Developing and Tuning Applications on UltraSPARC T1 Chip Multithreading Systems

---

Denis Sheahan, Senior Staff Engineer

UltraSPARC T1 Architecture Group

October 2005

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

Part No. 819-1319-10
Revision 1.0, October 2005

Copyright 2005 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, Java, J2EE, J2SE, JDK, JVM, Solaris, and Sun Fire 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.

ORACLE is a registered trademark of Oracle Corporation.

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 2005 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.

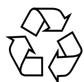
Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque enregistree aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

Sun, Sun Microsystems, le logo Sun, Java, J2EE, J2SE, JDK, JVM, Solaris, et Sun Fire 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.

ORACLE est une marque déposée registre de Oracle Corporation.

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

Developing and Tuning Applications on UltraSPARC T1 Chip Multithreading Systems

The UltraSPARC® T1 processor combines chip multiprocessing (CMP) and hardware multithreading (MT) with an efficient instruction pipeline to enable chip multithreading (CMT). The resulting processor design provides multiple physical-instruction execution pipelines and several active thread contexts per pipeline.

UltraSPARC T1 processors offer a new thread-rich environment for developers and can be thought of as symmetric multiprocessing (SMP) on a chip. Applications that currently scale well on SMP machines should also scale well in UltraSPARC T1 processor-based environments. Some simple modifications to an application can often achieve even greater scalability in the UltraSPARC T1 processor-based environment than can be achieved on an SMP machine.

This paper describes techniques that system architects, application developers, and performance analysts can use to assess the scaling characteristics of an application. It also explains how to optimize an application for CMT, in particular for systems that use UltraSPARC T1 processors.

This paper covers the following topics:

- "Processor Physical Characteristics" on page 2
- "Performance Characteristics" on page 3
- "Classes of Commercial Applications" on page 5
- "Assessing Performance on UltraSPARC T1 Processor-Based Systems" on page 7
- "Scaling Applications With CMT" on page 14
- "Tuning for General Performance" on page 23
- "Accessing the Modular Arithmetic Unit and Encryption Framework" on page 39
- "Minimizing Floating-Point Operations and VIS Instructions" on page 41

Note – This paper distinguishes between a hardware thread, also known as a strand, and a software thread of execution, also know as a lightweight process (LWP). This distinction is important because hardware thread scheduling is not under the control of software.

Processor Physical Characteristics

Sun's UltraSPARC T1 CMT hardware architecture has the following characteristics:

- Eight cores, or individual execution pipelines, per chip.
- Four strands, or active thread contexts, for each core that shares a pipeline. Each cycle of a different hardware strand is scheduled on the pipeline in round robin order.
- 32 threads total per UltraSPARC T1 processor.
- Cores that are connected by a high-speed, low-latency crossbar in the silicon. A UltraSPARC T1 processor can be considered SMP on a chip.
- Hardware strands that are presented by the Solaris Operating System (Solaris OS) as a "processor." For example, `mpstat`(1) and `psradm`(1M) show 32 "CPUs."
- Cores that have an instruction cache, a data cache, an instruction translation-lookaside buffer (iTLB), and a data TLB (dTLB) shared by the four strands.
- Strands defined by a set of unique registers and logic to control state.
- A twelve-way associative unified Level 2 (L2) on-chip cache. Each hardware strand shares the whole L2 cache.
- Memory latency that is unified from all cores—uniform memory access (UMA), not non-uniform memory access (NUMA).
- Full SPARC V7, V8, and V9 binary compatibility.
- Low-latency Double Data Rate 2 (DDR2) memory to reduce stalls. Four on-chip memory controllers that provide high memory bandwidth (theoretical maximum is 20 gigabytes per second).
- An OS scheduler that schedules LWPs on UltraSPARC T1 hardware strands. It is the task of the hardware to schedule strands in the core.
- A modular arithmetic unit (MAU) for each core that supports modular multiplication and exponentiation to help accelerate Secure Sockets Layer (SSL) processing.
- A single Floating Point Unit (FPU) that is shared by all cores, which makes UltraSPARC T1 processors a suboptimal choice for applications with floating point intensive requirements.

FIGURE 1 illustrates the structure of the UltraSPARC T1 processor.

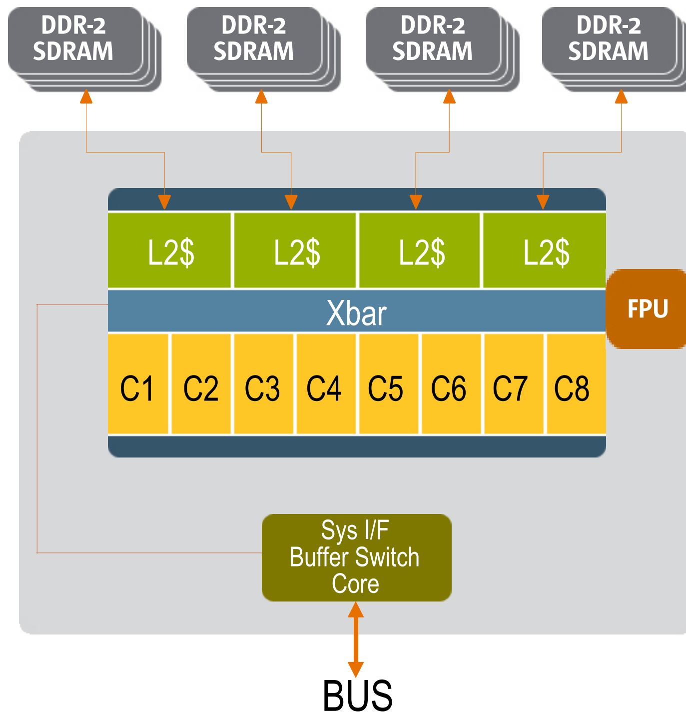


FIGURE 1 UltraSPARC T1 Processor Design

In addition, the processor exhibits the following behaviors and limitation:

- A strand that stalls for any reason is switched out and its slot on the pipeline is given to the next strand automatically. The stalled strand is inserted back in the queue when the stall is complete.
- An UltraSPARC T1 processor cannot be configured as an SMP system with multiple chips using a single Solaris image.

Performance Characteristics

The UltraSPARC T1 processor-based architecture has numerous implications for application performance, as described in the following sections.

Single Hardware Strands

In a CMT architecture, single-strand performance can mean one of two things:

- The performance of a strand that has exclusive access to a core. This is achieved by parking the other three strands, and is referred to as *exclusive single-strand performance*.
- The performance of a single strand when the other three strands are sharing the core, called *shared single-strand performance*.

In systems that use UltraSPARC T1 processors, the performance of a single hardware strand is lower than the performance of today's SPARC cores. Even exclusive single-strand performance is less than current SPARC architectures because the pipeline itself is simpler. However, although individual strands are weaker, aggregate performance can be much better than in the previous generation of SPARC processors.

Single-threaded performance is not a design goal of CMT—higher throughput is the aim of UltraSPARC T1 processors. By running LWPs on all 32 threads in parallel, the aggregate performance of UltraSPARC T1 processors is far better than today's SPARC cores. This is the tradeoff of single-thread performance versus throughput.

Threads Running Together on a Core

When threads run together on a core they affect each other; this effect can be positive or negative. For example, if a strand is stalled, its cycles can be utilized by other threads. Conversely, if the LWPs running on the four strands of a core have little or no stall component, then all strands get approximately one-quarter of the CPU cycles. In modern commercial applications, this is a rare scenario.

Further, if one thread is thrashing the Level 1 instruction cache, data cache, or TLBs on a core, it can adversely affect other threads on that core. And, if many LWPs are running the same application (as is common in today's deployments), they benefit from constructive sharing of text and data in the L2 cache.

Memory Latency

Memory latency is much less of an issue for application performance than on current non-CMT processors. Increases in processor speed have significantly outpaced memory speed bumps. The speed gap has increased over time. As a result, application performance is often constrained by memory latency. That is, instruction execution stalls when data is not in the cache and needs to be fetched from main memory.

The effect of automatically switching a stalled strand for a non-stalled strand is to hide latency in the chip. Of course, the individual strand will stall, but the core, and ultimately the chip, can continue to feed the pipeline by executing instructions from the other strands. The negative effects of memory latency are mitigated through more effective use of the pipeline.

Transaction-Lookaside Buffers

A translation-lookaside buffer (TLB) is a hardware cache that is used to translate a process's virtual address to a system's physical memory address. This piece of hardware is essential for performance. Each core in an UltraSPARC T1 processor-based system has a 64-entry iTLB and a similar dTLB. All four strands share this resource. Performance can be improved by using large pages, where possible, through the Solaris OS virtual memory features. For more information, see "Tuning a TLB" on page 36.

Classes of Commercial Applications

The UltraSPARC T1 processor was designed to scale well with applications that are throughput oriented. There are many classes of applications that exhibit throughput scaling, as described in the following subsections. Applications that fall under these categories should see improved performance when run in an environment that uses UltraSPARC T1 processors. Many applications that scale only to a small number of threads can also achieve good throughput by using multiple instances.

Multithreaded Applications

Multithreaded applications are characterized by a small number of highly threaded processes. These applications scale by scheduling work using Solaris OS threads. Threads often communicate through shared global variables.

Examples of multithreaded applications include the Sybase Database Engine and Siebel CRM.

Most multithreaded applications scale well on UltraSPARC T1 processor-based systems.

MultiProcess Applications

Multiprocess applications are characterized by many single-threaded processes, often communicating through shared memory or another interprocess communication (IPC) mechanism. These applications scale by scheduling work on processes.

Examples of multiprocess applications include ORACLE®, SAP, and Peoplesoft applications.

Most multiprocess applications scale well on UltraSPARC T1 processor-based systems.

