Solaris SPARC to Solaris x86 Porting Guide
Contents

1 Overview
   1.1 Solaris and HP
   1.2 Why Use Solaris x86 on HP?

2. Migration process
   2.1 Migration prerequisites
   2.2 Migration Issues and Recommendations

3. Custom code migration.
   3.1 Processor Architectural difference and porting issues.
   3.2 C and C++ Compiler differences between Sparc vs x86 Solaris.
   3.3 Using Solaris Developer Tools
   3.4 Linker and Loader Overview
   3.5 Data Migration.
   3.6 Performance benchmarking
   3.7 Programming to Specific Hardware and Configurations

4. Platform infrastructure
   4.1 I/O Architecture
   4.1 Drivers supported
   4.2 H/W supported

5. Database migration.
   5.1 Oracle
   5.2 Sybase
   5.3 Informix

6. Packaged application migration.

7. Services Migration.

8. References
1. Overview

1.1 Solaris and HP

Solaris has been certified to run on ProLiant since 1996. During this time, the joint solution has established a significant install base. HP ProLiant and Sun Solaris engineers have worked together on platform interoperability to satisfy customer needs, resolving issues and enhancing the customer experience.

The HP/Sun Solaris agreement allows HP, for HP ProLiant servers, to distribute 1 year and 3 year, Standard (9x5) and Premium (24X7) Solaris subscriptions which include the entitlement to Solaris 10 and online access to patches, fixes, and updates through Sun Connection for the duration of the subscription.

1.2 Why Use Solaris x86 on HP?

- Solaris is a key operating environment for ProLiant servers and blade servers
- HP offers greater choice of Solaris certified Proliant and StorageWorks platforms
- HP is the only OEM with Solaris and Sun Cluster certified platforms
- HP has SAP on Solaris certified ProLiant platforms
- HP has one of the largest worldwide support organizations in the industry
- HP customers benefit from the confidence, convenience, and choice that comes from a single point of purchase, contact, and accountability for their entire Solaris operating environment
2. Migration process

2.1 Migration prerequisites

Before performing the steps in the migration phase, perform the following:

- Check the Target Solaris x86 Operating System and HP Proliant Server compatibility support at http://h71028.www7.hp.com/enterprise/cache/492635-0-0-0-121.html
- Check for HP Storage support for Solaris x86 Operating System at http://h18006.www1.hp.com/storage/ossolaris.html
- Check for Hardware dependent application to be ported to Solaris x86 at sun site

2.2 Migration Issues and Recommendations

- Device naming schema may get changed in Solaris on x86
  Hardware dependent scripts needs to be modified accordingly

- Check 3 party Tools / Application support on Solaris on x86
  If using 3 party Tools / Application check the support on Solaris on x86.

- JumpStart Software

  An x86 machine can be jumpstarted from SPARC and vice versa. However, the x86 Solaris installation cannot be totally automated.

  Solution
  To jumpstart a Solaris OS x86 client you need one of the following: an already installed Solaris OS for Intel, a Solaris OS CD-ROM (and BIOS capable of booting off CD-ROM), or a Solaris Device Configuration Assistant (DCA) boot floppy. The installation is started by booting off of the appropriate device and media. The DCA is responsible for probing and configuring the devices. During an x86-based installation, the installation program automatically runs the 'kdmconfig' command. kdmconfig determines the type of keyboard, display and mouse attached to the system. kdmconfig can determine this information interactively from the user or can obtain its configuration information from the JumpStart server or from a file.
3. Custom code migration.

3.1 Processor Architectural difference and porting issues.

Endianness

To achieve the goal of multiple-platform, multiple-instruction-set architecture portability, the first dependency issue to address is the endianness (or byte ordering) of the processor. The method of storing information in a computer’s memory is referred to as endianness. When the most significant byte (MSB) of any multi-byte data field is stored at the lowest memory address, which is also the address of the larger field, then the system is said to be big-endian. When the least significant byte (LSB) of any multi-byte data field is stored at the lowest memory address, then the system is called little-endian.

For example, consider an integer 260 (hexadecimal equivalent is 0x0104). Suppose we are storing the value at address 100 and 101.

If the system is big-endian, the system will store the integer as follows.

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>04</td>
</tr>
</tbody>
</table>

If the system is little-endian, the integer will be stored as follows.

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>04</td>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Numeric data is stored in memory in a different format depending upon whether the target implementation platform is based on x86 or UltraSPARC technology. The SPARC processor-based systems implement the MSB first and so are big-endian, whereas the x86 systems implement the LSB first and are called little-endian.

The Solaris Operating System can be used both in big-endian mode (SPARC version) and in little-endian mode (x86 version). Endianness is mandated by the hardware platform used. Solaris OS handles the endianness issue, so that ISVs or developers do not need to be concerned about how quickly their applications will be available on x86 platforms. Sun’s software architecture allows big-endian applications for the Solaris SPARC Platform Edition to run on the little-endian Solaris x86 Platform when they are recompiled. In general, no re-engineering is needed. However, the endianness difference can cause problems, particularly if an application uses low-level code that takes optimal advantage of the hardware capabilities.
Porting Issues based on endianness that might arise on Solaris x86 include the following.

Sending Data over the Network

Applications running on Solaris systems interoperate with other systems over the network using standard network protocols that have enabled communication between big-endian and little-endian systems. However, if the applications transfer data (numeric data) over the network directly without any protocol, then an issue might arise.

Solution

Most of the modern protocols incorporate the External Data Representation (XDR) layer, which byte swaps data if needed during transfers; it is best to take advantage of these protocols. The standard C library also provides the routines ntohl(), ntohs(), htonl() and htons(), which convert numeric data between network byte order and host byte order. These routines are appropriately implemented on the host platform to handle the conversion between host and network byte order and should always be used to shield the application from the underlying architecture.

Accessing the Individual Bytes of the Numeric Data Using Pointers

If an application uses individual bytes of a numeric data type to store and retrieve values, then you might get different results due to the endianness of the underlying platform. In this case the code would not be portable from Solaris SPARC to x86. Value 0X01020304 on x86 (little-endian format)

| 00000100 | 00000011 | 00000010 | 00000001 |

Memory address: 100 101 102 103

```c
int flagVal = 0X01020304;
char *cptr = (char *)&flagVal;
printf("Val is %d\n", *cptr);
```

Solution

In the preceding example, the little-endian machine will print 4, and the big-endian machine will print 1. To make this code portable to multiplatform, use individual character-type variables to store these values rather than using a numeric data type. Alternatively, you may use a structure variable with individual fields for every value, and get and set these values using the structure field names.
Application Data Storage

File systems are neutral to endianness in general, and swapping files between Solaris SPARC and x86 is not an issue. But applications storing raw data that can be shared between platforms would be an issue. For example, if an application on the Solaris SPARC platform writes the data structures in a raw format to the files, then the data stored in these files would be endian-dependent. Reading or writing these data files from an x86 processor-based machine can create problems regarding the endianness of the data. In short, the binary data (raw data) stored on a file is not transferable between the SPARC and x86 platforms.

