Supporting Multiple Page Sizes in the Solaris™ Operating System

Richard McDougall, PAE

Sun BluePrints™ OnLine—March 2004

`http://www.sun.com/blueprints`

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95045 U.S.A.
650 960-1300

Part No. 817-5917-10
Revision 1.0, 3/10/04
Edition: March 2004

Copyright 2004 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95045 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology described in this document. In particular, and without limitation, these intellectual property rights may include one or more patents or pending patent applications in the U.S. or other countries.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, Sun BluePrints, Sun Forte, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

U.S. Government Rights—Commercial use. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the Far and its supplements.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, Californie 95045 Etats-Unis. Tous droits réservés.

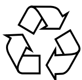
Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque enregistree aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

Sun, Sun Microsystems, le logo Sun, Sun BluePrints, Sun Forte, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.



Please
Recycle

Supporting Multiple Page Sizes in the Solaris™ Operating System

The availability of both processor and operating system (OS) support for 64-bit address spaces has enabled applications to take a quantum leap in the size and efficiency with which they manipulate data. UltraSPARC® processor-based servers from Sun Microsystems paired with the Solaris™ Operating System (Solaris OS) have enabled applications that were once limited to a virtual memory size of 4 gigabytes to a virtually unlimited span. This dramatic change in the size of virtual memory has changed the way developers create their applications and how organizations use them. Database management systems can have larger tables than ever before and often, they can fit entirely in main memory, significantly reducing their I/O requirements. Large-scale simulations can run with fewer restrictions on array sizes. In fact, working set sizes of several gigabytes are now common.

The performance of a memory-intensive application is dependent on the performance of the underlying memory management infrastructure. Time spent converting virtual addresses into physical addresses will slow down an application, often in a manner that is not evident to standard performance tools. In many cases, there is an opportunity to increase the performance of the underlying memory management infrastructure, resulting in higher application performance.

We can often increase an application's performance by increasing the memory management page size. The memory management unit (MMU) in Sun's UltraSPARC processors typically has just a few hundred entries, each of which can translate 8 kilobytes of address space by default, resulting in access to only a few megabytes of memory before performance is affected. Fortunately, recent improvements in the Solaris OS allow these limitations to be overcome.

Beginning with the Solaris 9 OS, multiple page sizes can be supported on UltraSPARC processors so administrators can optimize performance by changing the page size on behalf of an application. Typical performance measurement tools do not provide sufficient detail for evaluating the impact of page size and do not provide the needed support to make optimal page size choices.

This article explains how to use new tools to determine the potential performance gain. In addition, it explains how to configure larger page sizes using the multiple page size support (MPSS) feature of the Solaris 9 OS. The article addresses the following topics:

- "Understanding Why Virtual-to-Physical Address Translation Affects Performance" on page 2
- "Working With Multiple Page Sizes in the Solaris OS" on page 6
- "Configuring for Multiple Page Sizes" on page 14

Understanding Why Virtual-to-Physical Address Translation Affects Performance

The faster the microprocessor converts virtual addresses into physical addresses, the faster the application can run. Ideally, the MMU converts virtual addresses into physical addresses quickly enough (every microprocessor cycle) that the microprocessor won't stall and wait. Under certain circumstances, however, the translation process can take considerably longer. In fact, this can typically take from tens to hundreds of cycles.

We can often minimize the time taken to translate virtual addresses for applications with large working sets by increasing the page size used by the MMU. However, some applications that work well with small pages might see performance degradation when they are forced to use large pages. For example, processes that are short-lived, have small working sets, or have memory access patterns with poor spatial locality could suffer from the overhead of using large pages. Additionally, copy-on-write (COW) faults for a large page require a significant amount of time to process, as does first-touch access, which involves allocating and zeroing the entire page. For these reasons, we must analyze the application to determine whether the use of large pages is beneficial. Methods for determining when applications will benefit from large page sizes are presented later in this article.

Solaris OS Address Translation

Memory is abstracted so that applications only need to deal with virtual addresses and virtual memory. Behind the scenes, the OS and hardware, in a carefully choreographed dance, transparently translate the application's virtual addresses into the physical addresses for use by the hardware memory system.

The task of translating a virtual address into a physical address is accomplished by software and hardware modules. The software translates the mappings within an address space of a process (or the kernel) into hardware commands to program the microprocessor's MMU. The hardware then translates requests for virtual memory from the running instructions into physical addresses in real time. Optimally, this happens in the time of one microprocessor cycle. Some microprocessors, such as UltraSPARC, require assistance from the OS to manage this process, facilitated by a hardware-generated exception into system software where these helper tasks are performed.