Java Technology-Based Applications

Java technology-based applications scale through the Java™ Virtual Machine (JVM™ machine), a highly threaded process that provides scheduling and memory management.[1]

Examples include the Sun Java Application Server, BEA WebLogic, IBM WebSphere, and the open source Tomcat application servers. All Java 2 Platform, Enterprise Edition™ (J2EE™) applications that use an application server fall into this category.

UltraSPARC T1 processors are an excellent choice for Java technology-based applications.

Single-Threaded Applications

Single-threaded applications that have a severe response-time constraint are not good candidates for UltraSPARC T1 processor-based systems. The weaker single-thread performance can make the response time unobtainable. Single-threaded batch applications are an example of such code.

Floating-Point Intensive Code

The UltraSPARC T1 processor has a single floating-point unit. If more than 1 percent of an application consists of floating-point instructions, it is not a good candidate for UltraSPARC T1 processor-based systems. Netra ct 800 server simulation is an example of such an application.

1. References to the Java Virtual Machine and JVM are to the Virtual Machine for the Java platform.

Scaling Through Multiple Instances

If an application does not scale to a high number of CPUs, it is still possible to use UltraSPARC T1 processors by running multiple instances in parallel. This is a common technique for Java technology-based applications. Using the Solaris™ Zones feature, described "Using the Solaris Zones Feature to Isolate Workloads" on page 30, can simplify this process.

Assessing Performance on UltraSPARC T1 Processor-Based Systems

When assessing the performance of an UltraSPARC T1 processor-based system, consider the system's overall performance, parallelism, and throughput. This section explains how to make these assessments and provides information about using Solaris™ Dynamic Tracing (DTrace) for debugging and performance analysis.

Assessing Overall Performance

To begin assessing performance, observe an application running on a current or UltraSPARC T1 processor-based system with `mpstat`(1) and `vmstat`(1). `mpstat`(1) displays 32 processors on an eight-core system and 24 processors on a six-core system. The first line in the `mpstat` and `vmstat` sample should be discarded because it is an accumulation of data since booting the system.

CODE EXAMPLE 1 shows the `mpstat` output from a Sun Fire™ 490 server.

CODE EXAMPLE 1 `mpstat` Output From a Sun Fire 490 Server

```
 CPU minf  mjf xcal intr ithr  csw icsw migr smtx  srw syscl  usr  sys   wt  idl
  16    9    0    1  212  109    0    0    0    0    0    12    0    0    0  100
  17    0    0    1    4    2    1    0    1    0    0     5    0    0    0  100
  18   26    0   54    4    2    5    0    1    0    0    59    0    0    0  100
  19    7    0    1    4    1    2    0    1    0    0    14    0    0    0  100
 528    9    0    1    4    1   24    0    2    2    0    21    0    0    0  100
 529    2    0    1    4    1    1    0    0    0    0    10    0    0    0  100
 530   18    0   16    4    1    5    0    1    2    0    33    0    0    0  100
 531    9    0    1    4    1    7    0    1    1    0    25    0    0    0  100
```

CODE EXAMPLE 2 shows the vmstat output from the same system.

CODE EXAMPLE 2 vmstat Output From a Sun Fire 490 Server

```
kthr          memory            page                        disk        faults      cpu
r b w   swap      free    re  mf  pi  po  fr  de  sr  s0 s1  s6  s8 in   sy  cs us sy id
0 0 0  34020544 31463984  47  22   5   0   0   0   0   0  0   0   0 239  120 36  0 0 100
```

When migrating to a UltraSPARC T1 processor-based system, look at the following fields in mpstat:

- usr/sys/wt/idl (CPU utilization breakdown)

  These fields display how much of each CPU is consumed in mpstat(1). vmstat(1) aggregates utilization for all CPUs and rolls wt and idl into id. If a v490 or x86 system is mostly idle, this will also be true on a UltraSPARC T1 processor-based system. If a traditional SMP system is fully consumed, this should map to an UltraSPARC T1 system. If a two-way system is fully consumed, you need to determine how many LWPs are involved in the processing.

- icsw (number of involuntary context switches)

  This field displays the number of LWPs involuntarily taken off the CPU either through expiration of their quantum or the availability of a higher-priority LWP. This number indicates if there were generally more LWPs ready to run than physical processors. When run on an UltraSPARC T1 processor-based system, the icsw is usually less than it is on traditional systems because there are more hardware threads available for LWPs.

  The r b w fields of the vmstat output give another indication of how many LWPs are ready to run, blocked, or in wait state. If a high number of LWPs can be run, this is a good indication that an application will scale and is potentially a good fit for a thread-rich environment.

- migr (migrations of LWPs between processors)

  This field displays the number of times the OS scheduler moves a ready-to-run LWP to an idle processor. If possible, the OS tries to keep the LWP on the last processor on which it ran. If the processor is busy, it migrates. Migrations on traditional CPUs are bad for performance because they cause an LWP to pull its working set into cold caches, often at the expense of other LWPs.

  UltraSPARC T1 migrations between hardware strands on the same core are essentially free because all caches are shared. Migrations between cores also have less of an impact because most state is held in the unified L2 cache.

- csw (voluntary context switches)

  This field displays the number of LWP switches that occur either because tasks are short or they need to wait for I/O. These should be similar on UltraSPARC T1 processor-based systems.

Assessing Application Parallelism on a Traditional System

Before attempting to tune or optimize an application on a UltraSPARC T1 processor-based system, determine how many LWPs are actually involved in processing.

To make this determination, run the workload and use `prstat`(1) or `ps -ef`(1) to determine which processes are accumulating CPU. It is a good idea to take a few snapshots at intervals to determine which processes are accumulating CPU on an ongoing basis.

The default execution of `prstat`(1) sorts by CPU utilization and gives you the process LWP count, as shown in CODE EXAMPLE 3.

CODE EXAMPLE 3 `prstat`(1) Output—LWP Count

```
 PID  USERNAME  SIZE     RSS   STATE  PRI  NICE    TIME      CPU   PROCESS/NLWP
 8607 mauroj    145M    145M    cpu0   59    4     0:00:54    71%   cpuhog/9
 5148 andreash   35M   9632K   sleep  59    0    0:07:11   0.2%  mixer_applet2/1
 4455 andreash   16M     14M    sleep  59    0     0:02:10   0.1%   Xvnc/1
 4614 andreash   12M     10M    sleep  59    0     0:01:46   0.0%   gconfd-2/1
```

In the preceding example, `cpuhog` (PID 8607) is consuming 71 percent of available CPU and has nine LWPs.

CODE EXAMPLE 4 shows the results of running `ps -ef`.

CODE EXAMPLE 4 `ps -ef` Output—LWP Count

```
 UID     PID      PPID   C    STIME     TTY        TIME CMD
 daemon 100726       1   0    11:45:55  ?          0:00 /usr/lib/nfs/statd
 root   129953 129945   1    15:30:53  pts/5      0:43 /usr/dist/share/
 java,v1.5.0beta2/5.x-sun4/jre/bin/java -server -XX:CompileThres
 root   100742 100739   0    11:47:28  pts/2      0:01 -tcsh
 root   100405       1   0    11:45:41  ?          0:00 /usr/lib/ssh/sshd
 root   100409       1   0    11:45:41  ?          0:02 /usr/sfw/sbin/snmpd
 root   100724 100721   0    11:45:52  ?          0:00 /usr/lib/saf/ttymon
 root   129896 100251   0    15:30:02  ?          0:00 in.rlogind
 root   129945 129934   0    15:30:45  pts/5      0:09 /usr/dist/share/
 java,v1.5.0beta2/5.x-sun4//bin/java -classpath /export/home/XML
 root   129934 129869   0    15:30:36  pts/5      0:00 /bin/sh run_xmltest.sh
 root   129957 129898   0    15:31:08  pts/1      0:00 ps -ef
 root   101738 100251   0    11:51:59  ?          0:00 in.telnetd
```

In the preceding example, two Java processes are active, but PID 129953 is accumulating the greatest number of CPU cycles. If a process is accumulating more CPU time than has elapsed, the process is multithreaded. This result is confirmed by running the `ps -Lef` command as shown in the following example.

```
ps -Lef | grep 129953  | grep -v grep | wc -l
      48
```

As shown in the preceding example, there are 48 threads in the process. A closer look indicates that not all of them are active.

CODE EXAMPLE 5 `ps -Lef` Output—Thread Activity

```
ps -Lef | grep 129953
    UID    PID   PPID   LWP  NLWP   C   STIME TTY      LTIME CMD
   root 129953 129945     1    48   0 15:30:52 pts/5   0:06 /usr/dist/share/java,v1.5.0beta2/5.x-sun4/
jre/bin/java -server -XX:CompileThres
   root 129953 129945     2    48   0 15:30:53 pts/5   0:01 /usr/dist/share/java,v1.5.0beta2/5.x-sun4/
jre/bin/java -server -XX:CompileThres

....

   root 129953 129945    45    48   0 15:30:59 pts/5   6:24 /usr/dist/share/java,v1.5.0beta2/5.x-sun4/
jre/bin/java -server -XX:CompileThres
   root 129953 129945    46    48   0 15:30:59 pts/5   6:26 /usr/dist/share/java,v1.5.0beta2/5.x-sun4/
jre/bin/java -server -XX:CompileThres
   root 129953 129945    47    48   0 15:30:59 pts/5   6:24 /usr/dist/share/java,v1.5.0beta2/5.x-sun4/
jre/bin/java -server -XX:CompileThres
```

Some threads have accumulated six minutes of CPU time, and others have accumulated only a few seconds. From the preceding `ps` output, you can see that about 36 threads are active. If multiple LWPs are accumulating CPU, the application will most likely scale on the multiple cores of an UltraSPARC T1 processor-based system.

When assessing performance, try to scale up the workload on multiple CPUs of a traditional SPARC system. Look for any points in the throughput curve where performance declines. This is an indication of a possible bottleneck.

