LOW LATENCY

Eliminating Application Jitter with Solaris™
White Paper
May 2007

Table of Contents

Background 3

Defining Latency and Jitter 4

Scheduling Classes 5

Memory Locking 9

Early Binding 10

Processor Binding 11

Interrupt Shielding 12

Observability 13

Measuring Latency 16

Realtime Java™ 18

Chapter 1
Background

The financial markets in 2007 are facing an unprecedented number of market wrenching challenges. Two of the key challenges are the continued growth of algorithmic trading and the impending enactment of trading regulations, particularly in the U.S. and EU.

- Algorithmic trading is driving firms to re-architect their trading systems for low latency and performance. Firms are automating trade processes and adding throughput and processing power to their market data and trading systems.
- In 2007, United States Securities and Exchange Commission Regulation for National Market System (Reg NMS) in the U.S., and the Markets in Financial Instruments Directive (MiFID) in the EU, will require most broker-dealers to seek best execution for their clients. In response, firms are adding direct market access (DMA) to exchanges, optimizing their order routing and execution, putting in place high availability systems, and doubling their data storage. Firms that fail compliance face loss of order flow and regulatory fines.

Sun is working closely with leading application partners to address the latency and performance issues across the end-to-end trade process. A key focus is the market data space.1 The annual doubling in market data volume over past years will likely be eclipsed in 2007 due to Reg NMS and MiFID2,, penny quoting of options, proliferation of venues on which instruments trade and of dark pools of liquidity and the unprecedented growth of algorithmic trading. Large sell-side institutions are further exacerbating the issue by adding their own internally generated realtime data to their market data systems. Older market data systems are unlikely to keep up (as evidenced by a publicized hiccup at a major exchange), and trading firms are moving quickly to upgrade these systems.

Sun and its partners are focusing on how financial institutions can:

- Scale and process messages at a rate that may surpass 700,000 msgs/second in 2007 from certain venues such as the Options Price Reporting Authority (OPRA)
- Find and reduce latency within and between market data and trading systems with Solaris™ 10
- Minimize operating risk including the risk of catastrophic failure and security breaches,3
- Implement these changes cost-effectively and with minimal interruptions4.

Latency plays a key role, microseconds imply missed trading opportunities. Predictability is another dimension of latency, it is the guarantee that an application will respond within a bounded time interval to an external event. Jitter is defined as the variability from this interval. Jitter causes applications to behave in unpredictable ways, the anti-thesis of a low latency trading environment. The proliferation of algorithmic trading is driving the focus on reducing, even eliminating jitter, thus allowing exploitation of arbitrage opportunities in a predictable fashion.

Solaris has a well defined methodology for eliminating jitter. It has a concept of multiple Scheduling Classes, some of which are well suited for such applications. Processor binding, Interrupt Shielding, Memory Locking and Early Binding are associated techniques used to create well behaved low latency applications.

1 While market data is commonly thought of as external price quotes, orders, and news from market data vendors like Bloomberg and Reuters and from brokers/exchanges like ICAP and NYSE/Arca respectively, market data also includes internally generated data. This includes internal prices, broker pages, news including events, and post-trade data such as positions, pricing, analytics, risk, etc.