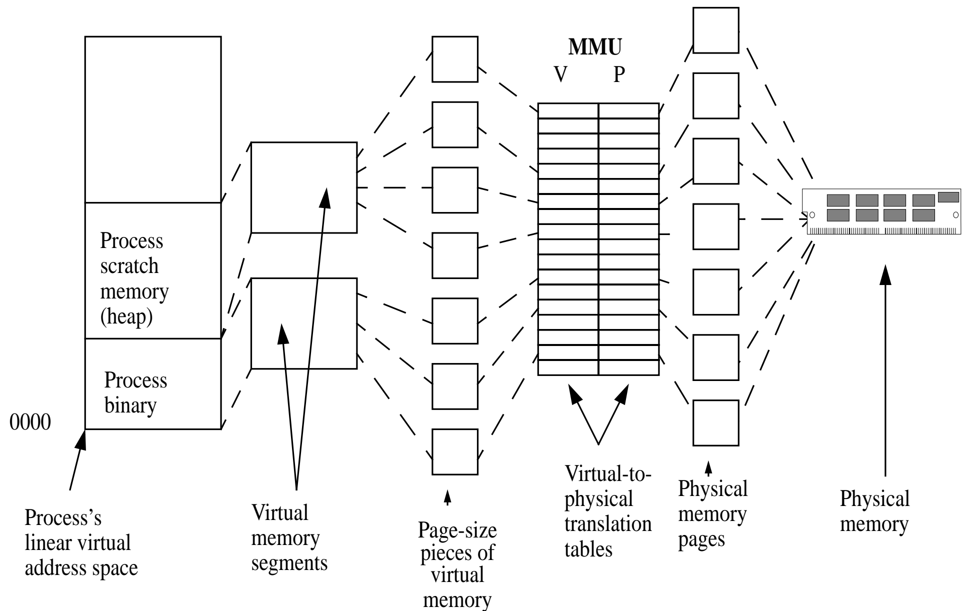


FIGURE 1 Solaris OS Virtual-to-Physical Memory Management

The combined virtual memory system translates virtual addresses into physical addresses in page-size chunks of memory as depicted in the preceding figure.

The hardware uses a table known as the *translation lookaside buffer* (TLB) in the microprocessor to convert virtual addresses to physical addresses on-the-fly. The software programs the microprocessor's TLB with entries identifying the relationship of the virtual and physical addresses. Because the size of the TLB is limited by hardware, the TLB is typically supplemented by a larger (but slower) in-memory tables of virtual-to-physical translations. On UltraSPARC processors, these tables are known as the *translation storage buffer* (TSB); on most other architectures, they are known as a *page table*. When the microprocessor needs to convert a virtual address into a physical address, it searches the TLB (a hardware search), and if a physical address is not found (for example, the hardware encounters a *TLB miss*), the microprocessor searches the larger in-memory table. The following figure illustrates the relationship of these components.

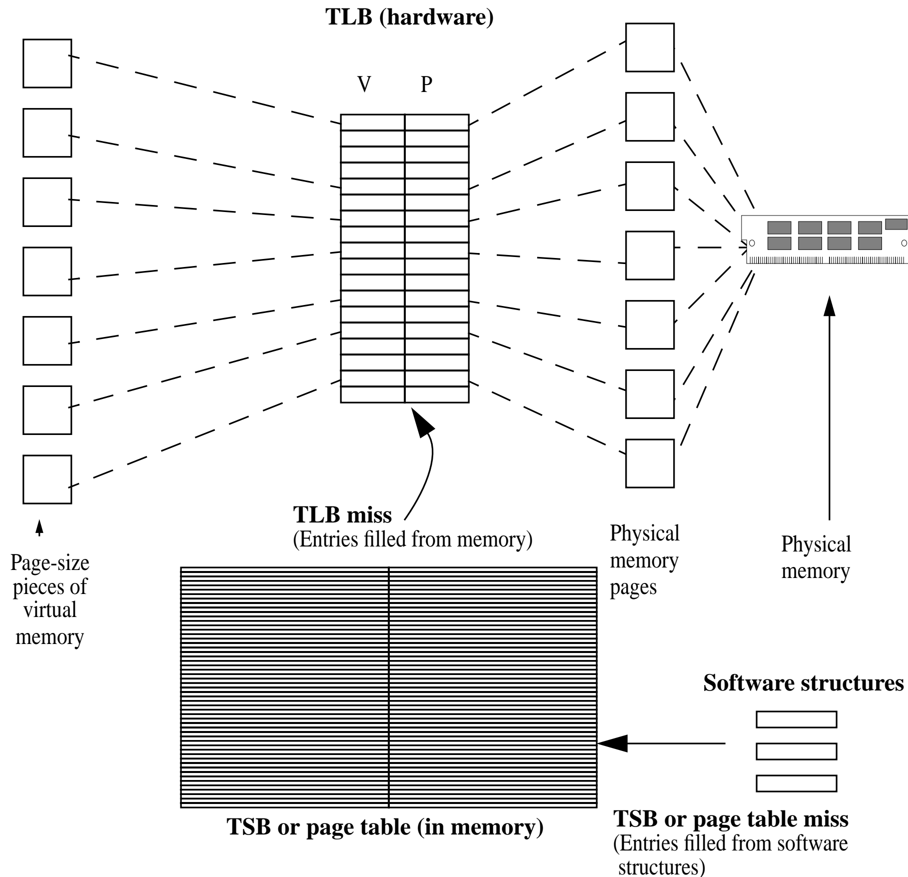


FIGURE 2 Virtual Address Translation Hardware and Software Components

UltraSPARC I–IV microprocessors use a software TLB replacement strategy. For example, when a TLB miss occurs, software is invoked to search the in-memory table (the TSB) for the required translation entry.

Let's step through a simple example. Suppose a process allocates some memory within its heap by calling `malloc`(). Further suppose that `malloc`() returns to the program a virtual address of the requested memory. When that memory is first referenced, the virtual memory layer requests a physical memory page from the system's free lists. The newly acquired page has an associated physical address within physical memory. The virtual memory system then constructs a translation entry in memory containing the virtual address (the start of the page returned by `malloc`) and the physical address of the new page. The newly created translation entry is then inserted into the TSB and programmed into an available slot in the microprocessor's TLB. The entry is also kept in software, linked to the address space

of the process to which it belongs. Later, when the program accesses the virtual address, if the new TLB entry is still in the TLB virtual-to-physical address is translated on-the-fly. However, if the TLB entry has been evicted by other activity, a TLB miss occurs. The corresponding hardware exception looks up the translation entry in the larger TSB and reloads it into the TLB.

The TSB is also limited in size. In extreme circumstances, a *TSB miss* can occur. Translating the virtual address of a TSB miss requires a lengthy search of the software structures associated with the process.