Even if multiple threads or processes are accumulating CPU, applications can also have phases of execution. Therefore, it is useful to take a number of `prstat`(1) or `ps`(1) snapshots to determine how the utilization changes over time, which will enable you to pinpoint periods of serialization or periods where a master thread feeds consumers.

Assessing CMT System Throughput

Processor utilization from mpstat is the most important performance indicator on a traditional system. When running on an UltraSPARC T1 processor-based system, however, the most important statistic is the number of instructions per second that the chip can execute, which ultimately determines the throughput of the system. The instructions per second of the chip is the sum of the number executed on each strand. The strand count is dependent on a number of different factors, including the following:

- The scalability of the application, for example, the number of LWPs that can be run in parallel
- The application's instruction mix and the amount of stall it generates
- The LWPs executing on the other strands of a core, their stall component, and use of the shared resources (instruction cache, for example)
- The total number of LWPs that the Solaris OS tries to schedule on the strand, which determines how often the application gets to run
- Other applications running on other cores and their use of the L2 unified cache

Each UltraSPARC T1 strand has a set of hardware performance counters that can be monitored using cpustat(1M). cpustat can collect two counters in parallel, the second always being the instruction count. For example, to collect iTLB misses and instruction counts for every strand on the chip, you can type the following:

```
/usr/sbin/cpustat -c pic0=ITLB_miss,pic1=Instr_cnt,sys 1 10
```

The output from the preceding command appears as shown in CODE EXAMPLE 6:

CODE EXAMPLE 6 cpustat(1) Output—iTLB Misses and Instruction Count

```
time   cpu   event    pic0   pic1
2.019   0    tick     6     186595695  # pic0=ITLB_miss,sys,pic1=Instr_cnt,sys
2.089   1    tick     7     192407038  # pic0=ITLB_miss,sys,pic1=Instr_cnt,sys
2.039   2    tick     49    192237411  # pic0=ITLB_miss,sys,pic1=Instr_cnt,sys
2.049   3    tick     15    190609811  # pic0=ITLB_miss,sys,pic1=Instr_cnt,sys
......
```

You must specify both a pic0 and pic1 register. ITLB_miss is used in the preceding example, although in this instance you are only interested in instruction counts.

The performance counters indicate that each strand is executing about 190 million instructions per second. To determine how many instructions are being executed per core, aggregate counts from four strands. Strands 0, 1, 2, and 3 are in the first core, strands 4, 5, 6, and 7 are in the second core; and so on. The preceding example

indicates that the system is executing about 760 million instructions per core per second. If the processor is executing at 1.2 gigahertz, each core can execute a maximum of 1200 million instructions per second, so you have an efficiency rating of .63.

To achieve maximum throughput, you need to maximize the number of instructions per second on each core and ultimately on the chip.

Other useful `cpustat` counters for assessing performance on an UltraSPARC T1 processor-based system are detailed in TABLE 1. All counters are per second, per thread.

Rather than deal with raw misses, you can accumulate the counters and express them as a percentage miss rate of instructions. For example, if the system executes 200 million instructions per second on a strand and `IC_miss` indicates 14 million instruction cache misses per second, then the instruction cache miss rate is 7 percent.

TABLE 1 `cpustat` Output—Performance-Assessment Counters

Counter Name	Counts	High Value	Impact	Potential Remedy
`IC_miss`	Number of instruction cache misses	> 7%	Small impact as latency can be hidden by strands	Compiler flag options to compact the binary. See compiler section.
`DC_miss`	Number of data cache misses	>11%	Small impact as latency can be hidden by strands	Compact data structures to align on 64-byte boundaries.
`ITLB_miss`	Number of instruction TLB misses	>.001%	Potentially severe impact from TLB trashing	Make sure text on large pages. See TLB section.
`DTLB_miss`	Number of data TLB misses	>.005%	Potentially severe impact from TLB trashing	Make sure data segments are on large pages. See TLB section.
`L2_imiss`	Instruction cache misses that also miss L2	>2%	Medium impact potential for all threads to stall	Reduce conflict with data cache misses if possible.
`L2_dmiss_ld`	Data cache misses that also miss L2	>2%	Medium impact potential for all threads to stall	Potential alignment issues. Offset data structures.

With these counters, you can obtain a good overall picture of the applications behavior. cpustat(1) gives statistics for all LWPs running on all cores. To isolate the statistics for a single process, use cputrack(1), as follows

```
cputrack -c pic0=ITLB_miss,pic1=Instr_cnt,sys   command
cputrack -c pic0=ITLB_miss,pic1=Instr_cnt,sys   -p pid
```

cputrack follows the process, taking into account when it is switched off CPU or migrates between strands.

UltraSPARC T1 processors also have a number of DRAM performance counters, the most important of which are read and write operations to each of the four memory banks. The tool to display DRAM counters is busstat (all on one line). Type the command as follows.

```
busstat -w dram0,pic0=mem_reads,pic1=mem_writes
-w dram1,pic0=mem_reads,pic1=mem_writes
-w dram2,pic0=mem_reads,pic1=mem_writes
-w dram3,pic0=mem_reads,pic1=mem_writes
```

CODE EXAMPLE 7 shows the output that results:

CODE EXAMPLE 7 busstat Output—DRAM Performance Counters

```
time   dev     event0            pic0          event1         pic1
1      dram0  mem_reads          16104         mem_writes      8086
1      dram1  mem_reads          15953         mem_writes      8032
1      dram2  mem_reads          15957         mem_writes      8069
1       dram3  mem_reads          15973         mem_writes      8001
```

The counts are of 64-byte lines read or written to memory; to get the total bandwidth, add all four counters together. In the preceding example, the system is roughly reading the following:

(4 * 16000 * 64) = 4096000 bytes / 3.9 megabytes per second

and writing the following:

(4 * 8000 * 64 bytes) = 2048000 bytes / 1.95 megabytes per second

Analyzing Performance With DTrace Software

UltraSPARC T1 systems are supported only on the Solaris 10 OS. One of the most powerful features of the Solaris 10 OS is DTrace, which is used for both debugging and performance analysis. For more information about DTrace software, visit the following web site:

`http://www.sun.com/bigadmin/content/dtrace/`

The *Solaris Dynamic Tracing Guide* is an excellent starting point for information about this tool. In the Solaris 10 OS, `lockstat`(1) has been rewritten and is now a DTrace client, and a new tool called `plockstat`(1) is always built on top of DTrace.

Note – The UltraSPARC T1 Architecture Group has used many scripts in assessing the performance of UltraSPARC T1 processors.

Scaling Applications With CMT

In most cases, applications that already scale on SMP systems should perform extremely well on UltraSPARC T1 processor-based systems. Sometimes, however, software fails to scale. The following sections describe how to determine scaling bottlenecks and some of the options for improving scaling.

Scaling Locks

Hot locks are the most common reason applications fail to scale when deployed on an UltraSPARC T1 processor-based system. Hot locks are more commonly exposed when an application is migrated from a two-way or four-way system, a very common deployment today. Applications might have many threads configured, but on these systems only two to four threads can actually run in parallel, which reduces contention by essentially serializing access to hot locks. When migrating to an UltraSPARC T1 processor-based system, many threads can run in parallel and locks can become hot as a result.

On traditional SMP systems, communication costs between threads are high. Communication costs include compare-and-swap (CAS) instructions, which are usually used for lock primitives, and coherency traffic. The constant intercache movement of locks and critical structures is a major performance inhibitor.

Cache-to-cache transfers cause a lot of stalls in applications, and independent software vendors (ISVs) dedicate a lot of effort to breaking up shared structures and scaling locks. The problem is worse on NUMA systems where transfers from remote caches are more costly. This trend has led the authors of parallel applications to use coarse-grained parallelism and avoid sharing.

In contrast to this, UltraSPARC T1 processor-based systems easily handle fine-grained sharing and communication through the high-bandwidth, low-latency, on-chip crossbar. The L2 is also 12-way associative, which helps reduce both cache conflicts and false sharing between cores. Hot structures and locks tend to reside in the unified L2 cache. The latency to access a lock is therefore a lot less, which can change the scaling characteristics of an application.

Users see the effects of lock contention as follows:

- No increase in throughput as more load is applied, or an actual decrease in throughput as CPU cycles are consumed waiting for resources
- Increasing response time of the application as more load is applied

Observing Hot Application Locks

Many commercial applications have in-built lock observability. Oracle applications, for instance, have the Statspck package with which a user can take multiple snaps of Oracle's internal statistics, including lock contention, hold times, waiters, and the like. A generator produces reports based on the deltas between the various snaps.

It is useful to take lock statistics for stems where the application currently runs and compare them with the statistics from an UltraSPARC T1 processor-based system.

Observing Hot Solaris Userland Locks

You can observe user-level locks in the Solaris OS using `plockstat`(1M). This command uses Dtrace to instrument a running process or a command of interest. To determine hot locks in a command, type the following command:

```
plockstat -A command arg...
```

To attach to a running process, type the following command:

```
plockstat -A -p <pid>
```

This command gives only the caller by default, but you can also use `-s <depth>` to give a stack trace of the contending LWPs.

Observing Hot Java Locks

Locking primitives are handled differently in various versions of the JVM machine. In JVM 1.4.2, a thread handles lock contention by entering the kernel with a system call. Hot locks generate a high number of `lwp_mutex_timedlock`() or `lwp_mutex_lock`() system calls. DTrace or `truss -c` can be used to get counts of system calls to see if these calls dominate. High lock contention in JVM 1.4.2 also results in high system time as displayed by `mpstat`.

If many threads are contending for the same lock, the effect is more pronounced on an UltraSPARC T1 processor-based system (or a 32-way SMP system) because all the hardware strands will enter the kernel. To preserve CPU cycles for other processes, isolate such an application in a processor set to restrict the number of threads contending.