2 For more information on Reg NMS and MiFID, please visit www.sun.com

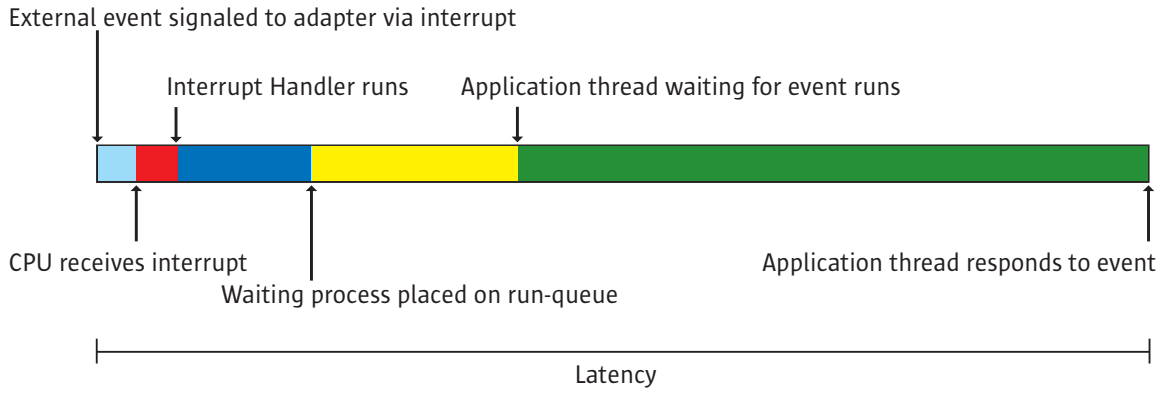
3 Introduced by off shoring, outsourcing or hosting at 3rd parties e.g. co-location at exchanges

4 Solaris' choice of hardware platforms and binary compatibility allow firms choice of when to upgrade and on which platform. Solaris support of Containers and CoolThread technologies allow firms to consolidate multiple applications onto a single server not only saving space and energy but reducing latency. This space and energy saving is further multiplied when running the application on CoolThread-based systems making Solaris 10 especially attractive as a platform for proximity hosting at exchanges by customers looking to reduce trade latency.

Chapter 2

Defining Latency and Jitter

Trading applications respond to external events. To an Operating System (OS), an external event is signalled via an interrupt. A series of events occur when a Server receives an interrupt. The hardware adapter receives the interrupt and uses a finite amount of time to pass this interrupt to the CPU by raising the CPU's interrupt pin. Subsequently the CPU passes the interrupt to the OS. The OS runs the interrupt handler, likely stealing the context of the running thread. In case of a network interrupt delivered due to the arrival of a packet, the interrupt handler will likely queue the packet and return. The applications thread waiting for this packet – likely a part of the trading application – via a poll() or similar system call, is then put on the run-queue. Depending on its priority and scheduling class, it could preempt another running thread, forcing a context switch. The processing thread now processes this external event, potentially causing a series of pages faults to page its working set in, and responds.



Latency is defined as the time spent between the Server receiving the external interrupt and the processing thread responding to it. The Server vendor is clearly responsible for the amount of time spent between the Server receiving the interrupt and the OS being notified. The OS vendor is responsible for the time spent between the OS receiving this signal and the waiting application thread being put on the CPU to run. The remainder of the Latency could be viewed as application specific – the time spent between the application thread being put on the CPU and it responding to the event. However the OS vendor is responsible for other runtime specifics, such as paging the working set in, allowing the application to lock its pages in memory to prevent paging, not hijacking the CPU for processing other interrupts, allowing the application to use the CPU without lowering its priority etc. It is clearly these factors that contribute to the unpredictability of application latency, thus introducing jitter.

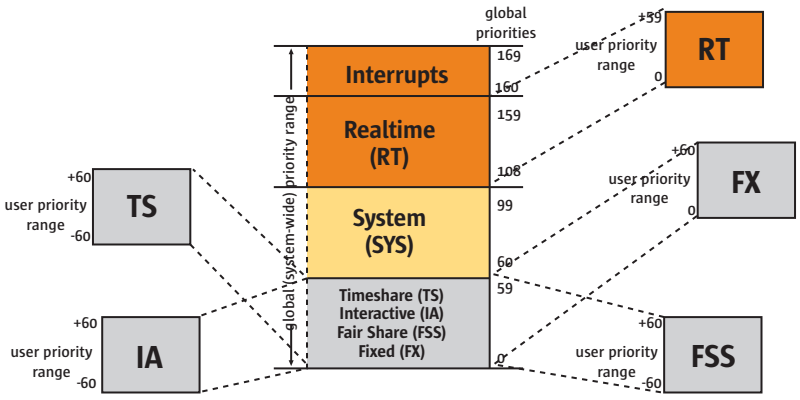
Solaris has specific features that eliminate all known causes of jitter, thus allowing Solaris to be classified as a General Purpose Real Time Operating System. These features are discussed in the following sections.