The mechanism is similar for other processors using hardware TLB-miss strategies, including the Intel x86, except that a hardware TLB replacement strategy is used to refill the TLB rather than software. When a TLB miss occurs, an in-hardware engine is invoked to search the page table.

TLB Reach and Application Performance

The objective of the TLB is to cache as many recent page translations in hardware as possible so that it can satisfy a process's memory accesses by performing all of the virtual-to-physical translations on-the-fly. Most TLBs are limited in size because of the amount of transistor space available on the CPU die. For example, the UltraSPARC I and II TLBs are only 64 entries. This means that the TLB can address no more than 64 pages of translations at any time; therefore, on UltraSPARC, the TLB can address 64×8 kilobytes (512 kilobytes).

The amount of memory the TLB can concurrently address is known as the *TLB reach*. The UltraSPARC I and II have a TLB reach of 512 kilobytes. If an application makes heavy use of less than 512 kilobytes of memory, the TLB will be able to cache the entire set of translations. However, if the application were to make heavy use of more than 512 kilobytes of memory, the TLB will begin to miss, and translations will have to be loaded from the larger TSB.

The following table shows the TLB miss rate and the amount of time spent servicing TLB misses from a study of older SPARC™ architectures. We can see from the table that only a small range of compute-bound applications fit well in the SuperSPARC™ TLB (`gcc`, `ML`, and `pthor`), whereas the others applications spend a significant amount of their time in the TLB miss handlers.

TABLE 1 Sample TLB Miss Data From a SuperSPARC Processor Study

Workload	Total Time (secs)	User Time (secs)	# User TLB Misses	% User Time in TLB Miss Handling	Cache Misses ('000s)	Peak Memory Usage (MB)
`coral`	177	172	85974	50	71053	19.9
`nasa7`	387	385	152357	40	64213	3.5
`compress`	99	77	21347	28	21567	1.4
`fftpde`	55	53	11280	21	14472	14.7
`wave5`	110	107	14510	14	4583	14.3

Workload	Total Time (secs)	User Time (secs)	# User TLB Misses	% User Time in TLB Miss Handling	Cache Misses (‘000s)	Peak Memory Usage (MB)
mp3d	37	36	4050	11	5457	4.8
spice	620	617	41923	7	81949	3.6
pthor	48	35	2580	7	6957	15.4
ML	945	917	38423	4	314137	32.0
gcc	118	105	2440	2	9980	5.6

TLB effectiveness has become a larger issue in the past few years because the average amount of memory used by applications has grown significantly (almost doubling year upon per year according to recent statistical data). The easiest way to increase the effectiveness of the TLB is to increase the TLB reach so that the working set of the application fits within the TLB’s reach.

The TLB reach can be improved using either of the following methods:

- Increase the number of entries in the TLB. This approach adds complexity to the TLB hardware and increases the number of transistors required; therefore, requiring more valuable die space.
- Increase the page size for each entry. This approach increases the TLB reach without the need to increase the size of the TLB.

One trade-off of increasing the page size is that doing so might boost the performance of some applications at the expense of slower performance elsewhere. This trade-off is caused by wasted space that results from larger memory allocation units. We would almost certainly increase the memory usage of many applications.

Luckily, a solution is at hand. Some of the newer processor architectures allow us to use two or more different page sizes at the same time. For example, UltraSPARC provides hardware support concurrently to select 8 kilobyte, 64 kilobyte, 512 kilobyte, or 4 megabyte pages. If we were to use 4 megabyte pages to map all memory, then the TLB would have a theoretical reach of 64×4 megabytes (256 megabytes).

Working With Multiple Page Sizes in the Solaris OS

This section introduces a strategy for measuring the potential performance gain that could be yielded from an increase in page size. We begin by describing a powerful tool in the Solaris 9 software, `trapstat`, for easily quantifying the potential gains of using a larger page size. This description is followed by sections that explain the methods we use to estimate the gain in the Solaris 8 OS using the `cpustat` command.

Deciding When to Use Large Pages

To determine whether we can improve application performance by using a larger page size, we need to determine the amount of time the microprocessor spends servicing TLB misses on behalf of a target application (See "Understanding Why Virtual-to-Physical Address Translation Affects Performance" on page 2 for a further information on why translation misses affect application performance).

TLB misses are typically accounted for in the context of the running process. For example, if a TLB miss occurs in a user-mode application, it will be counted as user time. Thus, an application might spend a large amount of time having TLB misses serviced, but still report that it spends 100 percent of its time in user mode, as shown in the following sample.

```
sol8# mpstat 1 3
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    2   0    1   234  134   91   46    0    0    0    25  100   0   0   0
  0    2   0    1   234  134   91   46    0    0    0    25  100   0   0   0
  0    2   0    1   234  134   91   46    0    0    0    25  100   0   0   0
```

Measuring Application Performance

Two different types of page size observability tools are available in Solaris software: those that describe the page sizes in use by the system or application, and those that help determine whether using large pages will benefit performance.

The `pmap`(1M), `pagesize`(1M), and `getpagesize`(3C) commands, and the `meminfo`(2) interfaces discover information about the system's ability to support different TLB page sizes. The `trapstat`(1M) and `cpustat`(1M) commands can approximate the amount of time that our target application spends waiting for the platform to service TLB misses.

We can use two methods to approximate the amount of time spent on servicing TLB misses:

- We can observe the rate of TLB misses and multiply rate of TLB misses by the cost of a TLB miss.
- Or, if TLB misses are serviced by system software, we can directly measure the time spent in TLB miss handlers.