From JVM 1.5 onward, locking has been moved to the VM itself. A back-off mechanism has been implemented for hotly contended locks, which translates to more idle on the system. Therefore, in JVM 1.5, idle might increase as more load is applied.

There is a lot of ongoing work in Java observability through the Java Virtual Machine Profiler Interface (JVMPI) and the JVM Tool Interface (JVMTI). This has been linked with Dtrace, as well. Additional information is available at the following web locations:

`http://blogs.sun.com/roller/page/ahl`

and

`https://solaris10-dtrace-vm-agents.dev.java.net/`

The Workshop Analyzer also uses this framework in its later releases.

These interfaces are available only in later versions of the JVM machine. Often, it is not possible to move to these later versions of the JVM machine. One option is to dump all the stacks on the running VM by sending a SIGQUIT (signal number 3) to the Java process using the `kill` command as follows.

```
kill -3 <pid>
```

This dumps the stacks for all VM threads to the standard error as shown in CODE EXAMPLE 8.

CODE EXAMPLE 8 Thread Dump Message

```
Full thread dump Java HotSpot(TM) Client VM (1.4.1_06-b01 mixed mode):

"Signal Dispatcher" daemon prio=10 tid=0xba6a8 nid=0x7 waiting on condition
[0..0]

"Finalizer" daemon prio=8 tid=0xb48b8 nid=0x4 in Object.wait()
[f2b7f000..f2b7fc24]
        at java.lang.Object.wait(Native Method)
        - waiting on <f2c00490> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:111)
        - locked <f2c00490> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:127)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0xb2f88 nid=0x3 in Object.wait()
[facff000..facffc24]
        at java.lang.Object.wait(Native Method)
        - waiting on <f2c00380> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:426)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:113)
        - locked <f2c00380> (a java.lang.ref.Reference$Lock)

"main" prio=5 tid=0x2c240 nid=0x1 runnable [ffbfe000..ffbfe5fc]
        at testMain.doit2(testMain.java:12)
        at testMain.main(testMain.java:64)

"VM Thread" prio=5 tid=0xb1b30 nid=0x2 runnable

"VM Periodic Task Thread" prio=10 tid=0xb9408 nid=0x5 runnable
"Suspend Checker Thread" prio=10 tid=0xb9d58 nid=0x6 runnable
```

If the top of the stack for a number of threads terminates in a mutex call, this is the place to drill down and determine what resource is being contended.

Sometimes removing a lock that protects a hot structure can require many architectural changes that are not possible. The lock might even be in a third-party library over which the developer has no control. In such cases, multiple instances are probably the best way to achieve scaling.

Tuning Locks

Locks are usually implemented as spin-mutexes. When a thread wishes to acquire a lock, it first tests (usually through CAS) whether the lock is unable to acquire the thread spins. If the lock fails to acquire them, we might choose to spin to avoid being context switched off CPU. Context switches are an expensive operation; they involve the immediate cost of getting another thread on CPU, as well as the added cost of TLB and cache pollution (the so-called cache reload transient).

On a classic SMP system, we spin to try to increase the utilization of the processor in question; we burn cycles hoping that the lock will soon become available. On a UltraSPARC T1 processor-based system, however, spinning steals cycles from other strands on the core. This has to be weighed against the disruption that a context switch will have on the other strands in a core.

Another reason for spinning on SMP systems is to avoid going off CPU and spending a long time on a Solaris run queue. In a CMT environment, there is a much higher likelihood that a strand will be available when the application is ready to run, which leads to shorter queues.

Testing the spin counts for locks is an iterative process on UltraSPARC T1 processors. The current setting is a good starting point and is sufficient in most cases. If throughput is low, but the number of instruction per thread is relatively high, the application could be doing excessive spinning. Also check any application lock hold and spin statistics.

When decreasing the spin count, look for changes in the involuntary context switch rates displayed by `mpstat`.

In multithreaded applications, consider making locks adaptive by initializing them with `PTHREAD_MUTEX_PRIVATE` (for pthreads) or `USYNC_THREADS` (for UI threads). In a thread-rich environment, there is a higher likelihood the holder will be on a CPU.

On typical SMP systems, spin algorithms often use long latency instructions between checks of the mutex. Sometimes, floating-point or long-integer instructions such as `div`, `mul`, or `fdiv` are used. This reduces coherency traffic caused by the lock check. However, on UltraSPARC T1 processor-based systems, coherency is not an issue and we can check the mutex more frequently. Also, with the number of threads, there is a higher likelihood that the lock holder will be on CPU, so the lock change will be seen more quickly.

Thundering herds can also occur on UltraSPARC T1 processor-based systems because 32 threads are ready to run at any given time. When the holder of a hot lock relinquishes control, there might be a number of other threads spinning, waiting to acquire the lock. Because the lock is held in unified L2 (it was just freed), the waiters all get to see it at the same time, which can lead to unforeseen deadlock situations

that are often resolved using an unoptimized back-off codepath. If all the waiters are backed off for a similar amount of time, they can all return to looking for the lock as a herd. This causes periods of reduced throughput on the system.

It might be necessary to break up the lock or assign random back-off periods to avoid these situations.

Tuning Critical Sections

On UltraSPARC T1 processor-based systems, a thread might take longer to pass through a critical section than it would on an SMP system. This is especially true if there are other active threads on the core sharing cycles. Critical sections are often protected using locks, so users might see longer hold times in the application's lock statistics. Tuning might require scaling these locks.

Performance gains can also be achieved by shortening the length of each section, which can be achieved as follows:

- Breaking up the structure and creating a number of locks for each of the sections
- Creating per-CPU structures, which then scale the associated locks
- In multithreaded environments, reducing the amount of global data. With the introduction of thread local storage (TLS), Solaris threads can have high-performing per-thread data.

Where possible, Java application writers are advised to use the `java.util.concurrent` facility for threading and synchronization. Many of the data structures provided by that package are lock-free and scale well on large systems like those using UltraSPARC T1 processors. For more information about this, visit the following link:

`http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html`

Another mechanism for enhancing the performance of critical sections is the `schedctl` API. For more information, see the `schedctl_init`(3SCHED) man page. These APIs enable a running LWP to give a hint to the kernel that preemptions of that LWP should be avoided. Calls to `schedctl_start`() and `schedctl_stop`() are placed around critical sections to indicate the LWP is holding a critical resource.

Sizing Processes and Threads

A central concept in CMT design is that threads are plentiful but less powerful. This is referred to as a thread-rich environment. The best way to achieve scalability in this environment is to apply additional threads to a given task. In a thread-rich environment, an LWP is more likely to be scheduled and running instead of waiting in the OS run queue.

Less contention to get on CPU significantly reduces the impact of the context switches that occur as a result of preemption and are a significant performance hit on traditional CPUs. Run queues also tend to be shorter in a thread-rich environment, and a process spends less time waiting to run than it would on an SMP system.

In multiprocess environments, increasing the number of application processes might be necessary to utilize the strands. For example, an Oracle application might need more shadow processes, or SAP might need more worker processes (WP) in each instance. When multiple LWPs are sharing a core, the response time of each might be slightly slower on CMT. In online transaction processing (OLTP) environments, however, response time is usually dominated by network and disk delays.

In multithreaded applications, you might need to resize thread pools. Such pools are often sized to the number of processors, which equates to a hardware strand in CMT. Sizing the pools can be an iterative process. Use `ps -Lef` or `prstat` to determine how many Solaris threads are required for a load.

Sometimes, increasing pool sizes might not be sufficient due to bottlenecks encountered in handoffs between threads. This problem can manifest itself as idle on the system even as the load applied is increased. Developers need to look at points of serialization in the application.

Sizing Java Garbage Collection

When running Java applications, you might need to scale back the number of garbage collection (GC) threads. This obviously depends heavily on the amount of garbage being generated. Too many GC threads can affect the throughput of the system.

To determine how many GC threads are being executed and their duration, add the following flags to the application's regular JVM command line:

```
-verbosegc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails
```

For example, add the flags as follows:

```
$JAVA_HOME/bin/java -server -Xbatch -Xms3800m -Xmx3800m -XX:+AggressiveHeap -
Xss128k -verbosegc spec.jbb.JBBmain -propfile SPECjbb.props
    [java] 4.865: [GC [PSYoungGen: 732160K->8192K(842240K)] 732160K-
>8192K(2443776K), 0.1209852 secs]
     [java] 8.567: [GC [PSYoungGen: 730112K->7168K(842240K)] 730112K-
>7168K(2443776K), 0.0202491 secs]
     [java] 11.897: [GC [PSYoungGen: 729088K->7168K(842240K)] 729088K-
>7168K(2443776K), 0.0166241 secs]
```

The preceding example indicates that at 4.86 seconds, there was a Young Generation GC that cleaned 2.4 gigabytes of Java heap and took 0.12 seconds.

The default number of parallel GC threads is the number of “CPUs” presented by the Solaris OS. On UltraSPARC T1 processor-based systems, this equates to the number of hardware threads. To set the number of threads, set the following flag:

```
XX:ParallelGCThreads=<number_of_threads>
```

A good starting point is to set the GC threads to the number of cores on the system running UltraSPARC T1 processors (for example, eight as shown in the following example).

```
$JAVA_HOME/bin/java -server -Xbatch -Xms3800m -Xmx3800m -XX:+AggressiveHeap -
Xss128k -XX:ParallelGCThreads=8 -verbosegc  spec.jbb.JBBmain -propfile
SPECjbb.props
```

Older versions of the JVM machine, such as 1.3, do not have parallel GC. This can be an issue on CMT processors because GC can stall the entire VM. Parallel GC is available from 1.4.2 onward, so this is a good starting point for Java applications on UltraSPARC T1 processor-based systems.

