit kernel	4 096	4 096	131 072	131 072	131 072	1 048 576	1 048 576
Maximum semaphores per semaphore ID	65 535	65 535	65 535	65 535	65 535	65 535	65 535
Maximum operations per semaphore call	1 024	1 024	1 024	1 024	1 024	1 024	1 024
Maximum undo entries per process	1 024	1 024	1 024	1 024	1 024	1 024	1 024
Size in bytes of undo structure	8 208	8 208	8 208	8 208	8 208	8 208	8 208



Semaphore maximum value	32 767	32 767	32 767	32 767	32 767	32 767	32 767
Adjust on exit maximum value	16 384	16 384	16 384	16 384	16 384	16 384	16 384

Table 19. Comparison of semaphores in AIX versions

Message Queues

AIX 6.1 versions	4.3.0	4.3.1	4.3.2	5.1	5.2	5.3	6.1
Maximum message size	4 MB	4 MB	4 MB	4 MB	4 MB	4 MB	4 MB
Maximum bytes on queue	4 MB	4 MB	4 MB	4 MB	4 MB	4 MB	4 MB
Maximum number of message queue IDs using 64-bit kernel	4 096	4 096	131 072	131 072	131 072	1 048 576	1 048 576
Maximum messages per queue ID	524 288	524 288	524 288	524 288	524 288	524 288	524 288

Table 20. Comparison of message queues in AIX versions

Shared Memory

AIX 6.1 versions	4.3.0	5.1	5.2	5.3	6.1
Maximum segment size (32-bit process - 32-bit kernel)	256 MB	2 GB	2 GB	2 GB	2 GB
Maximum segment size (64-bit process - 32-bit kernel)	256 MB	64 GB	1 TB	1 TB	1 TB
Maximum segment size (64-bit process - 64-bit kernel)	Not applicable	64 GB	1 TB	32 TB	32 TB
Minimum segment size	1	1	1	1	1
Maximum number of shared memory IDs (32-bit process)	4 096	131 072	131 072	131 072	131 072
Maximum number of shared memory IDs (64-bit process)	4 096	131 072	131 072	1 048 576	1 048 576
Maximum number of segments per process (32-bit)	11	11	11	11	11
Maximum number of segments per process (64-bit)	268 435 456	268 435 456	268 435 456	268 435 456	268 435 456

Table 21. Comparison of shared memory in AIX 6.1 versions

The increase in the maximum number of IDs to 1,048,576 in AIX 6.1 requires use of the 64-bit kernel. The maximum number of IDs when using the 32-bit kernel has not changed from AIX version 5.2

The shared memory limit and using EXTSHM

The limit that might cause a problem is the maximum number of shared memory segments that are available simultaneously per process. This number is 11. If the application needs more, the **extended shmat capability** has to be enabled using the environment variable `EXTSHM`. Here is an explanation of how extended shared memory works.

By default, each shared memory region (whatever its size) always consumes a 256 MB region of address space. AIX 4.2.1 and later implements Extended Shared Memory, which allows for more granular shared memory regions that can range size from 1 byte up to 256 MB. However, the address space consumption will be rounded up to the next page (4096 byte) boundary. Extended Shared Memory essentially removes the limitation of only 11 shared-memory regions.

This feature is available to processes that have the variable `EXTSHM` set to `ON` (`EXTSHM=ON`) in their process environment. Then, there is no limit on the number of shared memory regions that a process can attach. File mapping is supported as before, but it still consumes address space that is a multiple of 256 MB (segment size). Resizing a shared memory region is not supported in this mode. Kernel processes will still have the same behavior. Without this environment variable set, 11 regions of 256 MB are available.

Note: When a shared memory segment larger than 256 MB is attached, its size is rounded to a multiple of 256 MB, even if the extended `shmat` capability is being used.

Extended Shared Memory has the following restrictions:

- I/O support is restricted in the same manner as for memory-mapped regions.
- Only the `uphysio()` type of I/O is supported (no raw I/O).
- These shared memory regions cannot be used as I/O buffers where the unpinning of the buffer occurs in an interrupt handler. For example, these regions cannot be used for async I/O buffers.
- The segments cannot be pinned using the `plock()` subroutine because memory-mapped segments cannot be pinned with the `plock()` subroutine.