Solution

Applications storing data that is shared between platforms can handle the endianness issue in one of the following two ways:

Store data in an application-defined, endian-neutral format using text files and strings. Choose either the big-endian or little-endian convention and do byte swapping (potentially using enabling technology such as XDR) when necessary.

The need for cross-platform compatibility is so well-understood that major applications have been available on big-endian and little-endian Solaris environments for years without problems. They range from personal productivity applications to major database management systems from vendors including Oracle, Informix, and Sybase.

Shared Memory Implementation

Sharing memory with opposite-endian devices or processors constitutes data import and export. Access to shared data must take endianness issues into account. To avoid the overhead of the attendant system call, while transferring data to high-speed devices through an operating system, you can arrange to map the memory on a peripheral processor directly into an application's address space. The most common example of shared memory mapped into an application's address space is the case of a graphics adapter with a frame buffer shared between the host and the graphics processor. Some peripheral buses lend themselves to shared memory approaches to interprocessor communication. In general, some means of translating the data must be provided if the processors are not of the same endianness.

Solution

For the resolution of shared memory problems such as memory-mapped graphics adapters, the multi-byte values can be explicitly byte-swapped before storing to, or after loading from, the opposite endian device.
Another solution to the shared memory problem due to the endianness of the processor is the use of apertures for different endianness.

One example is a PCI-based graphics adapter using a little-endian processor internally: the adapter can provide two apertures (ranges of addresses) for its frame buffers. If an adapter accesses the little-endian aperture, the system stores the data as it is presented on the bus and moves it directly into the frame buffer. In contrast, if an adapter accesses the big-endian aperture, the system swaps the data bytes before storing the data. Thus, if the application is running on a little-endian processor, it can simply map the little-endian aperture while the application running on a big-endian processor maps itself to the big-endian aperture. Also a host processor can provide endian translation apertures. UltraSPARC processors handle endianness through allowing independent assignment of endianness to each mapped page. For example, if a file that contains x86 data were mapped into memory in little-endian mode on an UltraSPARC chip, the data would be transparently swapped as it was loaded from or stored to the memory into which the file was mapped. This mechanism solves the problems associated with shared memory if data is properly aligned.

Storage Order Differences

The order of storage of the data varies between platforms; therefore, code written assuming a particular storage order is not portable. The following code sample assumes that the year component may be stored before the month component. This code will work on the x86 machines but will not work properly on UltraSPARC machines.

```c
struct date {
    char year;
    char month;
    char day;
    char dummy; // Added so that the struct var size is 4
} planned_date, end_date;

if ( (*(long *) & end_date ) > (*(long *)&planned_date))
{
    printf("Sorry, You missed the deadline to support Solaris x86 platform \n");
}
```

Solution

Portable code should compare the individual fields of the structure variables separately and not rely upon any sort of storage order as in the following example:

```c
if (compare_dates(end_date, planned_date) > 0)
{
    printf("Sorry, You missed the deadline to support Solaris x86 platform \n");
}
int compare_dates(struct date date_to_compare, 
struct date date_compare_against) 
{
    if (date_to_compare.year != date_compare_against.year)
        return (date_to_compare.year - date_compare_against.year);
    if (date_to_compare.month != date_compare_against.month)
        return (date_to_compare.month - 
        date_compare_against.month);
    if (date_to_compare.day != date_compare_against.day)
        return (date_to_compare.day - date_compare_against.day);
}

Data Alignment Differences

The alignment of the field members in a structure differs across platforms, resulting in variable 
padding requirements and causing the structure to be a different size. In order to make code 
portable, the structure size should always be accessed using the sizeof operator.

Read Write Structures

Most programs read and write data to the permanent storage media such as a complete 
structure in binary form using standard C routines. These routines need the data size that is 
being read or written. Due to the different alignment and data type sizes on different platforms, 
the structure sizes vary, and therefore the sizeof operator should be used to specify the number 
of bytes to read or write in these routines.

Let's say your program needs to read a record of the type MyRecord (that you know has a total 
of 25 bytes of data). In this case, you would not write the following:

    MyRecord myrecord;
    fread(&myrecord, 25, 1, fp);

Instead, you would use the following convention:

    MyRecord myrecord;
    fread(&myrecord, sizeof(MyRecord), 1, fp);

Alignment of Double and Long Double

Differences in alignment of double and long double floating-point variables may cause porting 
problems. The SPARC processor enforces 8-byte alignment on double float variables, while the 
x86 processor enforces only 4-byte alignment.

The following example demonstrates byte alignment problems:

    #include <stdio.h>
    main()
    {
        unsigned char bbb[5] = {0x12, 0x34, 0x56, 0x78, 0x9a };
        printf("%x", *(int *)(bbb));
        printf("%x", *(int *)(bbb + 1));
        exit(0);
    }

This is an example of a poorly written, non-portable C program. This code may be made 
to run on SPARC processors using either the -misalign and -misalign2 compiler options, 
but not on x86.
Padding

As mentioned earlier, the alignment of the field members in a structure differs across platforms, resulting in variable padding requirements and causing the structure to be a different size. The sizeof (BITMAP), structure, shown in the following example, is not always the same on different platforms:

```c
typedef struct bitmap {
    WORD word_type;
    DWORD dword_type;
} BITMAP;
```

Compilers add padding bytes into structures so that all values are appropriately aligned. The amount of padding added is, of course, a function of the structure, the processor's architecture, and the compiler. Because of this, structures may increase in size when code is ported to a processor architecture with more bits (for example, from 16 to 32, or from 32 to 64). For example, if a platform requires that words (assume 2 bytes) must be aligned on 2-byte boundaries, and that dwords (assume 4 bytes) must be aligned on 4-byte boundaries, then the structure will be laid out like so:

- 0 word_type
- 2 (padding)
- 4 dword_type

`sizeof(BITMAP) == 8`

In contrast, on a platform not requiring any special alignment, it would be laid out like so:

- 0 word_type
- 2 dword_type

`sizeof(BITMAP) == 6`

This difference could cause some corruption. However, in most cases, problems are not expected when porting applications from the Solaris SPARC to x86 environments because the Solaris OS compilation system manages most alignment and structure padding.

Floating Point Operation Differences

In x86 systems, the floating-point registers are 80-bits wide, while in the SPARC systems, the floating point registers are 64-bits wide. Due to this, the computation results can differ because intermediate results of arithmetic computations can be in extended precision. Please refer to the Numerical Computation Guide for further detail.

Solution

The -fstore compiler flag minimizes these discrepancies. However, using the -fstore flag introduces a penalty in performance.
Some other issues and solutions include the following.

In x86 systems, each time a single or double precision floating-point number is loaded or stored in the x86 systems, a conversion to or from double extended precision occurs. Loads and stores of floating-point numbers can cause exceptions. In SPARC systems, a single or double precision floating-point number is always loaded or stored as single or double precision.

Solution: The -fprecision initializes the rounding-precision mode bits in the floating-point control word to single (24 bits), double (53 bits), or extended (64 bits), respectively. The default floating-point rounding-precision mode is extended. Note that on x86, only the precision, not exponent, range is affected by the setting of floating-point rounding precision mode.

In the x86 systems, gradual underflow is implemented entirely in hardware. As opposed to SPARC systems, x86 systems lack a nonstandard mode.

Solution: Use the option -fnonstd on x86 systems. This causes nonstandard initialization of floating-point arithmetic hardware. In addition, the -fnonstd option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions.

The fpversion utility is not provided in x86 systems. In SPARC systems, the fpversion utility distributed with the compilers identifies the installed CPU and estimates the processor and system bus clock speeds. fpversion determines the CPU and FPU types by interpreting the identification information stored by the CPU and FPU. It estimates their clock speeds by timing a loop that executes simple instructions that run in a predictable amount of time. The loop is executed many times to increase the accuracy of the timing measurements. For this reason, fpversion is not instantaneous; it can take several seconds to run. fpversion also reports the best -xtarget code generation option to use for the host system.

32-Bit and 64-Bit Issues

Enterprise-wide business critical applications that benefit from 64-bit technology include data warehousing, high-resolution graphics, computer-aided design, simulation and scientific data analysis, and multimedia web servers. These applications require more precision, a very large amount of simultaneous data processing capability, and a larger memory space than the 32-bit equivalent can address.

A 64-bit system differs from a 32-bit system in the following ways:

- Extended precision
- Large dataset support
- Large virtual address space

The Solaris OS, SPARC Platform Edition, has support for both 32-bit and 64-bit technology, while the x86 Platform Edition only supports the 32-bit family of x86 processors. Both Sun cc and gcc default to 32-bit mode on 64-bit machines, unless compiled with -xarch=v9 (64-bit SPARC), v9a (UltraSPARC extensions), or v9b (UltraSPARC III extensions). You can develop and run 32-bit applications in a 64-bit environment on the SPARC platform if you avoid using the architecture-dependent flag -xarch=v9. You can port the same application to the x86 platform after recompiling it on x86 platform without making any code changes.
Solution
It is also possible to generate two binaries (one 32-bit and one 64-bit) from a single source file on the SPARC platform. The Solaris compilers have compile-time flags that can create a resulting object file that is 32-bit or 64-bit. 64-bit specific code in the header files and source files are defined using #ifdef as appropriate to support the single source file for both 32-bit and 64-bit.
3.2 C and C++ Compiler difference between Sparc vs x86 Solaris.

ISVs and developers can choose between multiple compilers to use on Solaris Operating System, x86 Platform Edition. If your application is developed primarily on the SPARC Platform Edition using Sun Studio developer tools, it is easier to port to Solaris x86. If the application has been optimized for the SPARC Platform Edition using back-end compiler flags, some flags may be incompatible for the x86 platform. These flags need to be deleted or replaced with the flags that are valid on the x86 platform and used for optimization for the x86 platform.

Sun Studio Developer Tools

Sun Studio software consists of C, C++, Fortran compilers; the Performance Analyzer; and the debugger. The complete compilation architecture consists of a compiler, an assembler, and a link editor. The compiler is made up of a front-end component and a back-end component. The front end consists of syntax and semantic analyzer modules based on the source language, while the back end consists of code optimization and code generation modules based on the instruction set for a particular hardware.

Sun Studio developer tools are available for the Solaris OS, SPARC and x86 Platform Editions. Since the compilers share a common front end, most source code is easily ported between platforms. Applications with platform-specific definitions (for example, #ifdef __sparc), may require user intervention before porting between platforms.

Compiler Flags

Sun Studio 12 Compiler Collection consists of C, C++, and Fortran compilers. The compiler consists of a front-end component and a back-end component. The front-end consists of syntax and semantic analyzer modules based on the source language, while the back end consists of code optimization and code generation modules based on the instruction set for a particular hardware. The Sun Studio 12 Compiler Collection works on both the Solaris OS SPARC and x86 platforms; the front-end component works the same for both, so the source code can be ported without changes. However, when the intermediate code generated by the front-end has to be optimized, the backend compiler flags are used. These are architecture dependent and are different for SPARC and x86 platforms.

Compilers to Consider

Sun Studio 12 Compiler Collection contains different options for back-end compiler flags on a wide variety of SPARC processors as well as x86 architecture-based processors. Also, it is useful when the code is ported from one architecture to another, as there are equivalent options available in this compiler. Hence, it is recommended to use Sun Studio 8 Compiler Collection for a
successful porting of source code from the SPARC platform to the x86 platform. However, you can consider using the GCC compiler as well.

Compiler Options for SPARC Platform Not Used on x86 Platform

The following options are used in the SPARC version of Sun Studio software; they are not present in the x86 version. When a source program is compiled using these options on the Solaris OS, x86 Platform Edition, A warning is issued indicating an illegal option has been used, or else the compiler ignores the option.

Table 1: Options for Sun Studio (SPARC Version)

<table>
<thead>
<tr>
<th>Compiler Option</th>
<th>Explanation</th>