Polling, Events, and Async Operation

Polling is often used in applications to ensure response time requirements. An LWP is dedicated to constantly checking for incoming requests, responses, and the like. On a traditional CPU, polling is an acceptable option because there is only a single thread of execution for the scheduler to manage. Other LWPs in the dispatch queue that require CPU cycles will prevent the polling thread from burning CPU.

In a CMT architecture, polling is not a good option because it burns CPU cycles that could be used by other threads on a core. In a thread-rich environment, the polling thread will run far more often than it would on an SMP system.

A useful alternative is event ports which are available in the Solaris 10 OS. (Reference port_create(3C), port_get(3C), and port_send(3C) for details.) These APIs enable a thread to receive notification when an event is complete. A thread will be automatically scheduled when the event is ready at a port. A single thread can listen on multiple ports, or multiple threads can service a high-throughput port.

http://developers.sun.com/solaris/articles/event_completion.html

Similarly, kernel asynchronous I/O (kaio) can be used to avoid burning cycles. (Reference aio_read(3RT), aio_write(3RT), aio_waitn(3RT), and aio_suspend(3RT) for details.) Lists of I/O can be submitted with lio_listio(3RT). Rather than blocking, I/Os can be reaped asynchronously.

Improving the Number of Completed Interrupts per Second

As mentioned previously, a single CMT strand is not as powerful as a traditional SPARC CPU. Usually, when completing an interrupt, a packet must be delivered from the hardware to the user's data space requiring a certain number of instructions. This limits the number of interrupts per second a CMT strand can deliver. There are an abundance of strands in the chip, however, so a useful technique for improving the number of interrupts per second is fanout using either hardware or software.

A software technique involves minimal packet processing in the interrupt context of a driver. The driver simply deposits the packet on one of a number of queues for asynchronous processing. A kernel thread is then scheduled on any other strand to complete the interrupt. This greatly improves the interrupts per second that can be completed.

A good example is the GigaSwift Ethernet [Cassini (ce)] driver on SPARC systems. This driver has four task queues for processing packets. This driver performs very well on UltraSPARC T1 processor-based systems.

Fanout can also be achieved using hardware techniques, such as message signaled interrupts (MSIs), which are supported in PCI-Express and optional in PCI 2.2 devices. In the Solaris 10 12/05 OS, there is a DDI interface for MSI programmers. MSI lets interrupts be delivered to a range of CPUs or hardware threads in the case of UltraSPARC T1 processors. This greatly increases the throughput of a driver in a CMT system.

Tuning for General Performance

Improving the general performance of an UltraSPARC T1 processor-based system involves various system-tuning practices including properly sizing memory, selecting the most suitable JVM machine, scheduling strands, and the like. This section describes some of the options you have for tuning a system that uses UltraSPARC T1 processors.

Sizing Memory

UltraSPARC T1 processor-based systems ship with up to 32 gigabytes of DDR2 memory. Configurations of 4 gigabytes, 8 gigabytes, and 16 gigabytes are also available. The amount of memory has a large effect on both the price of the system and the power consumed, which is a great concern in most data centers today. Thus, correct memory sizing is essential for both price performance and performance per watt.

Today, the majority of applications are 32-bit, which limits their address space to 4 gigabytes, which, in turn, limits the amount of memory they consume. Most applications do not require more memory than this, although the trend is moving toward higher memory space. When moving to an UltraSPARC T1 processor-based system (or any SMP-based system), memory scaling issues are sometimes uncovered. These are often not seen when running on a two-way server limited to 4 gigabytes or 8 gigabytes of memory.

When sizing the memory on an UltraSPARC T1 processor-based system, consider the following questions:

- How much memory can an application really take advantage of?
- If multiple instances are used to increase scalability, how much memory does each instance consume?
- Can performance be gained from a larger file system cache?
- Can swapping or paging be avoided with more memory?

Memory sizing is often an iterative process. If possible, test with 16 gigabytes or even 32 gigabytes initially, and use `vmstat` while running the workload to see if memory can be reduced, as follows.

```
vmstat 5
kthr      memory                          page               disk           faults        cpu
r b w   swap       free    re   mf pi po fr de sr s0 s1 -- --   in   sy   cs     us sy id
 0 0 0 14890592 14773440 9   76   0   0  0  0  0  0  0  0  0  384  230  226      8  0  92
 0 0 0 15155536 14672920 1   10   8   0  0  0  0  0  0  0  0  276   65   52      0  0 100
```

In the preceding example, the free memory column indicates that 14 gigabytes are free on a 16-gigabyte system. It is important to have enough memory to avoid paging during high loads, which will seriously impact performance. Use the `-p` option as follows to determine which memory is being paged.

```
vmstat -p 10
memory                        page        executable        anonymous       filesystem
swap         free      re  mf  fr  de  sr  epi  epo  epf   api  apo  apf   fpi  fpo  fpf
14890616   14773432    9  76   0   0   0    0    0    0     0    0    0     0    0    0
```

The following list outlines the steps involved in sizing memory:

1. Run the application with 32 gigabytes and use `vmstat` to determine if less memory is needed. Determine the performance at this memory level.
2. Reduce the memory to 16 gigabytes and rerun the test. Determine if there is a performance impact.
3. Again, use `vmstat` to determine if there is memory available.
4. Reduce the memory to 8 gigabytes and repeat the test.

Many applications scale structures and threads based on the number of CPUs in the system, often determined by the `sysconf`(3C) call. Because the Solaris OS reports 32 processors for UltraSPARC T1 processor-based systems, these allocations might be too high. Application developers might need to tune down the allocation of resources as required.

Because more memory is available on an UltraSPARC T1 processor-based system, you can consider moving an application to 64-bit, thus removing the 4-gigabyte restriction. There are already 64-bit versions available of many applications such as databases and the JVM machine. When considering a move to 64-bit, the following considerations are important:

- Can the application take advantage of greater than 4 gigabytes of memory?
- Are 64-bit versions of all third-party libraries available for the application? A single process cannot mix 32-bit and 64-bit libraries.

- There is a performance cost in running a 64-bit application on any platform due to the extra instructions required to break the 4-gigabyte limit. This cost can be up to 12 percent.
- There will be an effort involved in porting from 32-bit to 64-bit and also testing the 64-bit version.

Use `pmap`(1) with the `-x` option on the process to determine the amount of resident physical memory being consumed. Shared mappings, such as shared object libraries, will not need to be added for every process. Physical address space tends to be dominated by process heap segments.

As a simple rule of thumb, if an application is already hitting the 4-gigabyte memory limit and the potential gain in performance is greater than a 12 percent hit, then a 64-bit version of the application should be considered.

Selecting a JVM Machine

A number of optimizations have been added to the Sun JVM for CMT and for UltraSPARC T1 processors in particular. The first JVM machine Sun will release with these optimizations will be Java™ 2 Platform, Standard Edition (J2SE™) 5.0_06 software, which can be downloaded from `java.sun.com`

The optimizations are also available in the Java™ Developers Kit (JDK™) 6.0 Binary Snapshot Releases available at:

`www.java.net/download/jdk6/binaries`

Developers can try them here first.

Customers are often reluctant or unable to move to the latest release of the JVM machine. Make every effort to move to at least version 1.4.2, which has parallel garbage collection and the `AggressiveHeap` option.

Compiling Applications

UltraSPARC T1 processors are fully SPARC-binary compatible for all architectures that Sun supports (for example, SPARC v7, v8, and v9). An application does not need to be recompiled or modified in any way to run on systems that use UltraSPARC T1 processors, and we do not expect ISVs to support a UltraSPARC T1 processor-specific binary. Binaries created by older versions of the Sun™ Studio software compiler will also work well.

All the current compiler optimizations will also work well on UltraSPARC T1 processor-based systems. In general, for all SPARC platforms we recommend using profile-feedback with `-xprofile=collect` and then `-xprofile=use`. This can improve performance up to 15 percent on most applications.

Profile feedback is described in the PDF file at the following location:

`http://developers.sun.com/solaris/articles/sol-perf/sol-app-perf.pdf`

The *Sun Studio 10 C User's Guide* from `docs.sun.com` also contains information about this topic.

UltraSPARC T1 processors have a relatively small instruction cache per core, and optimizations that compact the executable are worth investigating. Using postoptimization through the `-xlinkopt` option is the most effective mechanism to achieve this if feedback optimization is already being used. Information about this can be found at the following site:

`http://docs.sun.com/source/819-0494/cc_ops.app.html`

A mapfile is another alternative for optimizing applications. This is generated using the Sun Studio Workshop Analyzer. See the Analyzer documentation for details.

TABLE 2 provides recommendations for developers who decide to recompile an application for use on UltraSPARC T1 processor-based systems.

TABLE 2 Recompilation Recommendations

Binary type	Compiler	Flags

32-bit	Sun Studio 7, 8, 9, and 10 software	`-xO4`
		or
		`-fast -xO4 -xtarget=generic`

32-bit	Sun Studio 10, Update 1 software	`-xO4 -xtarget=ultraT1`
		or
		`-fast -xO4 -xtarget=ultraT1`

64-bit	Sun Studio 7, 8, 9, and 10 software	`-xO4 -xtarget=generic64`
		or
		`-fast -xO4 -xtarget=generic64`

64-bit	Studio 10, Update 1 software	`-xO4 -xtarget=ultraT1 -xarch=generic64`
		or
		`-fast -xO4 -xtarget=ultraT1 -xarch=generic64`

Note – If building a shared object, use `-KPIC` or `-Kpic`.

Scheduling LWPs

The Solaris OS handles the normal scheduling of LWPs on hardware strands. It also attempts to maintain the affinity of LWPs for their strands. If there is insufficient work, a strand enters the idle loop in the kernel, at which point the hardware strand is parked so its slot can be used by the other strands in the core. When work becomes available, the strand is automatically unparked.