RPC

AIX 6.1 supports two implementations of RPC — OSF RPC and TIRPC. The header files for OSF RPC are in `/usr/include/rpc`, and the header files for TIRPC are in `/usr/include/tirpc/rpc`.

OSF RPC uses the BSD sockets to communicate, whereas the TIRPC uses the TLI interface to communicate and which in turn uses the streams and Transport Layer interface. TIRPC, originally developed by Sun Microsystems, is multi-threaded and more popular than OSF RPC. If you want to use TIRPC, the program has to link with `/usr/lib/libnsl.a`.

The AIX 6.1 man pages provide information on OSF RPC, but not on TIRPC. For additional information on TIRPC, go to:

<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.cmds/doc/aixcmds4/rpcgen.htm&resultof=%22TIRPC%22%20&searchQuery=TIRPC&searchRank=0&pageDepth=0>


```
#include <signal.h>
typedef struct stack_frame {
    struct stack_frame *next;
    int        unused;
    int        pc;
} stackFrame;

void stackTrace()
{
    stackFrame *sf;
    (void)getContext(&uc);
    for(sf=(stackFrame*)uc.uc_mcontext.jump_context.gpr[1]; sf != NULL;
        sf = sf->next)
    {
        fprintf(stdout, "sf=%p, pc=%p, unused=%p\n", sf, sf->pc, sf->unused);
    }
}
```

[/proc file structure](#)

The Linux `/proc`, the Solaris `/proc`, and the AIX 6.1 `/proc` file structures are all different — there is no standard. The AIX 6 `/proc` directory structure is somewhat different from the one on Solaris; AIX 6 does not support the Solaris directory structure. (For example, there is no `/proc/self/*` directory on AIX 6.) If you wish to use `/proc`, you will have to go through the `/proc/<pid>` directory with your own pid just like any other process. You can get process usage information from the `/proc/<pid>/status` and `/proc/<pid>/psinfo` files. On AIX 6.1, you need to include the `sys/procfs.h` include file.

On Solaris, an application can use `/proc/pid/ctl` file to turn on/off micro-state accounting. The `/proc` file on AIX 6.1 does not have a means to turn on/off micro-state accounting. Thus, it does not have the `PR_MSACCT` flag or anything similar. The only accounting times `/proc` on AIX 6.1 keeps track of are the `pr_utime`, `pr_stime`, `pr_cutime`, and `pr_cstime` fields in the `/proc/<pid>/status` file. These values are *always* accurate to the seconds and nanoseconds of granularity.

For more information on `/proc`, including details on the directory structure, see the [AIX Version 6.1 Files Reference](#) at: <http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.files/doc/aixfiles/About.htm&tocNode=toc:front/front.cmb/0/1/3/>.

[Memory mapped files](#)

Memory mapping is used to map a file into memory for faster I/O. The call to map a file is `mmap` and the arguments to `mmap` are:

- `mmap(baseAddr, length, protection, flags, fileDescriptor, offSet)`
- `baseAddr` — Desired address to map the file to
- `length` — Number of bytes to map. The OS will round to the nearest page size
- `protection` — Access permissions for mapped region
- `flags` — Attributes of mapped region
- `fileDescriptor` — File descriptor of opened file to map into memory
- `offSet` — Offset from the `baseAddr` to where the file will be mapped

- The `mmap` call is available on Solaris as well but the behavior is slightly different on AIX 6.1. The `flags` argument allows you to specify certain attributes like `MAP_FIXED` or `MAP_VARIABLE`. The `MAP_FIXED` flag specifies that the region being mapped needs to be mapped to the `baseAddr` if possible. Otherwise, you will receive an error. The `MAP_VARIABLE` flag specifies that the OS must attempt to honor the `baseAddr` address. Otherwise, it will return some address that would accommodate the request.
- The first issue is a behavior difference between Solaris and AIX 6.1. If a process maps a file using `MAP_FIXED` at a specified address, such as `0x80000000`, and later attempts to either map a different or the same file to the same specified address, for example, `0x80000000`, Solaris will return without error. AIX 6.1 will fail with:
 - `ENOMEM` — There is not enough address space to map `length` bytes, or the application has not requested X/Open UNIX95 Specification compliant behavior and the `MAP_FIXED` flag was set and part of the address-space range (`baseAddr`, `baseAddr+length`) is already allocated.
 - Obviously, mapping to the same address without unmapping (`munmap`) is a problem that the application must correct.