Chapter 3

Scheduling Classes

Solaris 10 defines six distinct Scheduling Classes, all with varying characteristics. Each Scheduling Class has a “local” or “user” range of priorities. Depending on the Class, the Scheduler can change the user priority of a running thread on the fly, however moving a thread to a different class requires manual intervention with appropriate privileges. The local priorities map onto a Global Priority range, and it is this range that determines the system-wide priority of the thread. Tools such as nice(1) only affect the user priority.

As can be seen from the figure below, Timesharing (TS), Interactive (IA), Fixed Priority (FX) and the Fair Share Scheduling (FSS) classes map onto the lowest global priority range.



TS is the default scheduling class, and is characterized by a fixed time quantum, with the scheduler adjusting priorities based on the thread's CPU usage. Threads using up their entire quanta are penalized by lowering of their priorities, to assure fairness and maximize throughput of the system.

Without this, lower priority threads would rarely get a chance to run. The IA class is used for the Windowing system, the scheduler raises the priority of the thread associated with the focussed window.

```
ambreesh@vitalstatistix:~> dispadmin -l
CONFIGURED CLASSES
==================

SYS       (System Class)
TS        (Time Sharing)
IA        (Interactive)
FSS       (Fair Share)
RT        (Real Time)
FX        (Fixed Priority
```

The FX and FSS classes were introduced in Solaris 9. The FSS class was adopted from Solaris Resource Manager, it introduces the concept of scheduling based on userids (amongst other entities such as zones). Threads in the FSS scheduling class adhere to Shares assigned to the owner userid. Assume userids A, B and C are assigned 20, 30 and 50 Shares respectively. This will result in all sum total of the threads belonging to userid A being guaranteed a minimum of

20 / (20+30+50) = 20%

of the total CPU resources on the system. Threads belonging to userid B will get a minimum of 30%, and userid C 50%. The thread priorities are adjusted by the Scheduler accordingly to ensure these guarantees.

```
# sleep 5000&
23669

# ps -cf | grep 23669
    root 23670 23667   TS  49 11:19:27 pts/8       0:00 grep 23669
    root 23669 23667   TS  59 11:19:14 pts/8       0:00 sleep 5000

# priocntl -s -c FX -m 60 -p 60 23669

# ps -cf | grep 23669
    root 23673 23667   TS  49 11:20:58 pts/8       0:00 grep 23669
    root 23669 23667   FX  60 11:19:14 pts/8       0:00 sleep 5000
```

The FX class introduces the concept of Fixed Priority scheduling in the same global priority range as TS, IA and FSS. Processes in the FX class will never have their priorities adjusted by the scheduler.

Note that FX maps to the global priority range 0-60, whereas TS, IA and FSS map to 0-59. This allows processes assigned to FX at priority 60 to run at higher priority than all user processes in the TS, IA and FSS classes.

```
root@vitalstatistix:/# dispadmin -g -c FX
# Fixed Priority Dispatcher Configuration
RES=1000

# TIME QUANTUM                    PRIORITY
# (fx_quantum)                      LEVEL
        200                #         0
        200                #         1
        200                #         2
        200                #         3
        200                #         4
        200                #         5
        200                #         6
        200                #         7
        200                #         8
        200                #         9
        160                #        10
        160                #        11
        160                #        12
      <--- snip ---->
        120                #        27
        120                #        28
        120                #        29
         80                #        30
         80                #        31
      <--- snip ---->
         40                #        57
         40                #        58
         20                #        59
         20                #        60
```

The SYS class is meant for kernel threads, and like FX, is fixed priority.