In the Solaris 8 OS, the `cpustat`(1M) command measures the rate of TLB misses, whereas Solaris 9 software provides a new command, `trapstat`, which computes and displays the amount of time spent servicing TLB misses.

Determine the Number of TLB Misses With `trapstat`(1M)

The `trapstat` command in the Solaris 9 software provides information about processor exceptions on UltraSPARC platforms. Because TLB misses are serviced in software on UltraSPARC microprocessors, `trapstat` can also provide statistics about TLB misses.

Using the `trapstat` command, we can observe the number of TLB misses and the amount of time spent servicing TLB misses. The `-t` and `-T` options provide information about TLB misses. Again with `trapstat`, we can use the amount of time servicing TLB misses to approximate the potential gains we could make by using a larger page size or by moving to a platform that uses a microprocessor with a larger TLB.

The `-t` option provides first-level summary statistics. The time spent servicing TLB misses is summarized in the lower right corner; in this case, 46.2 percent of the total execution time is spent servicing misses. Miss details are provided for TLB misses incurred in the data portion of the address space, and for the instruction portion of the address space. Data is also provided for user-mode and kernel-mode misses. We are primarily interested in the user-mode misses because our application likely runs in user mode.

```
sol9# trapstat -t 1 111
cpu m| itlb-miss %tim itsb-miss %tim | dtlb-miss %tim dtsb-miss %tim |%tim
-----+-------------------------------+-------------------------------+----
  0 u|         1  0.0         0  0.0 |   2171237 45.7         0  0.0 |45.7
  0 k|         2  0.0         0  0.0 |      3751  0.1         7  0.0 | 0.1
=====+===============================+===============================+====
 ttl |         3  0.0         0  0.0 |   2192238 46.2         7  0.0 |46.2
```

For further detail, use the -T option to provide a per-page size breakdown. In this example, trapstat shows that all of the misses occurred are occurring on 8-kilobyte pages.

```
sol9# trapstat -T 1 111
cpu m size| itlb-miss %tim itsb-miss %tim | dtlb-miss %tim dtsb-miss %tim |%tim
----------+-------------------------------+-------------------------------+----
  0 u   8k|        30  0.0         0  0.0 |   2170236 46.1         0  0.0 |46.1
  0 u  64k|         0  0.0         0  0.0 |         0  0.0         0  0.0 | 0.0
  0 u 512k|         0  0.0         0  0.0 |         0  0.0         0  0.0 | 0.0
  0 u   4m|         0  0.0         0  0.0 |         0  0.0         0  0.0 | 0.0
- - - - - + - - - - - - - - - - - - - - - + - - - - - - - - - - - - - - - + - -
  0 k   8k|         1  0.0         0  0.0 |      4174  0.1        10  0.0 | 0.1
  0 k  64k|         0  0.0         0  0.0 |         0  0.0         0  0.0 | 0.0
  0 k 512k|         0  0.0         0  0.0 |         0  0.0         0  0.0 | 0.0
  0 k   4m|         0  0.0         0  0.0 |         0  0.0         0  0.0 | 0.0
==========+===============================+===============================+====
      ttl |        31  0.0         0  0.0 |   2174410 46.2        10  0.0 |46.2
```

We can conclude from this analysis that our application could potentially run almost twice as fast if we eliminated the majority of the TLB misses. Our objective in using the mechanisms discussed in the following sections is to minimize the user-mode data TLB (dTLB) misses, potentially by instructing the application to use larger pages for its data segments. Typically, data misses are incurred in the program's heap or stack segments. We can use the Solaris 9 software multiple-page size support commands to direct the application to use 4-megabyte pages for its heap, stack, or anonymous memory mappings.

Assess the Amount of Time Spent on TLB Misses With cpustat(1M)

The cpustat command programs and reads the hardware counters in the microprocessor. These counters measure hardware events within the processor itself. Typically, two counters and a larger number of events can be traced. The UltraSPARC III processors can count TLB miss events. Because the Solaris 8 OS lacks trapstat, the CPU counters can estimate the amount of time spent servicing TLB misses.

For example, the following `cpustat` command instructs the system to measure the number of dTLB miss events and the number of microprocessor cycles on each processor.

```
sol8# cpustat -c pic0=Cycle_cnt,pic1=DTLB_miss 1
  time   cpu event  pic0        pic1
  1.006   0  tick 663839993   3540016
  2.006   0  tick 651943834   3514443
  3.006   0  tick 630482518   3398061
  4.006   0  tick 634483028   3418046
  5.006   0  tick 651910256   3511458
  6.006   0  tick 651432039   3510201
  7.006   0  tick 651512695   3512047
  8.006   0  tick 613888365   3309406
  9.006   0  tick 650806115   3510292
```

By default, the `cpustat` command reports only counts that represent user-mode processes. This `cpustat` output shows us that on processor 0, a user mode process consumes approximately 650 million cycles per second and that 3.5 million dTLB misses per second are serviced. An UltraSPARC TLB miss typically ranges from about 50 cycles if the TLB entry being loaded is found in the microprocessor's cache to about 300 cycles if a memory load is required to fetch the new TLB entry. We can, therefore, approximate that between 175 million and 1050 million cycles are spent servicing TLB misses, per one-second sample.

A quick check of the processor speed allows us to calculate the ratio of time spent servicing misses.

```
sol8# psrinfo -v
Status of processor 0 as of: 11/10/2002 20:14:09
  Processor has been on-line since 11/05/2002 20:59:17.
  The sparcv9 processor operates at 900 MHz,
        and has a sparcv9 floating point processor.
```

Our microprocessor is running at 900 megahertz, providing 900 million cycles per second. Therefore, at least 175/900, or 19 percent of the time is spent servicing TLB misses. The actual number could be larger if a large fraction of the TLB misses require memory loads.