A second related issue is how memory mapped files are laid out in memory. Consider an application wanting to memory map two files. The first file mapped uses `MAP_FIXED` at address `0x80000000`. The second file mapped uses the address `0x0`. Mapping at `0x0` indicates that the operating system needs to pick the address for you. In this particular case, the second file ended up being mapped directly after the first mapped file. However, the application at a later time attempts to grow the first mapped file region. The application will `mmap` the file and attempts to `mmap` at `0x80000000` with the new larger size. However, `mmap` fails. What is wrong?

The problem is with the second mapped file and where it had been mapped. The second file was mapped using `0x0` so AIX 6.1 decided to map it right after the first file mapped at `0x80000000`. When AIX 6.1 attempted to map the large file, AIX 6.1 determines that the region available at `0x80000000` will not accommodate the new size and fails.

A third and generally unacceptable issue is unmapping files. On AIX 6.1, you can unmap as large a region as you want and turn around and `mmap` the region again without errors. Consider the preceding example of two files being memory mapped contiguously. You can `munmap` the region starting at `0x80000000` all the way to the end of the second mapped region because both regions are contiguous. Not a wise thing to do, but AIX 6.1 permits this type of operation. AIX 6.1 does not keep track of the original size of the mapped region when unmapping.

This unmapping can lead to trouble when the area unmapped is less than the area that was mapped. The `mmap` will succeed as long as the size to map is less than or equal to the area that was previously unmapped. An attempt to map a region with a large size at the same address will fail because a portion of the desired region has not been unmapped.



Java issues

See the *IBM Java Diagnostics guide* (<http://publib.boulder.ibm.com/infocenter/javasdk/v5r0/index.jsp>) for issues that deal with porting Java applications from Solaris to AIX 6.1.

Application performance issues

This section outlines potential performance issues.

AIX environment variables

For better performance of threaded applications, also try the following environment variables:

```
MALLOCMULTIHEAP=1
SPINLOOPTIME=500
YIELDLOOPTIME=500
AIXTHREAD_SLPRATIO=1:1
AIXTHREADSCOPE=S
AIXTHREAD_RWLOCK_DEBUG=OFF
AIXTHREAD_MUTEX_DEBUG=OFF
AIXTHREAD_COND_DEBUG=OFF
```

Certain multithreaded user processes that use the `malloc` subsystem heavily might obtain better performance by exporting the environment variable `MALLOCMULTIHEAP=1` before starting the application. The potential performance improvement is particularly likely for multithreaded C++ programs, because these might make use of the `malloc` subsystem whenever a constructor or destructor is called. Any available performance improvement will be most evident when the multithreaded user process is running on an SMP system, and particularly when system scope threads are used (M:N ratio of 1:1). However, in some cases, improvement might also be evident under other conditions, and on uniprocessors.

- **SPINLOOPTIME** controls the number of times the system will try to get a busy mutex or spin lock without taking a secondary action such as calling the kernel to yield the process. This control is intended for MP systems, where it is hoped that the lock being held by another actively running pthread will be released.
- **YIELDLOOPTIME** controls the number of times the system yields the processor when it tries to acquire a busy mutex or spin lock before actually going to sleep on the lock. The processor is yielded to another kernel thread, assuming there is another runnable thread with sufficient priority. This variable has been shown to be effective in complex applications where multiple locks are in use.
- **AIXTHREAD_SLPRATIO** is the ratio of kernel threads to keep on the side in support of sleeping pthreads. In general, fewer kernel threads are required to support sleeping pthreads, because they are generally woken one at a time.



- **AIXTHREAD_SCOPE=S** means that user threads created with default attributes will be placed into system-wide contention scope. If a user thread is created with system-wide contention scope, it is bound to a kernel thread and it is scheduled by the kernel. The underlying kernel thread is not shared with any other user thread. On AIX 6.1, the default thread model is M:N, which means M user threads are mapped to N Light Weight Processes (LWP) or kernel threads. When using multithreaded applications and especially when running on machines with multiple CPUs, we strongly recommend setting `AIXTHREADSCOPE=S`
- The **pthreads** library maintains a list of active mutexes, condition variables, and read-write locks for use by the debugger. Making mutexes, condition variables, and read-write locks available to the debugger has a certain amount of overhead associated with it. Setting the `AIXTHREAD_RWLOCK_DEBUG`, `AIXTHREAD_MUTEX_DEBUG`, and `AIXTHREAD_COND_DEBUG` to `OFF` will remove the associated overhead and improve performance where mutexes, condition variables, and read-write locks are used extensively.