A side effect of Solaris thread parking is that it blurs the concept of idle on a hardware strand. In a traditional SPARC core, the CPU can be in one of three states (user, system, or idle) as reported by `mpstat` or `vmstat`. A hardware strand in a UltraSPARC T1 processor-based system can be user, system, or parked.

If a strand is parked, its cycles are automatically distributed to the other strands in the core. Therefore, the shared resources of a core are not truly idle unless all the strands are idle. Idle, therefore, has meaning only at a core level, although `mpstat` and `vmstat` still reports user, system, or idle for each strand.

Having some idle strands in a core is inefficient, however, because it reduces the ability to absorb stall. A CMT processor's strength is in running many threads in parallel.

Partitioning Applications

With the abundance of threads, developers can also consider dedicating strands or even cores to tasks using `psrset`, `pbind`, or the equivalent APIs. Keep in mind that four strands share the facilities of a core. From a performance perspective, sub-core processor sets might not be effective because strands in multiple sets share the core resources, caches, pipeline, TLBs, and the like.

Strands on an eight-core UltraSPARC T1 processor-based system are numbered 0–31. Strands 0, 1, 2, and 3 are in core 0; strands 4, 5, 6, 7 are in core 1, and so forth. To create a processor set on the first core, use the following command:

```
psrset -c 0 1 2 3
created processor set 1
processor 0: was not assigned, now 1
processor 1: was not assigned, now 1
processor 2: was not assigned, now 1
processor 3: was not assigned, now 1
```

To bind a process into a processor set, use the following command:

```
psrset -b 1 2870
process id 2870: was not bound, now 1
```

Alternatively, you can bind the shell to a processor set and then launch the application. All subsequent processes will also reside in the processor set.

In the Solaris 10 OS, pbind and psrset have been extended to enable the binding of individual LWPs within a process. For example, the following command places LWP 1 from process 2870 into processor set 1:

```
psrset -b 1 2870/1
```

Note – When binding individual LWPs, be careful to ensure that the correct one is chosen. Use prstat or ps to determine which LWPs are accumulating CPU cycles.

Managing Real-Time Processes in a CMT Environment

Strand sharing cores has implications for real-time (RT) processing. If a process is running in the RT scheduler class on a traditional SPARC system, the Solaris OS gives it exclusive access to the processor whenever it is ready to run. In UltraSPARC T1 processor-based systems, a process gets exclusive access only to a strand. The other strands of the core continue to get access to the pipeline unless they are stalled or disabled. Dedicating a single core to RT processes might be necessary to achieve required response times.

You can use the following alternative methods:

- Place the RT process in its own single-core processor set. The Solaris OS parks all the idle threads in the set, leaving the RT exclusive access.
- pbind the RT process to a hardware strand, and use psradm to turn off the other strands in that core.

Consolidating Applications

The 8 cores and 32 threads of UltraSPARC T1 processor-based systems make it a good choice for application consolidation. Consolidation falls into two broad categories, horizontal and vertical.

Horizontal consolidation replaces several small servers running applications with a single UltraSPARC T1 processor-based system. All of the applications might be the same, such as J2EE application servers, or they might be a mixture of different ones.

Some customers have many applications running on older UltraSPARC II servers, which can easily migrate this way. The potential savings in power, cooling, space, and maintenance costs are huge. One disadvantage is that a system outage affects more applications, a problem that can be mitigated by using hot standby systems or redundancy in the application.

Vertical scaling is the collapsing of multiple tiers of the same application onto one UltraSPARC T1 processor-based system. Examples include running SAP and its back-end database (for example, an Oracle database) on the same system. This has the advantage of avoiding network delays and bandwidth limitations between the various tiers.

There are three main ways to approach application consolidation:

- Dedicate one or more cores per application using either processor sets or the Solaris Zones feature (described later). This has the advantage of isolating performance interference from other applications. It can also provide better Quality of Service (QoS) for users. The disadvantage of this approach is that some strands and cores might not be fully utilized while others are resource constrained.
- Let the applications float free across the cores. The advantage of this approach is that strands and cores will be more fully utilized. Potentially more applications can be accommodated on a single UltraSPARC T1 processor. The disadvantage of this approach is that CPU-hungry applications might dominate, starving out other processes. Such processes will sometimes perform better at the expense of others. QOS might, therefore, be harder to achieve.
- The fair share scheduling (FSS) class can be used in the unbound case, providing a means of allocating shares of available CPU to meet required service levels.

A good approach is to test consolidation one application at a time. Run the initial application on the system and determine how many cores it requires. It might be necessary to add some headroom for high load periods. When the number of cores is determined, place the application in its own processor set or zone.

Repeat the process for subsequent applications, giving each a processor set. If the number of applications has been determined but more throughput is required, look at removing the processor sets to let the applications run free.

One issue with application consolidation is the unified L2 cache which is shared by all applications even when partitioned using processor sets or zones. The L2 cache is implemented as a 12-way associative, however, which dramatically reduces the conflicts. Strand-switching technology also hides the stall from extra conflicts.

We have tested a number of application consolidations, and the performance scaled well. To monitor the L2 cache, use the `cpustat` counters `L2_imiss` and `L2_dmiss_ld` as shown in the following example:

```
/usr/sbin/cpustat -c pic0=L2_imiss,sys,pic1=Instr_cnt,sys -c
pic0=L2_dmiss_ld,sys,pic1=Instr_cnt,sys 1 300
```

As shown in CODE EXAMPLE 9, the output is on a per-strand basis. If processor sets are used, the application with the most misses can be determined. It is useful to check the counters as each new application is added.

CODE EXAMPLE 9 `cpustat` Output—Level 2 Cache Performance

```
Time    cpu    event      pic0     pic1
1.002   0      tick       475   1683777  # pic0=L2_imiss,sys,pic1=Instr_cnt,sys
1.012   1      tick        42    109825  # pic0=L2_imiss,sys,pic1=Instr_cnt,sys
2.002   0      tick      1208    848362  # pic0=L2_dmiss,sys,pic1=Instr_cnt,sys
2.012   1      tick      2812    374964  # pic0=L2_dmiss,sys,pic1=Instr_cnt,sys
```

For applications that scale to only a small number of CPUs on current SMP hardware, consider using multiple instances of the application on an UltraSPARC T1 processor-based system. This might put extra load on the L2 cache, so reassess the performance as each instance is added using the L2 performance counters.

Using the Solaris Zones Feature to Isolate Workloads

Processor sets and binding are useful when consolidating workloads on an UltraSPARC T1 processor-based system. In some cases, for example, those listed below, they cannot be used.

- For security reasons, it might not be possible for applications to share the same system.
- Each application might need to bind to the same network port.
- Processor sets are nonpersistent across reboots.

Using the Solaris Zones feature, available in the Solaris 10 OS, is an excellent mechanism for isolating these instances, if required. Zones are secure, lightweight, virtualized OS services that look like different Solaris instances. For more information, visit the following web site:

`http://www.sun.com/bigadmin/content/zones/`

Zones have the ability to present the same set of I/O interfaces to each instance of the application. Hundreds of zones can be created on a system with about only a 1 percent performance impact per zone. Using Solaris pools, we can also assign a number of processors to a zone.

Two of the main uses of zones are group isolation and performance scaling. In *group isolation*, zones are used to isolate groups of users from each other. UltraSPARC T1 processor-based systems can easily support many zones in this scenario. Performance is not a primary concern in this case, rather the focus is on the flexibility of presenting multiple virtual hosts to users. It is probably best in this case not to bind processors to zones. The UltraSPARC T1 processor's 32 hardware threads are used to schedule all zones.

In *performance scaling*, zones are used to run multiple instances of an application to increase throughput. For performance scaling, a single core is probably the finest granularity that is practical in a zone. Often, the best test is to add cores to a zone one at a time until the application no longer scales. Then create multiple zones of this size. Because each application is different, this sizing exercise is necessary.

The Solaris OS has a global zone by default. Other zones are created from the global zone. Configuration and booting is easy, and the user can then log in and install the application, as shown in the following example:

```
# mkdir /zone
# mkdir /zone/1
# zonecfg -z zone1
zone1: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:zone1> create
zonecfg:zone1> set zonepath=/zone/1
zonecfg:zone1> set autoboot=true
zonecfg:zone1> add net
zonecfg:zone1:net> set address=10.1.6.210
zonecfg:zone1:net> set physical=ipge0
zonecfg:zone1:net> end
zonecfg:zone1> info
zonepath: /zone/1
autoboot: true
pool:
inherit-pkg-dir:
        dir: /lib
inherit-pkg-dir:
        dir: /platform
inherit-pkg-dir:
        dir: /sbin
inherit-pkg-dir:
        dir: /usr
net:
        address: 10.1.6.210
        physical: ipge0
zonecfg:zone1> verify
zonecfg:zone1> commit
# zoneadm list -vc
  ID NAME             STATUS         PATH
   0 global           running        /
zone1            configured     /zone/1
# zoneadm -z zone1 install
# zoneadm -z zone1 boot
#
# zoneadm list -vc
  ID NAME             STATUS         PATH
   0 global           running        /
   1 zone1            running        /zone/1
```

If the network traffic to the zone is expected to be high, consider assigning a different network device to each zone. In the preceding example, we assigned ipge0 to zone1. We could assign ipge1 to zone2, and so forth.