The RT class is fixed priority, and provides a priority range for user threads higher than all other user and OS threads on the system. The exception is the interrupt priority range, which maps to a global range higher than RT. The Solaris RT scheduling class is POSIX 1003.1b compliant, and allows users to specify First-In-First-Out (FIFO) or Round-Robin scheduling policy. The FIFO scheduling policy allows the highest priority thread to run until it blocks, or a higher priority thread becomes runnable. Hence it is based on pure priority preemption. Round-Robin adds a time quantum to pure preemption. The highest priority thread will run until it either blocks, is preempted by a higher priority thread or its time quantum expires and a thread of equal or higher priority is ready to run. Different priorties have different default time quanta assigned. The values can be viewed via dispadmin(1M).

The Solaris kernel is fully pre-emptable, which makes Solaris a true real time operating system. It is ludicrous to believe that applying a “real time patch” makes a general purpose operating system into a real time OS. Unless the kernel is fully pre-emptable, RT threads would get hijacked if a kernel thread was running and the kernel was not at a preemption point. Readers need to question the ability of other OS vendors claiming to make their kernel fully preemptable – a requirement of a real time OS – by applying a patch.

For fixed priority scheduling classes (FX, SYS and RT), Priority Inversion occurs when a lower priority thread (LT) holds a resource that a higher priority thread (HT) is blocking on. Any thread with higher priority than LT can preempt LT, and hence run while HT, which is the highest of the three priorities, continues to block. There is no limit to the number of threads that can interpose between LT and HT, thus continuing to starve LT and block HT. Priority Inheritance is used avoid this situation. This allows LT to assume the priority of the blocked higher priority thread – HT, thus allowing it to run at a higher priority to complete its task and release the blocking resource. Kernel priority inheritance has been a part of the Solaris kernel since its inception. Solaris 8 added user level priority inheritance.

One source of jitter in an application is the preemption of its threads by higher priority threads. Another is the lowering of the priority of one or more of its threads by the Scheduler. The latter can be mitigated by putting the relevant threads in a fixed priority scheduling class – FX or RT. Using FX at priority 60 for these threads allow them to run at a priority level higher than any other user threads on the system, thus preventing them from being preempted by any other user threads. Unless there are RT threads running on the system, the only threads running at a higher priority are the threads in the SYS class and the interrupt threads, which is likely desirable for mature operating systems. Alternatively, running the relevant threads in RT make them higher priority than even the kernel threads, allowing rogue threads to hijack the system. The FX class at priority 60 is a safer alternative, and is available only on Solaris. Realize that it is possible to make the system unresponsive to interactive login or maintenance (processes that run in TA or IA class) if a process in the FX class at priority 60 becomes uncontrollable. However, processes in the SYS and RT class continue to run.

Chapter 4

Memory Locking

All modern operating systems support the concept of virtual memory, which inherently implies that the OS pretends to have more physical memory than is present. It does this by using the disk – swap for anonymous memory, and the filesystem for named memory – as an extension of the physical memory, by paging on demand. Since paging involves the disk subsystem, one needs to be very aware of the detrimental effects of paging since disks are about 6 orders of magnitude slower than physical memory (nanosecond vs. millisecond access times).

```
ambreesh@vitalstatistix:~>  cat mlockall.c
#include <sys/mman.h>
#include <stdio.h>

int main()
{

/*---------- Do setup work here ----------*/

if (mlockall (MCL_CURRENT|MCL_FUTURE) == -1) {
        perror ("MLOCKALL failed");
        return (-1);
        }

printf ("mlockall() successful.\n");

/*-------- Do critical work here ---------*/

munlockall();

}

ambreesh@vitalstatistix:~>  ppriv -v $$
875:    bash
flags = <none>
        E: file_link_any,proc_exec,proc_fork,proc_info,
proc_lock_memory,proc_session
        I: file_link_any,proc_exec,proc_fork,proc_info,
proc_lock_memory,proc_session
 -------------- snipped -----------

ambreesh@vitalstatistix:~> cc mlockall.c

ambreesh@vitalstatistix:~> ./a.out
mlockall() successful.
```

Solaris allows locking of a processes pages into memory using mlock(3C) and mlockall(3C). The requesting process must have the appropriate privileges. For any latency sensitive application, all memory should be pre-faulted (allocated and touched) and then locked down. All current and future faulted translations may be locked by specifying the MCL_CURRENT and MCL_FUTURE flags respectively to mlockall(3C).