AIX memory pages

Performance might also be affected by the page size used. Starting with AIX version 5.3 TL4 there are four page sizes supported. In addition to the original 4 KB page size, AIX had already introduced a 16MB page size. The two new page sizes that are also available are 64 KB and 16 GB. To take advantage of either of these new sizes you must be running on POWER6 (POWER5+ and up) hardware and using the 64-bit kernel in AIX 5.3 TL4 and up. There are also additional considerations to be considered when selecting page sizes. See the paper, *Guide to Multiple Page Size Support on AIX 5.3*, ibm.com/servers/aix/whitepapers/multiple_page.pdf for full details.

AIX memory affinity

IBM Power Systems aggregate processors, cache memory and memory management hardware into two or more multichip modules (MCMs). In systems that combine both of these features, the processors in a particular MCM are able to access real memory in their local domain (slightly) more rapidly than real memory at other locations. Applications that are sensitive to memory subsystem performance, therefore, might exhibit improvement if affinity can be established between an application thread and the memory domain that is local to the logical processor on which the thread runs.

Assigning the environment variable `MEMORY_AFFINITY=MCM` to ensures that memory allocations will be preferentially directed to the local memory pool. `MEMORY_AFFINITY= MCM@LRU=EARLY` will cause the kernel to attempt to replace pages from the local pool as soon as the latter has reached its low threshold; without the suffix option, the kernel replaces pages from the local pool only after all pools have reached their low threshold. Finally, `MEMORY_AFFINITY=MCM@SHM=RR` requests that shared memory be allocated in round-robin fashion; other allocation requests remain local.

Some applications might not benefit from the exploitation of memory affinity and placement. This is because interactions between hardware and complex and varied workloads are difficult to characterize. It is recommended that such workloads be benchmarked both with and without affinity facilities engaged to establish the optimal arrangement.

AIX application performance analysis tools

You can take advantage of the following application performance analysis tools:

- **prof:** generates flat profiles on stdout: subprogram entry/exit counts and user-mode execution times; requires code to be instrumented (recompiled and relinked with the `-p` option); part of the AIX `bos.adt.prof` file set.
- **gprof:** generates call-tree profiles on stdout: subprogram entry/exit counts and user-mode execution times, associated with the call tree; requires code to be instrumented (recompiled and relinked with the `-pg` option); part of the AIX `bos.adt.prof` fileset.
- **tprof:** generates a wide variety of performance data in flat AIX files; capable of *microprofiling* (profiling at the source statement level) and nanoprofiling (profiling at the instruction level) and of profiling shared libraries; part of the AIX `bos.perf.tools` fileset.
- **truss:** traces system calls, received signals (see `/usr/include/sys/signal.h` for a list), and machine faults (see `/usr/include/sys/procfs.h` for a list); part of the AIX `bos.sysmgt.serv_aid` fileset.
- **xprofiler:** graphical user interface tool that combines the function of `prof` and `gprof`, as well as the code annotation capability of `tprof` to provide microprofiling and nanoprofiling of source and disassembled object code; part of the AIX `ppe.xprofiler` fileset.
- **Probevue:** AIX command allows developers and administrators to dynamically place probes in user programs as well as kernel code without requiring a special source code or even recompilation.

AIX application optimization tools

You can take advantage of the following application optimization tools:

- **fdpr (FDPR-Pro/AIX):** Feedback-Directed Program Restructuring: two-pass post-link optimization tool: instruments and runs application binaries against a suitably representative workload, then restructures the executable based on the gathered profiling data; part of the AIX `perfagent.tools` fileset.
- **PDF (Profile-directed feedback) facility of the IBM XL compiler suite:** two-pass optimization: executables must be instrumented (recompiled and relinked with the `-qpdf1` option), run with an adequately representative workload, and then rebuilt (recompiled and relinked with the `-qpdf2` option).