To bind hardware threads to zones, use the pool commands as follows:

```
pooladm -e
poolcfg -c discover
poolcfg -c "create pool <poolname>"
poolcfg -c info
poolcfg -c "create pset <pset_name> (uint pset.min = $ncpus; uint
pset.max = $ncpus)"
poolcfg -c info
poolcfg -c "associate pool <poolname> (pset <pset_name> )"
poolcfg -c info
pooladm -c
poolbind -p <pool_name> -i zoneid <zone_name>
```

When logged in to the zone itself, mpstat and vmstat now see only the CPUs that have been assigned to them. ps(1) and prstat(1) only show the processes running in the zone you are logged into. You can use prstat(1) with the '-Z' flag from the global zone to monitor utilization across all zones.

```
# rsh zone1
# psrinfo
0      no-intr   since 02/21/2000 22:31:02
1      no-intr   since 02/21/2000 22:31:02
2      no-intr   since 02/21/2000 22:31:02
3      no-intr   since 02/21/2000 22:31:02
```

With these simple tools you can securely consolidate a number of applications on a single UltraSPARC T1 processor.

Tuning Interrupt Distributions

An important part of tuning a CMT system is interrupt distribution. The issues are similar to those on large SMP systems. Most hardware external devices, such as NICs and storage devices, generate interrupts that are delivered to the kernel. During a boot of the Solaris OS, each interrupt source is assigned a CPU to process its interrupts. The CPU is in effect bound to an interrupt source. With UltraSPARC T1 processor-based systems, the Solaris OS binds interrupts to a hardware strand.

To determine the source of interrupts, use `/usr/sbin/intrstat`. See the `intrstat`(1M) man page for more information.

```
device |       cpu0 %tim      cpu1 %tim      cpu2 %tim      cpu3 %tim
-------+-------------------------------------------------------------
ipge#0 |         0   0.0         0   0.0         0   0.0         0   0.0
ipge#3 |         0   0.0         0   0.0         0   0.0         0   0.0
 mpt#0 |         0   0.0         0   0.0         0   0.0         0   0.0
```

The following example shows how many interrupts each CPU is taking for a device. This can be correlated with `mpstat`.

```
CPU minf mjf xcal intr ithr csw icsw migr smtx srw syscl usr sys wt idl
 0     0   0  747   776  140 793   31  115   88   8  2378  26  4   0  70
.....
14     0   0 1166  1214 1005 446   11   32   46   8  1174  14  2   0  84
15     0   0  653   843  687 332   10   29   41   4   989  12  2   0  87
16     0   0  658   861  700 334    9   23   37   3   922  10  2   0  88
17     0   0  641   859  707 309    6   17   42   5   945  11  2   0  88
```

For CPUs taking interrupts, there will be a higher count in the `intr` column of the output. Keep the following issues in mind when taking interrupts on a CMT processor:

- A single CMT thread cannot sustain the same peak interrupt throughput as a regular core.
- Interrupts are very disruptive to other LWPs running on other strands of a core. They run in kernel context and tend to trash both caches and TLBs.
- Performance can often be gained by binding similar interrupt sources to different hardware strands on the same core. The interrupts will tend to have instruction-cache warmth and often data-cache warmth.

If there are a high number of interrupts on an UltraSPARC T1 processor-based system, it often makes sense to dedicate a core to them. This can be achieved using the psradm command. First, check the state of all the hardware threads in the system using psrinfo, as follows:

```
psrinfo
0       on-line   since 04/06/2005 20:39:36
1       on-line   since 04/06/2005 20:39:37
2       on-line   since 04/06/2005 20:39:37
.....
27      on-line   since 04/06/2005 20:39:37
28      on-line   since 04/06/2005 20:39:37
29      on-line   since 04/06/2005 20:39:37
30      on-line   since 04/06/2005 20:39:37
31      on-line   since 04/06/2005 20:39:37
```

If a strand can take interrupts, it displays on-line in the second column. To disable interrupts on a CPU, you can use psradm as follows:

```
psradm -i <cpuid>
```

After using psradm as described, the CPU can still run both userland and kernel LWPs, it is just not a candidate for interrupt processing. You'll also notice that when you now run psrinfo, its state changes to no-intr, as follows:

```
psrinfo
0       no-intr   since 02/25/2000 23:33:45
1       no-intr   since 02/25/2000 23:33:45
2       no-intr   since 02/25/2000 23:33:45
```

Unfortunately, when psradm -i is applied to CPUs, the Solaris OS redistributes the interrupts across the remaining hardware strands. On a system that uses UltraSPARC T1 processors, the best approach is to dedicate cores to interrupts if required and to disable the interrupts on all other cores with psradm. If using zones, consider moving the interrupts to the global zone so they do not interfere with local zones.

Each time you use psradm(1) to change the interrupt state, use intrstat(1) and mpstat(1) to determine which strands are currently taking interrupts.

Tuning Interprocess Communications

Because of the unified L2 cache between cores, shared memory and other IPC mechanisms such as message queues will be faster on UltraSPARC T1 processor-based systems. This can have implications for timing in applications. Codes that are tuned for the latency of responses on traditional SMP machines might exhibit different behavior if they are moved to a CMT environment. Messages might be delivered quicker and, because of the abundance of threads, the message receiver might already be on CPU. Tuning might require increasing the number of receive threads.

Tuning a TLB

A translation-lookaside buffer (TLB) is a hardware cache that is used to translate a process's virtual address to a system's physical memory address. As mentioned previously, UltraSPARC T1 processor-based systems have a 64-entry iTLB and a 64-entry dTLB per core. These TLBs are fully associative, with all 64 entries looked up in parallel.

The unit of translation is page size. Each entry in an UltraSPARC T1 TLB can support four page sizes—8 kilobytes, 64 kilobytes, 4 megabytes, and 256 megabytes. When memory is accessed and a mapping is not in the TLB, this is termed a *TLB miss*. There are software handlers in the OS to trap a TLB miss and load the appropriate mapping into one of the TLB entries.

The size of the pages in the TLB determine the "span" of memory it can translate. The bigger the page size, the better the span. The default page size for the Solaris OS is 8 kilobytes. Thus, with a 64-entry TLB, the span is only 512 kilobytes. For large data sets, this is often too small, and a large number of TLB misses can result.

Excessive TLB misses are bad for performance. To determine the number of misses per second, you can use `cpustat` or `cputrack` as follows:

```
/usr/sbin/cpustat -c pic0=DTLB_miss,pic1=Instr_cnt,sys -c
pic0=ITLB_miss,pic1=Instr_cnt,sys  1 10
   time cpu event      pic0      pic1
  5.021   0  tick     60620   231646080  # pic0=DTLB_miss,sys,pic1=Instr_cnt,sys
  5.031   1  tick     40386   235915027  # pic0=DTLB_miss,sys,pic1=Instr_cnt,sys
  5.061   0  tick     36884   236575873  # pic0=ITLB_miss,sys,pic1=Instr_cnt,sys
  5.061   1  tick     45842   230867805  # pic0=ITLB_miss,sys,pic1=Instr_cnt,sys
```

Because each TLB miss requires a number of instructions to complete, reducing the misses generally helps performance.

To determine the page size assigned to various segments of a process use `pmap -xs` as follows. (Note that the output is abridged.)

```
pmap -xs 14148

14148:  /usr/sap/SSO/SYS/exe/run/saposcol
Address           Kbytes       RSS        Anon      Locked Pgsz Mode   Mapped File
0000000100000000  320          320           -           -  64K r-x-- saposcol
0000000100050000  128            -           -           -    - r-x--
00000001001A6000    8            8           8           -   8K rwx-- [ anon ]
00000001001B0000  576            -           -           -    - rwx-- saposcol
0000000100240000  576          576         576           -  64K rwx-- [ heap ]
0000010000000000 4096         4096           -        4096   4M rwxsR    [ ism
shmid=0x2a00003f ]

FFFFFFFF7DD00000   64           64           -           -  64K r-x—libc.so.1

FFFFFFFF7FFFC000   16           16          16           -   8K rw--- [ stack ]
---------------- ---------- ---------- ---------- ----------
      total Kb   9104         7992        1072        4096
```

In the preceding example, the executable text of the process `saposcol` is on 64-kilobyte pages, as is the process heap. There is a shared (ISM) segment on 4-megabyte pages. The stack for the process is on 8 kilobytes.

Reducing TLB Misses

To reduce TLB misses, use large pages in a process. Different sections of the process require different options to achieve large pages.

For many years, the Solaris OS has had a performance optimization called Intimate Shared Memory (ISM), which allocates on large pages where possible and locks down memory. The default on UltraSPARC T1 processor-based systems is 256-megabyte pages. ISM was further enhanced to enable its dynamic reconfiguration, called Dynamic ISM (DISM). (Reference `shmat`(2) for more details.)

For ISM, add `SHM_SHARE_MMU` to the shared memory attach flags. DISM is achieved by adding `SHM_PAGEABLE` to the attach flags. Note that DISM requires backing swap space, unlike ISM.

The Solaris OS also attempts to put the executable segments on large pages. This helps to greatly reduce iTLB misses.

The stack, heap, and anon segments of a process can utilize large pages through the Multiple Page Size Support (MPSS) feature of the Solaris OS. For more information, see the mpss.so.1(1) man page. MPSS provides a library, mpss.so.1, that can be preloaded into a process using the following command:

```
export LD_PRELOAD=mpss.so.1
```

There are also a number of environment variables to control allocation. To put the heap or stack on large pages, use the following variables:

```
MPSSHEAP=<size>
MPSSSTACK=<size>
```

For each of these variables, <size> is 8 kilobytes (default), 64 kilobytes, 4 megabytes, or 256 megabytes.

Placing the heap on large pages nearly always improves performance on UltraSPARC T1 processor-based systems. Placing the stack on large pages has not shown major performance gains. Application stack usage varies greatly, however, so it is worth testing this option.

From Solaris 10 12/05 OS onward, the Solaris OS will attempt to place the heap and anon on large pages as well. Using pmap, ensure that the pages were actually allocated. The Solaris OS applies a set of heuristics for large page allocation, but for a particular application a different size might be optimal. In these cases, MPSS can be used to override the Solaris OS defaults.