Chapter 5
Early Binding

Solaris has supported the concept of dynamic linking for a long time. Dynamically linked libraries offer distinct advantages over statically linked libraries, since the linking of symbols in the shared libraries is done lazily at the first invocation of the symbol. This allows the dynamically linked binaries to be portable across OS releases, improves performance as most applications don't use all symbols at runtime, and reduces the size of the binaries. However, this comes at a cost of likely introducing jitter, as the application “pauses” while the dynamic library references are resolved and likely paged in, perhaps off the network. Solaris however provides for early binding of all symbols by setting the LD_BIND_NOW environment variable to a non-zero value. This causes all the symbols to be bound at application start-up time, likely increasing the application invocation time, however eliminating the jitter.

Chapter 6

Processor Binding

Solaris allows threads to be bound to specific CPUs, thus giving these threads warm caches. processor_bind(2) and pbind(1M) give details on how to accomplish processor binding. However this mechanism does not prevent other unbound

```
root@vitalstatistix:/# psrinfo
0       on-line   since 03/27/2007 10:03:39
1       on-line   since 03/27/2007 10:03:39
2       on-line   since 03/27/2007 10:03:39
3       on-line   since 03/27/2007 10:03:57

root@vitalstatistix:/# psrset -c 0 1
created processor set 1
processor 0: was not assigned, now 1
processor 1: was not assigned, now 1

root@vitalstatistix:/# psrset -b 1 `pgrep rvd`
process id 2667: was not bound, now 1
```

threads from running on these CPUs, thus somewhat reducing the warm cache effect. Solaris 2.6 introduced the concept of Processor Sets, which are groups of specific CPUs. Threads can be bound to a processor set, and only these bound threads run on the CPUs assigned to this processor set. This mechanism considerably mitigates the processor binding problem mentioned above. psrset(1M), pset_assign(2) and pset_bind(2) are used to create and manage processor sets.

Processor sets may be created in a multi CPU environment by invoking psrset(1M) with appropriate syntax.

A CPU could be a single processor (a traditional, single-core x86/UltraSPARC processor), a single core (on a multi-core UltraSPARC, AMD or Intel socket) or a single hardware thread (on the CMT UltraSPARC T1 processor). It is hence any processing entity seen by Solaris with the ability to have work dispatched and scheduled on it.

Creating a processor set and binding threads to it eliminates jitter caused by cache misses.

Chapter 7

Interrupt Shielding

Solaris 7 introduced the concept of preventing a CPU from handling unbound interrupts – interrupt

```
root@vitalstatistix:/# psrinfo
0       on-line   since 03/27/2007 10:03:39
1       on-line   since 03/27/2007 10:03:39
2       on-line   since 03/27/2007 10:03:39
3       on-line   since 03/27/2007 10:03:57

root@vitalstatistix:/# psradm -i 1 2 3

root@vitalstatistix:/# psrinfo
0       on-line   since 03/27/2007 10:03:39
1       no-intr   since 03/27/2007 10:24:24
2       no-intr   since 03/27/2007 10:24:24
3       no-intr   since 03/27/2007 10:24:24
```

shielding. psradm(1M), psrset(1M) are used to accomplish this. Shielding interrupts on CPUs running the low latency threads eliminates the jitter caused by interrupt processing, since interrupt threads run at the highest priority on the system and may interrupt time-critical threads that are of interest to us.

It is possible to disable interrupt handling on individual processors or on entire processor sets .