<th>Behavior on x86 Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cg89 (C++)</td>
<td>This option is the same as -xcg89 in cc. Used for improving runtime performance for the SPARC architecture. It compiles the C++ code for generic SPARC architecture. Equivalent to -xtarget=ss2.</td>
<td>Warning issued by C++ compilers [1]</td>
</tr>
<tr>
<td>-cg92 (C++)</td>
<td>This option is the same as -xcg92 in cc. Used for improving runtime performance for the SPARC architecture. It compiles the C++ code for SPARC V8 architecture. Equivalent to -xtarget=ss1000.</td>
<td>Warning issued by C++ compiler [1]</td>
</tr>
<tr>
<td>-dalign (C)</td>
<td>Equivalent to -xmemalign=8s. This flag allows the user to specify the maximum memory alignment of 8 bytes to be assumed by the compiler in indeterminable situations. It also specifies that SIGBUS be raised when misaligned memory access takes place.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-fns</td>
<td>Turns on the SPARC nonstandard floating-point mode. When nonstandard mode is enabled, floating-point arithmetic may produce results that do not conform to the requirements of the IEEE 754 standard.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-misalign</td>
<td>Equivalent to -xmemalign=1i. This flag allows the user to specify the maximum memory alignment of 1 byte to be assumed by the compiler in indeterminable situations. This option specifies that access be interpreted and execution be continued, when a misaligned memory access takes place.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-misalign2 (C)</td>
<td>Equivalent to -xmemalign=2i. This flag allows the user to specify the maximum memory alignment of 2 bytes to be assumed by the compiler in indeterminable situations. This option specifies that SIGBUS be raised when a misaligned access occurs.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xalias_level</td>
<td>Used to determine the assumptions that can be made by the compiler in order to perform optimizations (using the -xO option) using type-based alias-analysis.</td>
<td>Warning issued by C compiler [2]</td>
</tr>
<tr>
<td>-xautopar (C)</td>
<td>This option turns on automatic parallelization for multiple processors used in the SPARC architecture. It also does dependence analysis (analyzes loops for inter-iteration data dependence) and loop restructuring.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xcache (C++)</td>
<td>This option defines cache properties for use by the optimizer.</td>
<td>Warning issued by C++ compiler[1]</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>-xcg89 (C)</td>
<td>This option is used for improving runtime performance for the SPARC architecture. It compiles the C++ code for generic SPARC architecture. Equivalent to <code>-xtarget=ss2 (-xarch=v7 -xchip=old -xcache=64/32/1)</code></td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xcg92 (C)</td>
<td>This option is used for improving runtime performance for the SPARC architecture. It compiles the C++ code for SPARC V8 architecture. Equivalent to <code>-xtarget=ss1000 (-xarch=v8 -xchip=super -xcache=16/32/4:1024/32/1)</code></td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xcheck</td>
<td>This option adds a runtime check for stack overflow.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xcode</td>
<td>This option specifies the code address space.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xcrossfile (C)</td>
<td>This option enables optimization and inlining across source files.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xdepend (C)</td>
<td>This option analyzes loops for inter-iteration data dependencies and does loop restructuring.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xexplicitpar (C)</td>
<td>This option generates paralleled code based on specification of #pragma MP directives.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xhwcprof (C)</td>
<td>This option enables compiler support for hardware counter-based profiling.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xia (C++)</td>
<td>This option links the appropriate interval arithmetic libraries and sets a suitable floating-point environment.</td>
<td>Warning issued by C++ compiler[1]</td>
</tr>
<tr>
<td>-xipo</td>
<td>This option performs inter-procedural optimizations.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xjobs</td>
<td>This option specifies the maximum number of components the compiler will fork in parallel.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xldscope</td>
<td>This option is used to change the default linker scoping for the definition of extern symbols.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xlic_lib=sunperf</td>
<td>This option links in the Sun-supplied performance libraries.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xlinkopt (C)</td>
<td>This option instructs the compiler to perform link-time optimization on the resulting executable or dynamic library over and above any optimizations in the object files.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xloopinfo (C)</td>
<td>This option shows which loops are paralleled and which are not.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xmemalign</td>
<td>This option specifies maximum assumed memory alignment and behavior of misaligned data accesses.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xMerge</td>
<td>This option merges data segments into text segments.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xopenmp</td>
<td>This option enables explicit parallelization with OpenMP directives.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Warning or Ignored note</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>-xpagesize</td>
<td>This option sets the preferred page size for the stack and the heap.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xpagesize_heap</td>
<td>This option sets the preferred page size for the heap.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xpagesize_stack</td>
<td>This option sets the preferred page size for the stack.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xport64 (C++)</td>
<td>This option helps to debug code being ported to a 64-bit environment.</td>
<td>Warning issued by C++ compiler[1]</td>
</tr>
<tr>
<td>-xparallel (C)</td>
<td>This option parallelizes loops both automatically by the compiler and explicitly specified by the programmer.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xprefetch</td>
<td>This option enables prefetch instructions on those architectures that support prefetch.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xprefetch_level</td>
<td>This option controls the number of pre-fetch instructions as determined with -xprefetch=auto.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xprofile_ircache</td>
<td>This option is used with -xprofile=collect, or -xprofile=use to improve compilation time during the use phase by reusing compilation data saved from the collect phase.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xprofile_pathmap</td>
<td>This option is used with -xprofile=use and it uses the prefix of the UNIX pathname of a directory tree in which object files were compiled.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xreduction (C)</td>
<td>This option turns on reduction recognition during automatic parallelization.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xregs</td>
<td>This option specifies the usage of registers for the generated code.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xrestrict (C)</td>
<td>This option treats pointer-valued function parameters as restricted pointers.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xsafe=mem</td>
<td>This option allows the compiler to assume no memory-based traps occur.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xsafe=mem</td>
<td>This option passes the instruction to the compiler to not do optimizations or parallelization of loops that increase code size.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xthreadvar</td>
<td>This option controls the implementation of thread local variables.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xvector</td>
<td>This option enables automatic generation of calls to the vector library functions.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
<tr>
<td>-xvis</td>
<td>This option is used when the assembly-language templates defined in the VIS instruction-set Software Developers Kit (VSDK) are used.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-xvpara (C)</td>