Determining Which Page Sizes Have Been Allocated

The pmap command allows us to query a target process about page size information, and the meminfo system call provides a programatic query to the OS for information about the page sizes provided to it.

Query a Process for Page Size Information With pmap(1)

The pmap command displays the page sizes of memory mappings within the address space of a process. The -sx option directs pmap to show the page size for each mapping.

```
sol9# pmap -sx `pgrep testprog`
2909:   ./testprog
 Address  Kbytes     RSS    Anon  Locked Pgsz Mode   Mapped File
00010000       8       8       -       -   8K r-x--  dev:277,83
ino:114875
00020000       8       8       8       -   8K rwx--  dev:277,83
ino:114875
00022000  131088  131088  131088       -   8K rwx--    [ heap ]
FF280000     120     120       -       -   8K r-x--  libc.so.1
FF29E000     136     128       -       -    - r-x--  libc.so.1
FF2C0000      72      72       -       -   8K r-x--  libc.so.1
FF2D2000     192     192       -       -    - r-x--  libc.so.1
FF302000     112     112       -       -   8K r-x--  libc.so.1
FF31E000      48      32       -       -    - r-x--  libc.so.1
FF33A000      24      24      24       -   8K rwx--  libc.so.1
FF340000       8       8       8       -   8K rwx--  libc.so.1
FF390000       8       8       -       -   8K r-x--  libc_psr.so.1
FF3A0000       8       8       -       -   8K r-x--  libdl.so.1
FF3B0000       8       8       8       -   8K rwx--    [ anon ]
FF3C0000     152     152       -       -   8K r-x--  ld.so.1
FF3F6000       8       8       8       -   8K rwx--  ld.so.1
FFBFA000      24      24      24       -   8K rwx--    [ stack ]
-------- ------- ------- ------- -------
total Kb  132024  132000  131168       -.
```

The pmap command shows the MMU page size for each mapping. In this case, 8 kilobytes are used for all mappings. To demonstrate a larger page size, we can use the ppgsz command in the Solaris 9 software to set the page size for the heap of our test program to 4 megabytes. The ppgsz command is described in more detail in a later section.

```
sol9# ppgsz -o heap=4M ./testprog &
sol9# pmap -sx `pgrep testprog`
2953:   ./testprog
 Address  Kbytes     RSS    Anon  Locked Pgsz Mode   Mapped File
00010000       8       8       -       -   8K r-x--  dev:277,83
ino:114875
00020000       8       8       8       -   8K rwx--  dev:277,83
ino:114875
00022000    3960    3960    3960       -   8K rwx--    [ heap ]
00400000  131072  131072  131072       -   4M rwx--    [ heap ]
FF280000     120     120       -       -   8K r-x--  libc.so.1
FF29E000     136     128       -       -    - r-x--  libc.so.1
FF2C0000      72      72       -       -   8K r-x--  libc.so.1
FF2D2000     192     192       -       -    - r-x--  libc.so.1
FF302000     112     112       -       -   8K r-x--  libc.so.1
FF31E000      48      32       -       -    - r-x--  libc.so.1
FF33A000      24      24      24       -   8K rwx--  libc.so.1
FF340000       8       8       8       -   8K rwx--  libc.so.1
FF390000       8       8       -       -   8K r-x--  libc_psr.so.1
FF3A0000       8       8       -       -   8K r-x--  libdl.so.1
FF3B0000       8       8       8       -   8K rwx--    [ anon ]
FF3C0000     152     152       -       -   8K r-x--  ld.so.1
FF3F6000       8       8       8       -   8K rwx--  ld.so.1
FFBFA000      24      24      24       -   8K rwx--    [ stack ]
-------- ------- ------- ------- -------
total Kb  135968  135944  135112       -
```

Retrieve a Page Description With meminfo(2)

The meminfo() system call enables a program to inquire about the physical pages mapping its address space. This system call provides a programmatic way of determining the page sizes allocated within a process's address space. An array is filled with a description of each page that backs the mapping. For more information, refer to the meminfo(3c) man page.

Discovering Supported Page Sizes

This section describes the three commands that enable us to determine information about the page size supported by the Solaris 9 OS.

Determine Page Size With `pagesize`(1M)

The `pagesize` command displays the default page size used by the Solaris OS on the given microprocessor. The default is currently 8 kilobytes for all UltraSPARC platforms.

```
sol8# pagesize
8192
```

The `pagesize` command can also display the available page sizes on the given microprocessor in the Solaris 9 OS. In this example, we can see that four page sizes are available on our UltraSPARC processor.

```
sol9# pagesize -a
8192
65536
524288
4194304
```

Retrieve the Base Page Size With `getpagesize`(3C)

The `getpagesize`() function returns the base page size in bytes.

Retrieve the Microprocessor's Available Page Size With `getpagesizes`(3C)

The `getpagesizes`() function reports the available page sizes on the given microprocessor. For more information, refer to the `getpagesizes`(3c) man page.

Configuring for Multiple Page Sizes

After determining that our application warrants the use of large pages, we need to construct a strategy for determining what parts of our application to enhance to use large pages. For example, we should consider whether we should attempt to enable large pages for our target process's heap and stack. The `trapstat` utility provides a little information about the types of address space that incur TLB misses.

The *instruction TLB* (iTLB) miss information is likely a result from the process's text and library text because instructions typically reside in these mappings. It is possible, however, for a program to execute code from other mappings; for example, the Java virtual machine compiles instructions on-the-fly into its heap and then executes from there. However, for the vast majority of applications, we can first guess that iTLB misses result from the text or library mappings.