A CPU under the control of Solaris is always in one of three possible states:

on-line (the default state: available for thread scheduling and interrupt handling)
off-line (not available for thread scheduling)
no-intr (available for thread scheduling, but does not handle network or I/O interrupts)

Interrupt Shielding eliminates jitter caused due to interrupt threads preempting low latency threads, which would otherwise result in context switches and cache thrashing.

```
  0    0   0    3  3459 3083  523   16    0  204    0   685    1   4   0  95
  1    0   0    0    84    0  848    0    0  153    0  9617    5   6   0  89
  2    0   0    0   199    0 1460    0    0    0    0  2535    4   1   0  95
  3    0   0    0    59    0  905    0    0   94    0  4570    3   1   0  96
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0   46  3397 3026  509   12    0  225    0  7194    4   5   0  91
  1    0   0    0    99    0  786    0    0  147    0  9198    6   6   0  89
  2    0   0    0   208    0 1407    0    0    1    0  2394    4   1   0  95
  3    0   0    0    73    0  859    0    0   89    0  4335    4   2   0  94
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   2  459  3375 2980  553   15    0  190    0  2668    3   4   0  93
  1    0   0    0   245    0  928    0    0  151    0  9555    5   5   0  90
  2    0   0    0   326    0 1386    0    0    0    0  2440    4   2   0  94
  3    0   0    0   215    0  891    0    0  105    0  4452    3   1   0  96
^C
perf-win77:/rce/rv-ip/bin>
```

Chapter 8

Observability

Solaris 10 introduces DTrace, a dynamic tracing mechanism. With DTrace, all of the above defined sources of potential jitter and latency can easily be identified and quantified. Since DTrace has no disabled probe-effect and can be used to dynamically instrument the system and any running application and has very low enabled probe-effect, it serves as the perfect tool to observe sources of unwanted latency and jitter affecting an application.

For instance, if one wanted to quantify the number and sources of preemption affecting an application, the /usr/demo/dtrace/whopreempt.d script shows which processes were preempted (and by whom) and the number of times it occurred. Also, the current priorities of the preemptor and preempted are identified. This can be used as a clear picture to determine if, for instance, the FX scheduling class would be a candidate to address the issue. With a simple modification, the script can also be made to show the amount of time we spend off-cpu in the event of a preemption.