For more information on MPSS and TLB tuning, reference “Supporting Multiple Page Sizes in the Solaris Operating System” by Richard McDougall.

http://www.sun.com/blueprints/browsesubject.html#performance

Placing Java Heap on Large Pages

Another important optimization is using large pages for the Java heap, which is put on anon segments. From JVM 1.4.2 software onward, the -XX:+AggressiveHeap option has been available. One of the effects of this option is to put the Java heap on 4-megabyte pages. To achieve maximum performance, it is best to preallocate the heap using the options -Xmx<size>m and -Xms<size>m, as follows:

```
$JAVA_HOME/bin/java -server -Xbatch -Xms3800m -Xmx3800m -XX:+AggressiveHeap
-Xss128k spec.jbb.JBBmain -propfile SPECjbb.props
```

Using the preceding options causes the system to attempt to preallocate 3.8 gigabytes on 4 megabyte pages. As always, use `pmap -xs` to ensure that pages were allocated. On UltraSPARC T1 processor-based systems, placing the Java heap on large pages generally gives a large performance increase.

It is important to remember that MPSS is only advisory to the OS. If there are insufficient pages of the requested size in the OS, the heap or stack is placed on the next best fit. It is a good idea to always check the process using `pmap -xs`.

Also note that extra page sizes, 512 kilobytes and 32 megabytes, are supported on other SPARC architectures. If the user specifies one of these page sizes on a UltraSPARC T1 processor-based system, the nearest page size below it is selected (for example, 64 kilobytes for 512 kilobytes, and 4 megabytes for 32 megabytes). Applications might already be using MPSS with these options when they are migrated to a UltraSPARC T1 processor-based system, so it is always a good idea to check.

Accessing the Modular Arithmetic Unit and Encryption Framework

UltraSPARC T1 processors have one modular arithmetic unit (MAU) per core that supports modular multiplication and exponentiation to help accelerate SSL processing. Applications that need to do BigNum computations can access the MAUs through the Solaris Encryption Framework (Solaris EF). This is very common for web servers and application servers that employ Secure Sockets Layer (SSL) for secure transmissions.

If the SSL component of these servers complies with the Public Key Cryptography Standards (PKCS) #11, they can be configured to use the EF for RSA key generation operation. The EF will use the MAU to automatically offload RSA operations. RSA operations are very expensive, and tests have shown that the MAU provides a significant performance boost.

The hardware thread that initiated the MAU stalls for the duration of the operation, but the other three threads on the core can progress normally. The eight MAUs (one per core) result in very high throughput on UltraSPARC T1 processor-based systems for RSA operations.

The MAU is presented to the Solaris OS as a device called ncp—the device through which the Solaris EF processes all operations. There are two kstat counters that can be used to monitor the performance of the Solaris EF and ncp. The following example shows the use of the kstat -n option:

```
kstat -n ncp0
module: ncp                              instance: 0
name:   ncp0                             class:    misc
        crtime                           196.2432695
        dsasign                          26
        dsaverify                        10
        mau0qfull                        0
        mau0qupdate_failure              0
        mau0submit                       42
        mau1qfull                        0
        mau1qupdate_failure              0
        mau1submit                       102
        mau2qfull                        0
        mau2qupdate_failure              0
        mau2submit                       74
        mau3qfull                        0
        mau3qupdate_failure              0
        mau3submit                       88
        mau4qfull                        0
        mau4qupdate_failure              0
        mau4submit                       93
        mau5qfull                        0
        mau5qupdate_failure              0
        mau5submit                       91
        mau6qfull                        0
        mau6qupdate_failure
        mau6submit                       84
        mau7qfull                        0
        mau7qupdate_failure              0
        mau7submit                       105
        rsaprivate                       211
        rsapublic                        211
        snaptime                         536150.521888115
        status                           online
```

Using kstat -n ncp0 | grep rsa shows the number of RSA jobs completed. Using kstat -n ncp0 | grep submit shows the number of RSA jobs submitted to each MAU.

Minimizing Floating-Point Operations and VIS Instructions

UltraSPARC T1 processors have a single floating-point unit (FPU) per chip. The FPU can accommodate a single floating-point (FP) operation at a time with a 40-cycle penalty. Other strands waiting for the FPU are stalled. Some of the more common FP instructions are processed in the core itself to reduce this contention.

In UltraSPARC T1 processor-based systems, excessive FP operations in an application degrade performance. We have done extensive studies and found little or no floating-point contention in most commercial applications. Testing on UltraSPARC T1 hardware has proven this to be correct.

To determine the number of floating-point operations, use the `cpustat` command, as follows:

```
/usr/sbin/cpustat -c pic0=FP_instr_cnt,pic1=Instr_cnt,sys 1 10
   time cpu event   pic0      pic1
  5.021   0  tick    60620   231646080  # pic0=FP_instr_cnt,sys,pic1=Instr_cnt,sys
  5.031   1  tick    40386   235915027  # pic0=FP_instr_cnt,sys,pic1=Instr_cnt,sys
  5.061   0  tick    36884   236575873  # pic0=FP_instr_cnt,sys,pic1=Instr_cnt,sys
  5.061   1  tick    45842   230867805  # pic0=FP_instr_cnt,sys,pic1=Instr_cnt,sys
```

As a rough guideline, performance degrades if the number of floating-point instructions exceeds 1 percent of total instructions. In the preceding example, divide the first column by the second to get the number of FP instructions.

UltraSPARC T1 processors handle floating-point operations differently than previous SPARC processors. FP SPARC instructions FST, FLD, FMOV, FABS, FNEG, Block Load, Block Store, and SIAM have been moved from the FPU to the core. Many of the simple VIS instructions that are not emulated are also moved to the core. The effect is that USIII or USIV floating-point performance counters might give much higher numbers for an application than would be seen for UltraSPARC T1 processors.

If a thread is executing an FP operation and a second thread in the same core also attempts to execute an FP operation, the second thread stalls. Note that the stall also affects in-core FP operations for strands on the same core, but not in-core FP operation on other cores. Therefore, if an application has FP operations from multiple threads, it is best to place them on different cores using `psrsets` or `pbind`.

Emulating Floating Point and VIS Instructions

Finally there are a number of floating point and VIS instructions that are emulated in software. These are not common in commercial applications and require a lot of silicon real estate. The following is a list of these for UltraSPARC T1 processors:

ALLCLEAN, ARRAY{8,16,32}, BMASK, BSHUFFLE, BST_COMMIT, EDGE{8,16,32}{L{N}, FABSq, FADDq, FCMPq, FCMPEq, FCMPEQ{16,32}, FCMPGT{16,32}, FCMPLE{16,32}, FCMPNE{16,32}, FDIVq, FdMULq, FEXPAND, FMOVq, FMOVqcc, FMOVqr, FMULq, FMUL8SUx16, FMUL8ULx16, FMUL8x16, FMUL8x16AL, FMUL8x16AU, FMULD8SUx16, FMULD8ULx16, FNEGq, FPACKFIX, FPACK{16,32}, FPMERGE, FSQRT(s,d,q), F(s,d,q)TO(q), FqTOi, FqTOx, FSUBq, FxTOq, IMPDEP1, IMPDEP2, INVALW, LDDFA, LDQF, LDQFA, NORMALW, OTHERW, PDIST, POPC, PST, SHUTDOWN, STDFA, STQF, STQFA

The emulation is done in the kernel and a number of `kstat` options are available to determine if it is being called. Use the command `kstat | grep fpu_sim` to determine FPU emulations and use `kstat | grep vis_sim` for VIS emulation, as shown in the following example:

```
kstat | grep fpu_sim
        fpu_sim_fabsd             0
        fpu_sim_fabsq             0
        fpu_sim_fabss             0
        fpu_sim_faddd             0
        fpu_sim_faddq             0
        fpu_sim_fadds             0
        fpu_sim_fcmpd             0
        fpu_sim_fcmped            0
..........
```

If the `kstat` results indicate there are calls to emulation, use the following `dtrace D` script to find the caller for FPU emulation:

```
#!/usr/sbin/dtrace -s
fpuinfo:::fpu_*
{
        printf ("pid = %d\n", pid) ;
        ustack (20) ;
}
```

The output from a Java workload appears as shown in CODE EXAMPLE 10.

CODE EXAMPLE 10 Java Workload Output

```
 0   535 _fp_fpu_simulator:fpu_sim_fsqrtd pid = 13233
           libm.so.2`sqrt+0x4c
           libm.so.2`__1cKPSScavengeTinvoke_at_safepoint6FIiri_v_+0xb58

           libm.so.2`__1cUParallelScavengeHeapUcollect_at_safepoint6Mn0AOCollect
 ionType_Iri
 _v_+0x44
           libm.so.2`__1cMVM_OperationIevaluate6M_v_+0x9c

           libm.so.2`__1cIVMThreadSevaluate_operation6MpnMVM_Operation__v_+0x94
           libm.so.2`__1cIVMThreadEloop6M_v_+0x320
           libm.so.2`__1cIVMThreadDrun6M_v_+0x78
           libm.so.2`_start+0x20
           libc.so.1`_lwp_start
```

Use the following `dtrace` `D` script to find the caller for VIS emulation:

```
#!/usr/sbin/dtrace -s

:::vis_fmul:entry
{
        printf ("pid = %d\n", pid) ;
        ustack (20) ;
}
```

About the Author

Denis Sheahan is a Senior Staff Engineer in Sun Microsystems UltraSPARC T1 Architecture Group. During the 12 years he has been at Sun, Denis has focused on application software and Solaris OS performance, with an emphasis on database, application server, and Java technology products. He is currently working on UltraSPARC T1 performance for current and future products.

Denis holds a B.Sc degree in Computer Science from Trinity College Dublin, Ireland. He received the Sun Chairman's Award for innovation in 2003.