Service and support

You can download AIX 6.1 fixes and updates from www-933.ibm.com/support/fixcentral/main/System+p/AIX.



Summary

This paper outlined some of the issues that are involved in migrating applications from Solaris to the AIX 6.1 platform. The authors have touched on differences ranging from compiler options to stack-frame issues. Various scenarios have also been presented to handle the Solaris-to-AIX 6.1 migration simply. For more information regarding AIX 6.1, see the information that is available in the documents listed in the “Resources” section of this paper.

Resources

These Web sites provide useful references to supplement the information contained in this paper:

- IBM System Information Center
<http://publib.boulder.ibm.com/infocenter/systems/index.jsp>
 - IBM System p and AIX Information Center
<http://publib.boulder.ibm.com/infocenter/pseries/index.jsp>
 - AIX Version 6.1 Files Reference
<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.files/doc/aixfiles/About.htm&tocNode=toc:front/front.cmb/0/1/3>
- Search for the following items:
- Performance management and tuning
 - Program address space overview
 - Understanding memory mapping
 - rpcgen command (to review information about TIRPC)
 - /proc
 - directory structure
- AIX 5L online documentation
<http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp>
- Search for the following terms:
- Performance management and tuning
 - Program address space overview
 - Understanding memory mapping
 - rpcgen command (to review information about TIRPC)
 - /proc
 - Directory structure
 - Kernel extensions and device driver support
<http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.doc/aixprggd/kernextc/kernextc.htm>
- IBM XL C/C++ Café. The goal for the Café is to build an XL C/C++ community where customers, partners, and compiler developers can meet to learn, share experiences, and exchange ideas. There are postings providing tips and tricks as well as in-depth information to help users take full advantage of XL C/C++ compiler products on IBM POWER, System z, BlueGene and Cell Broadband Processor™ systems. Blogs and online forums are also available to facilitate discussions on the future direction of the C and C++ programming languages and to gather suggestions and requirements for future releases of XL C/C++ compiler products. We welcome IBM developers using our compilers to participate as well.
ibm.com/rational/cafe/community/ccpp
 - IBM Systems on PartnerWorld
ibm.com/partnerworld/systems



- IBM Publications Center
www.elink.ibm.link.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=US
- IBM Redbooks®
ibm.com/redbooks
 - *AIX 5L Porting Guide (SG246584)*
ibm.com/redbooks/abstracts/sg246034.html?Open
 - *AIX 5L Reference for Sun Solaris Administrators (SG246584)*
ibm.com/redbooks/abstracts/sg246584.html?Open
 - *Developing and Porting C and C++ Applications on AIX (SG245674)*
<http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg245674.html>
 - *Quick Reference: Solaris to AIX (redp0104)*
ibm.com/redbooks/abstracts/redp0104.html
 - *Developing and Porting C and C++ Applications on AIX*
<http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg245674.html>
 - *Quick Reference: Solaris to AIX*
ibm.com/redbooks/abstracts/redp0104.html
- IBM developerWorks®
ibm.com/developerworks
 - *AIX Linking and Loading Mechanisms*
ibm.com/developerworks/eserver/pdfs/aix_ll.pdf
 - *Solving coding problems with temporary variables when porting applications from Solaris to AIX*
ibm.com/developerworks/eserver/articles/code_prob.html
 - *AIX Linking and Loading Mechanisms*
ibm.com/developerworks/eserver/pdfs/aix_ll.pdf
- *IBM XL C/C++ for AIX V10.1 Compiler Reference*
ibm.com/support/docview.wss?uid=swg27012860&aid=1
- Porting roadmaps
ibm.com/servers/enable/site/porting/pseries/porting.html
- XLC/C++ V10.1 for AIX documentation, including a detailed mapping of Solaris C/C++ compiler options to AIX
ibm.com/software/awdtools/xlcpp/multicore/library
- *Guide to Multiple Page Size Support on AIX 5L Version 5.3*
ibm.com/servers/aix/whitepapers/multiple_page.pdf
- *AIX Memory Affinity Support*
ibm.com/servers/aix/whitepapers/memory_affinity.pdf
- AIX 6.1 fixes and updates
www-933.ibm.com/support/fixcentral/main/System+p/AIX