Prior to DTrace, one could see page faults occurring and identify the applications affected with tools like mpstat(1M) and prstat(1M). Now however, it is trivial to identify the source of the page faults and the memory associated with them.

```
ambreesh@vitalstatistix:~> # dtrace -n vminfo::as_fault:'{@[pid, execname, ustack()] = count()}'
^C
     8624  java
              libzip.so`0xfe741fe1
              libzip.so`0xfe741b87
              libzip.so`ZIP_Open+0x21
libjvm.so`__1cLClassLoaderXcreate_class_path_entry6FpcnEstat_ppnOClassPathEntry__v_+0x1ad
libjvm.so`__1cLClassLoaderbCupdate_class_path_entry_list6Fpkc_v_+0x56
libjvm.so`__1cLClassLoaderbBsetup_bootstrap_search_path6F_v_+0xc0
              libjvm.so`__1cLClassLoaderKinitialize6F_v_+0x100
              libjvm.so`__1cQclassLoader_init6F_v_+0x15
              libjvm.so`__1cMinit_globals6F_i_+0x24
libjvm.so`__1cHThreadsJcreate_vm6FpnOJavaVMInitArgs_pi_i_+0x1c5
              libjvm.so`JNI_CreateJavaVM+0x8f
              java`main+0x565
              java`0x80512ca
               55
     8624  java
              libc.so.1`memset+0x90
              libzip.so`0xfe741ebc
              libzip.so`0xfe741b87
              libzip.so`ZIP_Open+0x21
libjvm.so`__1cLClassLoaderXcreate_class_path_entry6FpcnEstat_ppnOClassPathEntry__v_+0x1ad
libjvm.so`__1cLClassLoaderbCupdate_class_path_entry_list6Fpkc_v_+0x56
libjvm.so`__1cLClassLoaderbBsetup_bootstrap_search_path6F_v_+0xc0
              libjvm.so`__1cLClassLoaderKinitialize6F_v_+0x100
              libjvm.so`__1cQclassLoader_init6F_v_+0x15
              libjvm.so`__1cMinit_globals6F_i_+0x24
libjvm.so`__1cHThreadsJcreate_vm6FpnOJavaVMInitArgs_pi_i_+0x1c5
              libjvm.so`JNI_CreateJavaVM+0x8f
              java`main+0x565
              java`0x80512ca
              102
     8619  dtrace
              libc.so.1`_malloc_unlocked+0x1fb
              libc.so.1`malloc+0x33
              libdtrace.so.1`dt_aggregate_snap_cpu+0x3a8
              libdtrace.so.1`dtrace_aggregate_snap+0x8c
              libdtrace.so.1`dtrace_work+0xb7
              dtrace`main+0x1269
              dtrace`0x4027bc
              140
     8624  java
              libc.so.1`memcpy+0x1b
              libzip.so`0xfe741b87
              libzip.so`ZIP_Open+0x21
libjvm.so`__1cLClassLoaderXcreate_class_path_entry6FpcnEstat_ppnOClassPathEntry__v_+0x1ad
libjvm.so`__1cLClassLoaderbCupdate_class_path_entry_list6Fpkc_v_+0x56
libjvm.so`__1cLClassLoaderbBsetup_bootstrap_search_path6F_v_+0xc0
              libjvm.so`__1cLClassLoaderKinitialize6F_v_+0x100
              libjvm.so`__1cQclassLoader_init6F_v_+0x15
              libjvm.so`__1cMinit_globals6F_i_+0x24
libjvm.so`__1cHThreadsJcreate_vm6FpnOJavaVMInitArgs_pi_i_+0x1c5
              libjvm.so`JNI_CreateJavaVM+0x8f
              java`main+0x565
              java`0x80512ca
              177
```

Another common source of latency and jitter (not previously discussed), is lock contention within an application. Heavily contended locks will cause competing threads to spin and then block (if the lock cannot be acquired) until the lock is released. This spin and block time can become quite substantial and cause serious jitter in an application as the number of threads competing for locks becomes larger, or if the critical sections are substantial. A command - plockstat(1M) - implemented as a DTrace consumer can be easily used to identify heavily contended locks and to quantify the amount of time spent competing for these locks.

This just barely scrapes the surface of DTrace's ability to help identify and eliminate jitter in an application. Observability of the problem is the major hurdle to overcome that allows for the appropriate application of the technologies available in Solaris to address jitter and latency.

Chapter 9

Measuring Latency

As important as elimination of jitter from latency is the ability to measure latency. With markets requiring microsecond latencies, one must have timers and timestamps available to get measurements with such fine granularity.

Solaris implements fast nanosecond timers via gethrtime(3C). gethrtime(3C) is not correlated to time-of-day, and should be used for performance related timestamps. Applications requiring time-of-day timestamps should use gettimeofday(3C).