Data TLB misses are likely to occur from the program's writable segments (its heap, stack, data mapping, and read-only data within the text mapping).

The default page size for the Solaris OS is 8 kilobytes on UltraSPARC and 4 kilobytes on Intel x86 microprocessors. Larger pages of 4 megabytes are used by the Solaris kernel for its instruction and data sections; however, user applications requiring larger pages must explicitly request them.

The use of larger page sizes in the Solaris 2.6 OS through the Solaris 8 OS is only available through a special form of System V shared memory. To optimize database performance, we can use a form of shared memory called intimate shared memory (ISM). ISM is requested by the `shmat`(2) system call with the `SHM_SHARE_MMU` flag and is allocated as 4 megabyte pages, if possible. Databases such as Oracle, Informix, and Sybase request shared memory by using this flag and typically perform as much as 10 percent to 20 percent better as a result of a reduced TLB miss rate.

The Solaris 9 OS introduces a generic framework for allowing user applications to request larger page sizes. At the same time, ISM was also enhanced to take advantage of the other supported large page sizes, for example, 64 kilobytes and 512 kilobytes. Unmodified applications can be directed to use larger page sizes by means of the `ppgsz`(1M) command and the `libmpss.so` library. Applications can also be customized to request larger page sizes by the `memcntl`(2) system call.

The Solaris 9 OS large-page infrastructure allows larger pages to be requested for the mappings of `/dev/zero`, that is, for the heap, stack, and other anonymous mappings.

Enabling Large Pages in the Solaris 9 OS

The new framework, MPSS, provided in the Solaris 9 OS allows larger page sizes to be requested for user processes. The `memcntl`() system call specifies page-size advice for a given address range. A wrapper program, `ppgsz`, and an interposition library, `libmpss.so`, call `memcntl`() on behalf of the target process so that unmodified binaries can make use of larger page sizes.

Advising Page-Size Preferences With `ppgsz`(1M)

The `ppgsz` command is a wrapper that advises a preferred page size for a process's heap or stack of a target process. These page-size preferences are inherited across `fork`() but not across `exec`(). Thus, if the target program spawns (forks then execs) another program, page sizes are not inherited. If inheritance of page sizes is required, the `libmpss.so` library should be used instead.

For example, to start a target process with 4 megabyte pages for its heap, we could use the `ppgsz` wrapper.

```
sol9# ppgsz -o heap=4M ./testprog &
sol9# pmap -sx `pgrep testprog`
2953:   ./testprog
 Address  Kbytes     RSS    Anon  Locked Pgsz Mode   Mapped File
00010000       8       8       -       -   8K r-x--  dev:277,83
ino:114875
00020000       8       8       8       -   8K rwx--  dev:277,83
ino:114875
00022000    3960    3960    3960       -   8K rwx--    [ heap ]
00400000  131072  131072  131072       -   4M rwx--    [ heap ]
FF280000     120     120       -       -   8K r-x--  libc.so.1
FF29E000     136     128       -       -    - r-x--  libc.so.1
FF2C0000      72      72       -       -   8K r-x--  libc.so.1
FF2D2000     192     192       -       -    - r-x--  libc.so.1
FF302000     112     112       -       -   8K r-x--  libc.so.1
FF31E000      48      32       -       -    - r-x--  libc.so.1
FF33A000      24      24      24       -   8K rwx--  libc.so.1
FF340000       8       8       8       -   8K rwx--  libc.so.1
FF390000       8       8       -       -   8K r-x--  libc_psr.so.1
FF3A0000       8       8       -       -   8K r-x--  libdl.so.1
FF3B0000       8       8       8       -   8K rwx--    [ anon ]
FF3C0000     152     152       -       -   8K r-x--  ld.so.1
FF3F6000       8       8       8       -   8K rwx--  ld.so.1
FFBFA000      24      24      24       -   8K rwx--    [ stack ]
-------- ------- ------- ------- -------
total Kb  135968  135944  135112       -
```

Interposing Shared Libraries With libmpss.so

The mpss.so shared object in /usr/lib provides a means by which the preferred stack or heap page size can be selectively configured for launched processes and their descendants. The library has an the advantage over the wrapper in that page sizes are inherited across exec(). To enable mpss.so, ensure that the following string is present in the environment (see ld.so.1(1)) along with one or more MPSS environment variables.

```
sol9# LD_PRELOAD=$LD_PRELOAD:mpss.so.1
```

Once preloaded, the mpss.so.1 shared object reads the following environment variables to determine preferred page size requirements and processes for which these requirements are specified.

```
MPSSHEAP=size

     MPSSSTACK=size
           MPSSHEAP and  MPSSSTACK  specify  the  preferred  page
           sizes for the heap and stack, respectively. The speci-
           fied  page  size(s)  are  applied   to   all   created
           processes.

     MPSSCFGFILE=config-file
           config-file is a text file which contains one or  more
           mpss configuration entries of the form:

           exec-name:heap-size:stack-size
```

For example, the following commands enable 4-megabyte pages for the heap of all subsequently started processes.

```
sol9# export LD_PRELOAD=$LD_PRELOAD:mpss.so.1
sol9# export MPSSHEAP=4M
sol9# ./testprog
```

See mpss.so.1(1) for all available configuration options.

Compiling Your Application to Request Larger Page Sizes

The Sun Forte™ 8 compilers provide options to cause the target application to request specific page sizes. The following options are supported for the compiler.

Set Stack and Heap Page Size With `-xpagesize=n`

(SPARC) Sets the preferred page size for the stack and the heap. The `n` value must be one of the following: 8K, 64K, 512K, 4M, 32M, 256M, 2G, 16G, or default.