<td>This option warns about loops that have #pragma MP directives specified when the loop may not be properly specified for parallelization.</td>
<td>Option ignored [3]</td>
</tr>
<tr>
<td>-Zll (C)</td>
<td>This option creates the program database for lock_lint, but does</td>
<td>Option</td>
</tr>
</tbody>
</table>
not generate executable code. | ignored [3]

Key: (C) denotes compiler option used by cc (C compiler). (C++) denotes compiler option used by CC (C++ compiler). Otherwise the option is valid for both cc and CC (C and C++ compilers).

Notes:
[1] The C++ compiler option works only for the SPARC platform and would give a compiler warning when used on the x86 platform. To avoid this warning, this compiler option should not be used on the x86 platform.
[2] The C compiler option works only for the SPARC platform and would give a compiler warning when used on the x86 platform. To avoid this warning, this compiler option should not be used on the x86 platform.
[3] The option is valid only on the SPARC platform. While its use on the x86 platform does not generate any warning, this option is ignored during the generation of binaries on the x86 platform.
Compiler Options for x86 Platforms Not Used on SPARC Platform

Here we explain the options that are used on the x86 version of Sun Studio software and are not present in the SPARC version. When a source program is compiled using these options on the Solaris OS, SPARC Platform Edition, a warning is issued indicating that an illegal option was used, or the compiler ignores the option.

Table 2: Options for Sun Studio (x86 Version)

<table>
<thead>
<tr>
<th>Compiler Option</th>
<th>Explanation</th>
<th>Behavior on SPARC Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>-386 (C++)</td>
<td>Same as -xtarget=386. Specifies the target platform as 386-based instruction set and optimization.</td>
<td>Warning issued by C++ compiler[1]</td>
</tr>
<tr>
<td>-486 (C++)</td>
<td>Same as -xtarget=486. Specifies the target platform as 486-based instruction set and optimization.</td>
<td>Warning issued by C++ compiler[1]</td>
</tr>
<tr>
<td>-fprecision</td>
<td>This option initializes the rounding-precision mode bits in the floating-point control word to single (24 bits), double (53 bits), or extended (64 bits).</td>
<td>Option ignored[3]</td>
</tr>
<tr>
<td>-fstore</td>
<td>This option causes the compiler to convert the value of a floating-point expression or function to the type on the left side of an assignment rather than leave the value in a register when: The expression or function is assigned to a variable. The expression is cast to a shorter floating-point type.</td>
<td>Option ignored[3]</td>
</tr>
<tr>
<td>-nofstore</td>
<td>This option disables forced precision of an expression.</td>
<td>Option ignored[3]</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>-x386 (C)</td>
<td>This option optimizes the code for the 80386 processor.</td>
<td>Option ignored[3]</td>
</tr>
<tr>
<td>-x486 (C)</td>
<td>This option optimizes the code for the 80386 processor.</td>
<td>Option ignored[3]</td>
</tr>
<tr>
<td>-xpentium</td>
<td>This option optimizes the code for the Pentium processor.</td>
<td>Warning issued by C compiler[2]</td>
</tr>
</tbody>
</table>

Key: (C) denotes compiler option used by cc (C compiler). (C++) denotes compiler option used by CC (C++ compiler). Otherwise the option is valid for both cc and CC (C and C++ compilers).

Notes:
[1] The C++ compiler option works only for the x86 platform and would give a compiler warning when used on the SPARC platform. To avoid this warning, this compiler option should not be used on the SPARC platform.
[2] The C compiler option works only for the x86 platform and would give a compiler warning when used on the SPARC platform. To avoid this warning, this compiler option should not be used on the SPARC platform.
[3] The option is valid only on the x86 platform. While its use on the SPARC platform does not generate any warning, this option is ignored during generation of binaries on the SPARC platform.
Compiler Options Used on Both Solaris Versions but with Different Implications

The following options in the Sun Studio developer tools are used on both SPARC and x86 versions, but with different implications that are explained in the table. When a source program is compiled on the Solaris OS using these options, the values passed to the options are different on both platforms.

Table 3: Options for Sun Studio

<table>
<thead>
<tr>
<th>Compiler Option</th>
<th>Explanation</th>
<th>Values for SPARC</th>
<th>Values for x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Aname[(tokens)] (C)</td>
<td>This option associates name with the specified tokens like a #define preprocessing directive.</td>
<td>The values for preassertions are: machine(sparc) cpu(sparc)</td>
<td>The values for preassertions are: machine(i386) cpu(i386)</td>
</tr>
<tr>
<td>-Dname[tokens]</td>
<td>This option associates name with the specified tokens like a #define preprocessing directive.</td>
<td>The values are: sparc (not valid in -Xc mode) __sparc (valid in all modes)</td>
<td>The values are: i386 (not valid in -Xc mode) __i386 (valid in all modes)</td>
</tr>
<tr>
<td>-fast</td>
<td>This option selects a set of baseline options for optimizing benchmark applications.</td>
<td>The option expands to the following values on SPARC platform: -fns -fsimple=2 -fsingle -ftrap=%none -xalias_level=basic -xarch -xbuiltin=%all -xdepend -xlibmil -xmemalign=8s -xO5 -xprefetch=</td>
<td>The option expands to the following values on x86 platform: -fns -fsimple=2 -fsingle -ftrap=%none -nofstore -xarch -xbuiltin=%all -xdepend -xlibmil -xO5</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Equivalent</td>
<td>Note</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>-fnonstd</td>
<td>This option causes nonstandard initialization of floating-point arithmetic hardware.</td>
<td>This option expands to: fns -ftrap=common</td>
<td>This option expands to: ftrap=common</td>
</tr>
<tr>
<td>-KPIC</td>
<td>This option is used to compile source files when building a shared library.</td>
<td>Equivalent to: -xcode=pic32</td>
<td>Same as -Kpic</td>
</tr>
<tr>
<td>-Kpic</td>
<td>This option is used to compile source files when building a shared library.</td>
<td>Equivalent to: -xcode=pic13</td>
<td>Compiles with position-independent code.</td>
</tr>
<tr>
<td>-PIC (C++)</td>
<td>This option is used to compile source files when building a shared library.</td>
<td>Equivalent to: -xcode=pic32</td>
<td>Same as -Kpic</td>
</tr>
<tr>
<td>-pic (C++)</td>
<td>This option is used to compile source files when building a shared library.</td>
<td>Equivalent to: -xcode=pic13</td>
<td>Same as -Kpic</td>
</tr>
<tr>
<td>-xarch</td>
<td>This option specifies the instruction set architecture (ISA).</td>
<td>The values are: generic generic64 native native64 v7 v8a v8 v8plus v8plusa v8plusb v9a v9b v9</td>
<td>The values are: generic 386 pentium_pro</td>
</tr>
<tr>
<td>-xchip</td>
<td>This option specifies the target processor for use by the optimizer.</td>
<td>The values are: generic old super super2 micro micro2 hyper hyper2 powerup ultra</td>
<td>The values are: generic 386 486 pentium pentium_pro</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Old Options</td>
<td>New Options</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>-xO</td>
<td>This option optimizes the object code depending on the levels set.</td>
<td>ultra2 ultra2e ultra2i ultra3 ultra3cu</td>
<td>xO1: Does basic local optimization. xO2: Does basic local and global optimization. xO3: Performs like -xO2, but also optimizes references or definitions for external variables. xO4: Performs like -xO3, but also automatically inlines functions contained in the same file for faster execution. xO5: Generates the highest level of optimization.</td>
</tr>
<tr>
<td>-xtarget</td>
<td>This option specifies the target system for instruction set and optimization.</td>
<td>native native64 generic generic64 &lt;&lt;platform-name&gt;&gt;</td>
<td>native generic 386 486 pentium pentium_pro</td>
</tr>
</tbody>
</table>

Key: (C) denotes compiler option used by cc (C compiler). (C++) denotes compiler option used by CC (C++ compiler). Otherwise the option is valid for both cc and CC (C and C++ compilers).
3.3 Using Solaris Developer Tools