Solaris has various timers available. Timers are alarms, and are used to send signals to user threads at specific intervals. BSD-style interval timers are available – setitimer(2), getitimer(2). Solaris 2.6 introduced support for POSIX.1b timers – timer_create(3RT) and timer_settime(3RT). These timers are based on CLOCK_REALTIME, the BSD timers and poll(2) timeouts are dispatched by the system clock.

```
ambreesh@vitalstatistix:~> cat gethrtime_example.c
#include <sys/time.h>
#include <stdio.h>
#include <unistd.h>

#define HOSTNAME 100

int main()
{

hrtime_t begin, end;
int i, n = 100;
char hostname [ HOSTNAME] ;

begin = gethrtime();  /*-- Get timestamp --*/

/*-- Do work --*/
for (i = 0; i < n; i++)
            gethostname(hostname, HOSTNAME);

end = gethrtime();  /*-- Get another timestamp --*/

printf("Avg gethostname() time = %lld nsec\n", (end - begin)/n);
}

ambreesh@vitalstatistix:~> cc gethrtime_example.c

ambreesh@vitalstatistix:~> ./a.out
Avg gethostname() time = 1101 nsec
```

```
ambreesh@vitalstatistix:~> cat clock_getres_example.c
#include <stdio.h>
#include <time.h>

int main()
{

struct timespec a_timespec;

if (clock_getres (CLOCK_HIGHRES, &a_timespec) < 0) {
     perror ("clock_getres() failed\n");
     return (-1);
}

printf ("CLOCK_HIRES resolution: %ld.%09ld sec\n", a_timespec.tv_sec,
a_timespec.tv_nsec);
}

ambreesh@vitalstatistix:~> cc clock_getres_example.c -lrt

ambreesh@vitalstatistix:~> ./a.out
CLOCK_HIRES resolution: 0.000000009 sec

ambreesh@vitalstatistix:~> psrinfo -v
Status of virtual processor 0 as of: 04/08/2007 15:18:36
  on-line since 04/08/2007 15:03:47.
  The i386 processor operates at 2000 MHz,
        and has an i387 compatible floating point processor.

/*------- On SPARC ------*/

ambreesh@getafix$ psrinfo -v
Status of virtual processor 0 as of: 04/09/2007 16:57:09
  on-line since 10/30/2006 02:44:05.
  The sparcv9 processor operates at 1281 MHz,
     and has a sparcv9 floating point processor.

ambreesh@getafix$ ./a.out
CLOCK_HIRES resolution: 0.000000100 sec
```

By default the system clock executes at 100Hz, yielding a resolution of 10 milliseconds. Starting with Solaris 2.6, this value can be adjusted via setting hires_tick to 1 in /etc/system. This value yields a system clock resolution of 1 millisecond.

Even though one can adjust the system clock resolution via hires_tick, a more unobtrusive mechanism for generating high resolution timers is desired. Additionally, a 1ms timer is likely still too coarse, and certain applications desire an interval timer rate dissociated from the system clock rate.

Solaris 8 introduced a new POSIX clock type – CLOCK_HIRES. CLOCK_HIRES timers don't rely on the system clock, they deal directly with the hardware timer source. Timer resolution is as fine grained as the platform, up to a 1 nanosecond resolution. clock_getres(3RT) can be used to determine the platform's CLOCK_HIRES resolution.

Chapter 10

Realtime Java™

Java is used by financial firms worldwide for various applications, including trading applications. The inherent nature of Java makes it an ideal platform for application development, both due to the shortened application development lifecycle and application portability. However Java applications still suffer from the perception of being non-deterministic in nature, primarily due to the Garbage Collection (GC) process. As the name suggests, GC cleans up freed memory and places it in a free memory pool to be reused. However the GC routines run in an unpredictable fashion, depending primarily on the application's memory consumption and freeing patterns and the JVM tuning parameters.

The JVM thread runs at a high priority, which implies preemption and preference being given to the GC thread, thus introducing significant jitter.

Sun's implementation of the Real-Time Specification for Java (RTSJ), the Java RTS product, enables realtime processing by using techniques that protect low latency threads from the garbage collector. These threads can and will run at the highest priority and they will interrupt any and all other activity on the system to accomplish their functions. This means that trading systems can confidently monitor the market and take action well within the window of opportunity. Sun's Java RTS system has a latency of 20 microseconds on it's reference platform (a relatively modest V240 running dual 1 GHz processors and Solaris 10).

Java RTS is fully compatible with standard J2SE applications, allowing customers to run their existing J2SE code and run it on Java RTS without any failures. They can then add the necessary realtime components as needed.