You must specify a valid page size for the Solaris OS on the target platform, as returned by `getpagesize`(3C). If you do not specify a valid page size, the request is silently ignored at run-time. The Solaris OS offers no guarantee that the page size request will be honored. You can use `pmap`(1) or `meminfo`(2) to determine page size of the target platform.

The `-xpagesize` option has no effect unless you use it at compile time and at link time.

Note – This feature is not available on the Solaris 7 OS and the Solaris 8 OS. A program compiled with this option will not link on the Solaris 7 OS or the Solaris 8 OS.

If you specify `-xpagesize=default`, the Solaris OS sets the page size. `-xpagesize` without an argument is the equivalent to `-xpagesize=default`.

Compiling with this option has the same effect as setting the `LD_PRELOAD` environment variable to `mpss.so.1` with the equivalent options, or running the `ppgsz`(1) command in the Solaris 9 software with the equivalent options before running the program. See the man pages for the Solaris 9 OS for details.

This option is a macro for `-xpagesize_heap` and `-xpagesize_stack`. These two options accept the same arguments as `-xpagesize`: 8K, 64K, 512K, 4M, 32M, 256M, 2G, 16G, or default. You can set them both with the same value by specifying `-xpagesize` or you can specify them individually with different values.

Set Heap Page Size in Memory With `-xpagesize_heap=n`

(SPARC) Sets the page size in memory for the heap. The `n` value can be 8K, 64K, 512K, 4M, 32M, 256M, 2G, 16G, or default. You must specify a valid page size for the Solaris OS on the target platform, as returned by `getpagesize`(3C). If you do not specify a valid page size, the request is silently ignored at run time.

You can use pmap(1) or meminfo(2) to determine page size at the target platform. If you specify -xpagesize_heap=default, the Solaris OS sets the page size. -xpagesize_heap without an argument is the equivalent to -xpagesize_heap=default.

Compiling with this option has the same effect as setting the LD_PRELOAD environment variable to mpss.so.1 with the equivalent options, or running the ppgsz(1) command in the Solaris 9 software with the equivalent options before running the program. See the man pages for the Solaris 9 OS for details.

Note – This feature is not available on the Solaris 7 OS and the Solaris 8 OS. A program compiled with this option will not link on the Solaris 7 OS or the Solaris 8 OS.

Set Stack Page Size in Memory With -xpagesize_stack=n

(SPARC) Set the page size in memory for the stack. n can be 8K, 64K, 512K, 4M, 32M, 256M, 2G, 16G, or default. You must specify a valid page size for the Solaris OS on the target platform, as returned by getpagesize(3C). If you do not specify a valid page size, the request is silently ignored at run-time. You can use pmap(1) or meminfo(2) to determine page size at the target platform.

If you specify -xpagesize_stack=default, the Solaris OS sets the page size. -xpagesize_stack without an argument is the equivalent to -xpagesize_stack=default.

Compiling with this option has the same effect as setting the LD_PRELOAD environment variable to mpss.so.1 with the equivalent options, or running the Solaris 9 command ppgsz(1) with the equivalent options before running the program. See the man pages for the Solaris 9 OS for details.

Note – This feature is not available on the Solaris 7 OS and the Solaris 8 OS. A program compiled with this option will not link on the Solaris 7 OS or the Solaris 8 OS.

Enhancing an Application to Request Larger Page Sizes

The memcntl(3C) interface has been enhanced to allow page size requests to be made on behalf of a process. Thus, an application can automatically request larger page sizes when appropriate. Such an application wanting to request a larger page size should do so by using the existing memcntl() interface.

```
int memcntl(caddr_t addr, size_t len, int cmd, caddr_t arg,int attr, int mask);
```

With the cmd argument, we can now specify a new control operation, MC_HAT_ADVISE, for page-size operations. When the cmd argument is set to MC_HAT_ADVISE, the arg argument is interpreted as a pointer to a new structure, as shown below. Currently, only three commands are supported; each command sets a preferred page size. mha_flags must always be set to zero. It is reserved for future use. Only one command can be specified at a time.

```
struct memcntl mha{
        uint_t mha_cmd; /* command(s) */
        uint_t mha_flags; /* flags */
        size_t mha_pagesize;
};
```

If mha_cmd is set to MHA_MAPSIZE_VA, we apply the set preferred page-size operation to the address range (addr, addr + len). mha_pagesize must be a supported page size, as returned by getpagesizes(), or zero to let the system select the page size. The address and size of the range must be aligned to the new preferred page size. The access protections within new page-size regions contained in the range must be the same or the operation will fail. If there are holes in the address range or if the mapping is mapped with MAP_NORESERVE, the operation will fail. The address range can be contained inside a larger mapping or can span many mappings of varying sizes.

The memcntl() interface promotes or demotes the preferred page sizes for any MAP_PRIVATE /dev/zero mappings, provided that the constraints mentioned above are met. Two special objects in the user address space require special handling: the process's heap and the primary thread stack (not the stack for additional threads).

The heap consists of the last .bss adjacent to the brk area and the brk area itself. The following figure illustrates the mapping procedure.

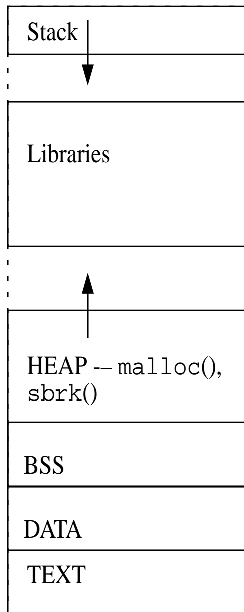


FIGURE 3 Process Address Space Mappings

For these two cases we have separate commands.

```
MHA_MAPSIZE_STACK  /* token for processes main stack */
MHA_MAPSIZE_BSSBRK /* token heap */
```

When MHA_MAPSIZE_STACK and MHA_MAPSIZE_BSSBRK are used, mha_pagesize must be a supported page size, as returned by getpagesizes(3C), or zero to let the system select the page size. The operation is then applied to the entire existing stack or heap mappings. The advice is then used for future page allocations. These commands for changing the preferred page size for stack or heap may first adjust the existing range in accordance with the new page size. This could involve creating new segments to pad out the base and length of the existing range to the new, preferred, page-size alignment.