To build software on the Solaris OS, you must install the tools you need on your development machine. Whether you want to use the standard tools that are bundled in Solaris OS, or use the add-on tools Sun Studio IDE and NetBeans IDE, you must first install the appropriate Solaris software for a developer environment.

When you install the Solaris OS, you must select one of the following software groups, which contain the developer packages:

- Developer
- Entire
- Entire Plus OEM

With these software groups, you get compilers such as the GNUC compiler (gcc) and the Java compiler (javac). Also installed are the GNU source-level debugger (gdb) and the Modular Debugger (mdb), a linker (ld), source control utilities (sccs), and build utilities such as make. The files are installed in /usr/css and /usr/sfw.

Kernel-Level Debugging

The Modular Debugger mdb is an extensible, general purpose debugging tool for the Solaris OS. The Solaris Modular Debugger mdb(1) command to debug complex software systems.

For more information, refer to Solaris Modular Debugger Guide at www.sun.com

Using the Sun StudioTools

The Sun™ Studio software provides modules for creating, editing, building, debugging, and analyzing the performance of a C, C++, or Fortran application. Many Sun Studio tools have both a GUI and command-line equivalent. Those tools with GUIs provide online help. For the command-line versions, use the associated man pages.

The Sun Studio software includes the following tools:

- Sun Studio IDE – An integrated development environment that provides access to the Sun Studio tools. In addition to C, C++, and Fortran tools, the IDE includes a set of basic Java language support modules.
- Sun Studio C compiler – Includes a C compiler, incremental link editor, and lint program.
• Sun Studio C++ compiler – Includes a full-featured C++ compiler and interval arithmetic library.
• Sun Studio Fortran compiler – Includes a full-featured environment and libraries for both f95 and f77.
• dbx Debugger – An interactive, source-level, command-line debugging tool.
• Sun Studio dmake make tool – A command-line tool for building targets in distributed, parallel, or serial mode.
• Math libraries – A floating-point environment that is supported by software and hardware on SPARC® and x86 platforms that run the Solaris Operating System.
• OpenMP API – Directives and pragmas for converting legacy Sun parallelization directives to OpenMP, which is a portable, parallel programming model for shared memory multiprocessor architectures.
• Performance Analyzer – A GUI and command-line tool for collecting and analyzing performance data.
• Thread Analyzer – A GUI and command-line tool for analyzing the execution of multithreaded programs and checking for a variety of multithreaded programming errors.
• Sun Performance Library – A library of Sun-specific extensions and features for using optimized, high-speed mathematical subroutines for solving linear algebra and other numerically intensive problems.

Using the NetBeans IDE
NetBeans IDE provides tools to help you build cross-platform applications for the Solaris OS and any other operating platform. The IDE contains the following:
• Swing GUI Builder (formerly ProjectMatisse), a layout manager for designing user interfaces that use Swing and AWT components. Developers can design user interfaces by dragging and dropping GUI components from a palette onto a canvas.
• Web application support with a bundled Tomcat server or with a wide variety of Java EE Application Servers.
• Full Java EE 5 support, and easy creation of CRUD applications that are based on JavaServer™ Faces and the Java Persistence API.
• Support for building plug-in modules for NetBeans IDE or building your stand-alone project on top of the NetBeans platform.
3.4 Linker and Loader Overview

Link-Editor
The link-editor, ld(1), concatenates and interprets data from one or more input files. These files can be relocatable objects, shared objects, or archive libraries. From these input files, one output file is created. This file is either a relocatable object, an executable application, or a shared object. The link-editor is most commonly invoked as part of the compilation environment.

Runtime Linker
The runtime linker, ld.so.1(1), processes dynamic executables and shared objects at runtime, binding the executable and shared objects together to create a runnable process.

Shared Objects
Shared objects are one form of output from the link-edit phase. Shared objects are sometimes referred to as Shared Libraries. Shared objects are importance in creating a powerful, flexible runtime environment.

Object Files
The Solaris OS link editors work with files that conform to the executable and linking format, otherwise referred to as ELF.

Link-editing takes a variety of input files, typically generated from compilers, assemblers, or ld(1). The link-editor concatenates and interprets the data within these input files to form a single output file. Although the link-editor provides numerous options, the output file that is produced is one of four basic types:

Relocatable object
A concatenation of input relocatable objects that can be used in subsequent link-edit phases.

Static executable
A concatenation of input relocatable objects that have all symbolic references resolved. This executable represents a ready-to-run process.

Dynamic executable
A concatenation of input relocatable objects that requires intervention by the runtime linker to produce a runnable process. A dynamic executable might still need symbolic
Shared object

A concatenation of input relocatable objects that provide services that might be bound to a dynamic executable at runtime. The shared object can have dependencies on other shared objects. These output files, and the key link-editor options used in their creation, are shown in Figure 1–1.

Dynamic executables and shared objects are often referred to jointly as dynamic objects.

32–Bit Environments and 64–Bit Environments

The link-editors are provided as 32–bit applications and 64–bit applications. Each link-editor can operate on 32–bit objects and 64–bit objects. However, a link-edit can not contain a mix of 32–bit objects and 64–bit objects. On systems that are running a 64–bit environment, both versions of the link-editor can be executed. On systems that are running a 32–bit environment, only the 32–bit version of the link-editor can be executed. Although a 32–bit link-editor can generate a 64–bit object, the size of the generated object, not including the .bss, is restricted to 2 Gbytes.
No command-line option is required to distinguish a 32–bit link-edit or 64–bit link-edit. The link-editor uses the ELF class of the first relocatable object on the command-line to govern the mode in which to operate. Specialized link-edits, such as linking solely from a mapfile or an archive library, are uninfluenced by the command-line object. These link-edits default to a 32–bit mode. A 64–bit link-edit can be enforced by using the link-editor -64 option.

Support Tools

The Solaris OS also provides several support tools and libraries. These tools provide for the analysis and inspection of these objects and the linking processes. These tools include elfdump(1), lari(1), nm(1), dump(1), ldd(1), pvs(1), elf(3ELF), and a linker debugging support library.

3.5 Data Migration

Data Migration:
Data migration is the process of importing legacy data to a new system. This can involve entering the data manually, moving disk files from one folder (or computer) to another, database insert queries, developing custom software, or other methods. The specific method used for any particular system depends entirely on the systems involved and the nature and state of the data being migrated.

There are 4 steps invloved for Data migration from Solaris/SPARC to Solaris/x86:
- Backup the data on Stoarge.
- Configure storage on x86 servers
- Creating target disk layout
- Migrate data to target machine

Backup the data on Stoarge:

To backup the raw data Solaris has tar, pax, cpio, volcopy, gzip etc commands.

Tar: it was originally created for backups on magnetic tape, but it can be used to create tar files anywhere on a filesystem.

Tar is file-based and essentially should be considered to be an outdated zero-compression archiver.

Pax:
pax is an attempt to merge functionality previously provided by both cpio and tar

cpio:
It is also old file based zero-compression archiver. Has options to perform byte-swapping, write a number of different archive formats, and pipe the data to other programs. cpio does not know how to walk the directory tree and a list of files must be provided through stdin

volcopy:
a rarely used utility to make an image or literal copy of a partition to another partition. volcopy makes a literal copy of the ufs filesystem using a blocksize matched to the device to disk or tape.

gzip:
The standard GNU clone of zip compression utility, replacing older and weaker compress.
Configure storage on Solaris x86 servers:

It is necessary to configure the storage on Solaris/x86, while migrating from Solaris/SPARC. It should be configured as it was on source machine.

Creating target disk layout:

Solaris volume manager would be useful for creating disk layout on target machine.
http://docs.sun.com/app/docs/doc/816-4520

Migrate data to target machine:

Main hurdle for data migration from Solaris/SPARC to Solaris/x86 is Endianness issue. It can hit when you try to manipulate multi-byte data. It could be raw files which are used by applications or individual data types using pointers. As discussed, to overcome such issues, do byte swapping.
3.6 Performance benchmarking

Sun has not published any major benchmarks on SPARC during 2009. However, SUN has published a number of single system benchmark results in 2009, on the x86 platform. These systems have Intel Xeon X5570 processors and Solaris 10 or OpenSolaris Operating system. Following benchmarks have been published:

- TPC-H benchmark at 1000 GB scale factor for all non-clustered systems
- Unicode result on the two-tier SAP SD Standard Application Benchmark
- SPECweb2005 benchmark
- SPEC CPU2006
- Thomson Reuters Market Data System benchmark
- SPECjbb2005
- SPEC OMP benchmark
- SPECjvm2008
3.7 Programming to Specific Hardware and Configurations

Known Solaris Compatible Hardware
The Hardware Compatibility List (HCL) identifies hardware that is compatible with the Solaris OS. Separate lists identify systems and components that are compatible with the latest commercial versions of the Solaris OS and with the latest development versions. The HCL is available in a comma-separated value (CSV) format, for searching and sorting the list in a spreadsheet or database.

Testing x86 Hardware for Solaris Compatibility
The Hardware Certification Test Suite (HCTS) includes system certification tests, controller certification tests, a command-line interface, and a browser-based user interface. HCTS tests certify systems, motherboards, and various network, storage, and serial I/O controllers to work with the Solaris Operating System. HCTS collects log files, determines test pass or fail results, and generates a final test report. Hardware that passes HCTS certification testing is eligible to be included in the Hardware Compatibility List as Certified.

Developing in the x86 Assembly Language
The Solaris OS provides an assembly language for the x86 platforms. The assembler that generates code for the 32-bit x86 processor architecture and translates source files that are in assembly language format into object files in linking format.

For more information, refer to Assembly Language Reference Manual

Developing 64–bit Applications for the Solaris OS
The Solaris OS provides a 64–bit computing environment along with backward compatibility for 32–bit applications.

For more information, refer to Solaris 64-bit Developer’s Guide

The Solaris 64-bit Developer’s Guide is written primarily for the application developer. The book provides guidance for choosing whether to use the 32–bit Solaris application development environment or the 64–bit environment. The manual explains the similarities and differences between the two environments, and explains how to write code that is portable between the two environments. This book also describes some of the tools provided by the operating system for developing 64-bit applications.
4. Platform infrastructure
   4.1 I/O Architecture
   Unlike the x86 family, which uses special IN/OUT instructions to access the PCI I/O space, the
   SPARC chip family treats access to the PCI I/O space the same as it treats access to the memory
   space. Communication with I/O devices in the SPARC platform is accomplished through memory;
   a range of memory locations is logically replaced by device registers.
   The x86 platform accesses I/O ports through the I/O address space by means of a set of I/O
   instructions and a special I/O protection mechanism. Accessing I/O ports through memory-
   mapped I/O is handled with the processor’s general-purpose move and string instructions, with
   protection provided through segmentation or paging. I/O ports can be mapped so that they
   appear in the I/O address space or the physical-memory address space (memory-mapped I/O) or
   both.
   The SPARC platform assumes that input/output registers are accessed by means of load/store
   alternate instructions, normal load/store instructions, coprocessor instructions, or read/write
   ancillary state register instructions (RDASR, WRASR). In the load/store alternate instructions
   case, the I/O registers can only be accessed by the supervisor. If normal load/store instructions,
   coprocessor instructions, or read/write Ancillary State Register instructions are used, then
   whether the I/O registers can be accessed outside of supervisor code or not depends on the
   implementation.

4.2 Drivers supported
4.3 H/W supported
5. Database migration.

5.1 Oracle

Database backup: (Oracle databases)
Oracle provides several methods which one can use to perform logical database backup and recovery. Of these, the common utilities used are the import and export tools. Many times they are used to move schema definitions and data between different Oracle instances and users. The export/import utilities are commonly used to perform the following tasks:

- Backup and recovery (small databases only, say < +50GB, if bigger, use RMAN instead)
- Move data between Oracle databases on different platforms (for example from Sparc to x86)
- Reorganization of data/ eliminate database fragmentation (export, drop and re-import tables)
- Upgrade databases from extremely old versions of Oracle (when in-place upgrades are not supported by the Database Upgrade Assistant anymore)
- Detect database corruption. Ensure that all the data can be read

* Transporting tablespaces between databases

For copying database back to target machine,
- The schema(s) should be exported using the exp utility using the ROWS=N parameter. During this stage all the objects which are non data, such as PL/SQL, Sequences, Views, Table definitions, Object privileges etc. will be exported. There will be one such exp file per each schema exported.
- The data from the large tables (fact tables and the large dimensions) should be exported
- The destination database should be setup with all the same global roles, users, user privileges, system triggers (on_logon) and tablespaces (i.e. all the settings which are not exported using exp). If there is a need for a different tablespace/storage configuration then this should be taken into account in the latter stages.
- The schema(s) should be imported using imp (schemas and small tables)

run the imp/exp commands is following manner
exp scott/tiger file=emp.dmp log=emp.log tables=emp rows=yes indexes=no
exp scott/tiger file=emp.dmp tables=(emp.dept)

imp scott/tiger file=emp.dmp full=yes
imp scott/tiger file=emp.dmp fromuser=scott touser=scott tables=dept
6. Packaged application migration

Needs to be updated.
7. Services Migration.

**Migrating Apache Web Server on Solaris OS From SPARC Platforms to x86 Platforms**

The Apache Web Server 2.0.55 source was downloaded in a compressed tar file from the Apache web site. After extracting the downloaded tar file in a tmp directory, a series of steps were followed for compiling and installing the build on the machines running on SPARC and x86 technology. The study used the C compiler found in the Sun Studio 12, Compiler Collection developer tools, as this compiler is supported on both the SPARC and x86 platform editions of the Solaris OS. This compiler contains different option flags for optimizing the code for both platform editions, and you can use it to port source code from one platform to another.

First the code was built on the SPARC platform, by following these steps:

- **configure**