- *Solaris Internals*, by Jim Mauro and Richard McDougall (ISBN 0-13-022496-0)
- Porting your Solaris solution to AIX Version 6.1 solution roadmap
ibm.com/partnerworld/wps/roadmap/aix/v6r1_solaris/port
- *Java Tuning for Performance and IBM POWER6 Support*
ibm.com/systems/p/software/whitepapers/java_tuning.html



About the authors

Anita Govindjee is a consultant in the IBM Solutions Development group. She works on porting software vendors' applications to the IBM Power Systems servers running AIX. Anita has more than 10 years' experience working on UNIX platforms, including AIX, Solaris and HP-UX, as well as doing software development in C, C++ and Java. She holds a BS from the University of Illinois, Urbana-Champaign and an MS degree from Stanford University, both in Computer Science.

Vandana Kumar is a senior AIX consultant at IBM. She works with many software vendors in porting and tuning their applications on AIX. Her specialization is in the AIX kernel's linking and loading mechanisms.

Glen "Tex" Chalemin is a senior AIX consultant at IBM in Austin, Texas. He was part of the original AIX 1.0 development team. He has worked with IBM customers and independent solution providers (ISVs) in porting and running applications on AIX. Currently, he works primarily with Oracle in porting and tuning their applications to AIX.

Nam Keung is a senior programmer for IBM in Austin, Texas. He has worked in the area of AIX ISDN communication, AIX SOM/DSOM development, AIX Multimedia development, NT Clustering technology and Java performance. His current assignment involves helping ISVs in porting, deploying applications, performance tuning, and education for the System p platform. He has been a programmer with IBM since 1987.

Lee Cheng currently is an IBM senior technical staff member for IBM Power Systems and AIX software vendors. She provides support to them in the areas of application benchmarks, performance tuning, application porting and internationalization. Before joining the System p ISV Technical Support group, she was a developer for compilers and the AIX system-management component. She holds an MS in Computer Science from the University of Kentucky.

Brad Cobb is a senior technical consultant in the IBM AIX Collaboration Center, where he assists ISVs to enable their applications on IBM Power Systems servers. Brad has more than 10 years experience with porting applications to AIX and more than six years experience working for IBM. During this time, he has worked with various application types that have allowed him to file five patent applications, publish several articles and speak at developer conferences.

Wayne Huang is a senior consultant for IBM Power Systems and AIX servers with a focus on e-business, banking, finance and securities industries. He provides AIX support to ISVs in the areas of application design, problem determination, system performance tuning and application benchmarks. He holds a BS in Physics from National Taiwan University and an MS in Computer Science from the University of Texas at Austin.

Ruzhu Chen is a software engineer consultant in the IBM ISV Solutions and Enablement organization. He provided supports to ISV and IBM worldwide customers in the area of application benchmarks, porting, performance tuning and optimizations. He joined IBM in HPC Benchmark Center in 2001 to work on benchmarking and performance analysis of scientific and technical applications on IBM System p and System x servers. Prior to work for IBM, he performed two years of postdoctoral research in University of Oklahoma and has 10 years' research experience in the Academy of Sciences of China.



Venkat Dadi works as a technical consultant for IBM STG ISV enablement where he is responsible for helping drive the IBM technological innovation with Oracle ISV. He is client-focussed and assist Oracle product development to enable their solutions on IBM POWER servers. Venkat has more than 10 years experience with porting applications to AIX and he likes to work on improving processes and policies with software vendors. You can reach Venkat at vdadi@us.ibm.com.

Kathleen Dowling is an information developer who has worked on documentation for the IBM XL C/C++ and IBM XL Fortran compilers in the Compilation Technology area for 13 years. She has produced information centers containing user documentation on the features of these compilers and their use on AIX operating systems. She holds a B. of Applied Science in Industrial Systems Engineering from the University of Regina.

Acknowledgements

The authors want to thank the following key contributors and reviewers of this paper:

- **Dwayne Moore**, IBM Rational Compilers Technical Enablement Lead
- **Sean Perry**, IBM Rational C++ front end development
- **Ed Ruddick**
- **Mark Charette**



Trademarks and special notices

© Copyright IBM Corporation 2010. All Rights Reserved.

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.