Applications need to know what to align their memory requests on to attain maximum performance (for example, when using mmap() for creating new mappings) and to avoid misaligned mprotect(), munmap(), and mmap() requests that could result in page demotion, which is when larger pages are broken up into smaller pages.

Most applications that use mmap() pass in NULL for its addr argument to let the OS manage its address space. If applications also want to use large pages with memcntl(), they should suggest to the OS that it specify, by means of a new flag, MAP_ALIGN, the minimum page size alignment desired. If specified, mmap() interprets the addr argument only as the required minimum alignment and is free to find a hole in the user address space that satisfies the minimum alignment specified in the addr argument. The alignment must be a power of two multiple of PAGESIZE, or zero to let the system choose the alignment. If MAP_ALIGN is specified along with MAP_FIXED, the request will fail. If the alignment request cannot be satisfied, mmap() will also fail.

For reference, we provide the following example. This code fragment sets the page size for the program's heap to 4 megabytes. Note the use of memalign to align the request on a 4-megabyte boundary. Because the heap starts on a boundary that is not 4-megabyte aligned, the first few megabytes of the heap can reside on 8-kilobyte pages. If the performance-sensitive data structures reside within this area, the program might not realize the full benefits of a larger page size. By allocating a 4-megabyte aligned area, we increase the chance that the subsequent virtual addresses allocated will land on a large page.

```
#include <sys/types.h>
#include <sys/mman.h>
#include <stdlib.h>

#define MEGABYTE ((size_t)(1024 * 1024))
#define FOUR_MEGABYTE ((size_t)4 * MEGABYTE)

int
main(int argc, char *argv[])
{
        struct memcntl_mha mha;
        char *my_memory;

        /* Set pagesize to 4MB for heap */
        mha.mha_cmd = MHA_MAPSIZE_BSSBRK;
        mha.mha_flags = 0;
        mha.mha_pagesize = FOUR_MEGABYTE;
        memcntl(NULL, 0, MC_HAT_ADVISE, (char *)&mha, 0, 0);

        /* Ensure user memory starts on first large page */
        my_memory = (char *)memalign(FOUR_MEGABYTE, (size_t)100 *
MEGABYTE);
```

Determining Whether Your UltraSPARC CPU Model Works Well With Large Pages

The TLB configurations are quite different across versions of UltraSPARC processors, but they share a few items in common. UltraSPARC I through IV supports four page sizes: 8 kilobytes, 64 kilobytes, 512 kilobytes, and 4 megabytes. In addition, there are separate TLBs for the instruction and data paths.

UltraSPARC I and II

The UltraSPARC I and II microprocessors (143 megahertz–480 megahertz) have two TLBs, one for the instruction path and one for the data path. Each TLB is a 64-entry, fully associative TLB that supports all four page sizes. User applications can use any of the four page sizes.

750 Megahertz UltraSPARC III

The 750 megahertz UltraSPARC III microprocessor has four TLBs: two for instruction and two for data. The instruction TLBs are implemented as a 16-entry, fully associative TLB that supports all four page sizes and a larger 128-entry TLB that supports only 8 kilobyte entries. The data TLBs are implemented as a 16-entry, fully associative TLB that supports all four page sizes and a larger 512-entry, two-way set associative TLB that supports only 8 kilobyte entries.

The 16-entry dTLB has nine locked entries, which are locked by software for the Solaris kernel, leaving only seven slots for large page sizes. Thus, use of large pages is typically not beneficial on 750 megahertz UltraSPARC III systems.

900 Megahertz+ UltraSPARC III

The 900 megahertz onwards UltraSPARC III microprocessors have five TLBs: two for instruction and three for data. The instruction TLBs are configured as a 16-entry, fully associative TLB that supports all four page sizes and a larger 128-entry TLB that supports only 8 kilobyte entries. The data TLBs are configured as a 16-entry, fully associative TLB that supports all four page sizes and two larger 512-entry, two-way set associative TLBs that support one page size per process. The increased size of the data TLBs on 900 megahertz UltraSPARC III provides a large TLB spread (2 gigabytes when 4 megabyte pages are used) and typically increases performance significantly for large memory applications.

The large data TLBs are configured automatically in accordance with the most common page sizes in a process's address space. A process using one large page size in addition to the base page size (8 kilobytes) will have one of its large TLBs

automatically programmed to enable the large page size when eight or more pages are using the larger page size within the process. It is assumed that the smaller TLB is available if there are fewer than eight pages.

Because the large TLBs support all four page sizes, large pages can be used effectively on UltraSPARC III. However, because the large TLBs can be configured for only one page size at a time per process, only two page sizes should be used concurrently. One of those page sizes should be the system's base page size (8 kilobytes) for mappings not using large pages—for example, the program text or libraries. The other larger page size is available for the remainder of the mappings. The most common selections for page sizes are 8 kilobytes and 4 megabytes, providing the greatest TLB spread for the large TLB.

About the Author

Richard has over 15 years of UNIX experience including application design, kernel development, and performance analysis. Richard specializes in operating system tools and architecture.

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at: `http://www.sun.com/blueprints/online.html`