  This is a shell script generated by the GNU autoconf utility to configure values for system-dependent variables and create Makefiles. It was used for configuring the Apache source tree, as follows:
  ```
  CFLAGS=" -xparallel -fast -xtarget=ultra -xarch=v8plus" \n  LDFLAGS=" -xparallel -fast -xarch=v8plus" \n  ./configure --prefix=/home/testuser/apache/apache2
  ```
  Here CFLAGS represents options for the C compiler, LDFLAGS are options to be passed to the linker, and --prefix specifies the directory in which to install the Apache files. If not specified, the default is /usr/local/apache2.

The compiler flags were put in explicitly using the environment variables CFLAGS and LDFLAGS. These flags were used for back-end optimization of code on the SPARC platform:

- `-xparallel` was used to parallelize loops for faster performance.
- `-fast` is a specially optimized macro option that closely binds the intermediate code generated by the compiler front end to take advantage of the underlying hardware in order to enhance the performance of the application.
- `-xtarget=ultra` was used to specify the exact target system for instruction set and optimization. It overrides some of the values specified by the option `-fast`.
- `-xarch=v8plus` was used to override the `-xarch` option used in the `-xtarget` macro, because this value is more suitable for enabling the compiler to generate code for good performance on the SPARC architecture.
The configure utility checked for the path of the build tools that were to be used during the build, and it also verified that the tools worked in the path specified in the PATH environment variable. The utility configured the APR (Apache Portable Runtime) library used for implementing Apache-specific calls in the native implementation of the OS platform.

configure checked for various executables like gcc and cc, along with the basic options make, awk, ld, sed, and so on. It verified the presence of different header files that would be used during the compilation of the source code. These included header files for libraries implementing threads, shared memory, processes, locking, and networking support (including IPv6). It also read all environment variables that had been set. Finally, the utility created Makefile for each module of Apache Web Server.

1. make
   The make utility built the various modules and components that form the Apache Web Server package. The compilation process involved basic options for compiling, as well as some platform-specific back-end optimization options, as provided in the environment variable CFLAGS and LDFLAGS. During the compilation process under the make tool, the following warnings were generated when using the optimization flags:

   cc: Warning: -xarch=native has been explicitly specified, or implicitly specified by a macro option, -xarch=native on this architecture implies -xarch=v8plusa, which generates code that does not run on pre UltraSPARC processors

   This warning occurs due to the option -fast, which expands to -xarch=native, and says to optimize code for the native architecture. But this flag is overridden by -xarch=v8plus, to generate code for good performance on the SPARC architecture.

2. make install
   The make utility uses this option to install the Apache package in the path specified by the option --prefix, during the configure process. This option installed all the executables, libraries, configuration files, documentation, man pages, and so on, in the path specified. This was the final step in the process and the Apache Web Server was installed successfully.

Differences in Compiler Options Used

The compiler flags used to compile the application on the SPARC platform were as follows:
- xparallel: A macro equivalent to -xautopar, -xdepend, and -xexplicitpar. This option is used to parallelize loops specified automatically by the compiler and explicitly by the programmer. For faster code, this option is used for a multiprocessor machine. If optimization level is not at -xO3, it is raised to that level and a warning is issued.
- fast: This macro specifies a collection of options for optimizing the execution of applications. This macro expands to:

   -fns, -fsimple=2, -ftrap=%none, -xalias_level=basic, -xarch=native, -xbuiltin=%all, -xdepend, -xmemalign=8s, -xO5, -xprefetch=auto, explicit -xtarget=ultra

The -fast option is used to specify the target system for instruction set and optimization. In this case study, the machine used has value SUNW,Ultra obtained by the command `uname -i`. That's why -xtarget=ultra and -xarch=v8plus are being used to override these values in the -fast macro.
- xtarget=ultra: This option specifies the target platform for instruction set optimization. Each value of -xtarget expands into a specific set of values for the -xarch, -xchip, and -xcache options. The reason to use this option was to override the -target option used in the -fast option because, based on the target machine, this value is more suitable for enabling the compiler to generate code for good performance on the UltraSPARC architecture.
When the value of -xtarget is set to ultra then this option expands to:

-xtarget=ultra
  -xarch=v8plusa -xchip=ultra -xcache=64/32/4:8192/512/2
  -xarch=v8plus: This specifies the target instruction set architecture (ISA). This option was used to override the -xarch option used in the -xtarget option because this value is more suitable for enabling the compiler to generate code for good performance on the V8plus ISA.

The abovementioned compiler options worked correctly for the SPARC version, but if the same macros or compiler flags need to be used on the x86 architecture, then not all of them will be supported by the x86 platform. However, sometimes the same macro or option can be used on x86 but with different values.

Some of the SPARC options used to compile the Apache Web Server in this case study can be used on the x86 platform, but with different values. The options that can be used on the x86 version of the compiler are as follows:

- x-target=pentium_pro: This option is used to specify the target system for instruction set and optimization. In this case study, the processor used on the x86 machine is a Pentium III processor that falls in the Pentium_Pro processor family. The -xtarget macro expands to:
  -xarch=pentium_pro -xchip=pentium_pro -xcache=generic

Building the Application on Solaris 10 OS, x86 Platform Edition

The same steps were followed on the x86 machine:

1. configure
   First the configure utility was run without any modification in the flags what had been used for the SPARC version to obtain the Makefiles. This was done as follows:
   2. CFLAGS= "-xparallel -fast -xtarget=ultra -xarch=v8plus"
   3. LDFLAGS=" -xparallel -fast -xarch=v8plus"
   4. ./configure --prefix=/home/testuser/apache/apache2
   5. make

   Then make was used to build the application from the source code. When the same compiler flags were used on the x86 machine, the option -xparallel was ignored, while the options -xarch, -xtarget gave the following compiler warnings:

6. cc: Warning: illegal option -xtarget=ultra4
7. cc: Warning: illegal option -xarch=v8plus

   To prevent these warnings, a different set of compiler flags was used on the x86 platform. See the Solution for Solaris OS, x86 Platform Edition section for the compiler flags used.

8. make install
   The next step, make install, installed the Apache package in the path specified by the option --prefix during the configure process.
Solution for Solaris x86:
When built using the flags given by the SPARC version compiler, the source code gave a warning that improper options had been used. To compile the source on the x86 machine, the flag values were changed to:

9. \texttt{CFLAGS= -fast -xtarget=pentium\_pro -xarch=pentium\_pro} \ \textbackslash
10. \texttt{LDFLAGS= -fast -xarch=pentium\_pro} \ \textbackslash
11. \texttt{./configure –prefix=/home/testuser/apache/apache2}

The explanation of these options is:

- \texttt{-fast}: This macro differs from the SPARC platform in the options it expands to.
- \texttt{-xtarget=pentium\_pro}: This option optimizes code for the \texttt{pentium\_pro} architecture, since the processor used in the machine (Pentium III 550-MHz) for the case study belongs to the P6 (Pentium Pro) family.

After we chose these compiler flags, the build successfully completed on the x86 machine without any further compiler warnings.
8. References:
   - http://docs.hp.com/
   - http://docs.sun.com/