

A large, abstract, light gray graphic on the left side of the page, consisting of several overlapping, curved shapes that create a sense of depth and movement.

# DEVELOPING SCALABLE APPLICATIONS for **THROUGHPUT COMPUTING**

Reaping the Benefits of the Chip Multithreaded  
UltraSPARC<sup>®</sup> T1 Processor with CoolThreads<sup>™</sup> Technology

White Paper  
November 2005

# Table of Contents

<b>Executive Summary</b> .....	<b>1</b>
<b>An Introduction to Throughput Computing</b> .....	<b>2</b>
Throughput Computing and Chip Multithreading (CMT) Technology .....	2
The Diminishing Returns of Complex Processor Design .....	2
Hardware Multithreading .....	4
Chip Multithreading (CMT) .....	5
Throughput Computing and Performance .....	6
Ideal Applications for CMT Based Systems .....	6
<b>Overview of the UltraSPARC T1 Processor with CoolThreads Technology</b> .....	<b>8</b>
The UltraSPARC T1 Processor at a Glance .....	8
Solaris 10 Features for Throughput Computing .....	9
Applications and the UltraSPARC T1 Processor .....	10
<b>Optimizing Applications for Throughput Computing and CMT Processors</b> .....	<b>11</b>
Sun Studio Compiler Optimization .....	11
JVM Optimizations .....	12
Locking Mechanisms .....	12
Large Pages .....	14
Large Pages for Native Solaris Applications .....	14
Large Pages for Java Applications .....	14
Critical Sections .....	15
Polling and Asynchronous Operations .....	15
Performance Tools .....	16
Sun Studio Performance Analyzer .....	16
Solaris Dtrace .....	16
Solaris Commands .....	17
<b>Conclusion</b> .....	<b>18</b>
<b>References</b> .....	<b>19</b>
<b>Glossary</b> .....	<b>20</b>

## Executive Summary

Traditional processor design has long emphasized the performance of a single hardware thread of execution, and focused on providing high levels of instruction-level parallelism (ILP). These increasingly complex processor designs have been driven to very high clock rates (frequencies), often at the cost of increased power consumption and heat production. Unfortunately, the impact of memory latency has meant that even the fastest single-threaded processors spend most of their time idle, waiting for memory. Complicating this tendency, many of today's complex commercial workloads are simply unable to take advantage of instruction-level parallelism, instead benefiting from thread-level parallelism (TLP).

Consistent with Sun's eco-responsibility stance, Throughput Computing represents a new paradigm in system design, with a focus toward maximizing the overall throughput of key commercial workloads while dramatically changing energy consumption for power and cooling. Chip multithreading (CMT) processor technology is key to this approach, providing a new thread-rich environment that drives application throughput and processor resource utilization while effectively masking memory access latencies. For example, Sun's new UltraSPARC® T1 processor with CoolThreads™ technology gives applications access to up to 32 hardware threads (or strands), essentially providing a symmetric multiprocessing (SMP) system on a single chip.

Systems based on the UltraSPARC T1 processor are expected to be ideal for commercial applications, especially those powering web and transaction services. In addition to providing ideal platforms for Java application servers and enterprise application servers (e.g. ERP and CRM), these systems will enable considerable web- and application-tier consolidation in the data center, dramatically reducing the number of servers required to deliver a given workload while requiring considerably less power and space. At the same time, these reductions will help make room for the expansion of services and content in the data center as required by the business.

Sun was an early leader in multithreaded SMP systems—experience that is now paying dividends for Throughput Computing workloads. The UltraSPARC T1 processor is the culmination of these efforts, delivering the advantages of SMP architectures on a single chip for enhanced performance, increased reliability, and lower costs. In addition to its advanced systems for Throughput Computing, Sun now has the most complete and robust software infrastructure of any vendor today—from the proven Solaris™ Operating System to high-performance threaded libraries and compilers, to innovative Java technology. These strengths combine to provide drastically improved delivered performance per watt of power consumed, and systems that are less expensive to run and to cool.

What ever the advantages of this new approach, the last thing that developers or customers need is another computing architecture to port to and support, so Sun is bringing this revolutionary new technology to market in the familiar SPARC® architecture. Thanks to Sun's Binary Compatibility Guarantee and consistent architecture, existing Solaris applications for the SPARC architecture simply run with no modification whatsoever. Applications that scale well on today's SMP systems should also scale well on CMT platforms such as those powered by the UltraSPARC T1 processor with CoolThreads technology. At the same time, developers need to consider this new direction in system architecture so that existing and future applications can continue to take full advantage of the massive thread-level resources that CMT technology provides. This paper discusses the implications of CMT technology for software developers, and covers techniques for maintaining scalable application design, with emphasis on both the Solaris Operating System and Java™ technology.

## Chapter 1

# An Introduction to Throughput Computing

Organizations that depend on information technology (IT) face a sort of perfect storm of seemingly contradictory challenges. An endless variety of new networked devices and users are demanding ever-higher levels of performance, capacity, availability, and security from applications and services. At the same time, real estate concerns along with rapidly rising energy costs for both power and cooling are now significant factors that discourage merely adding endless racks of traditional servers. The cost and complexity of managing very large numbers of systems is another very real concern, especially when coupled with the very low levels of utilization found in traditional infrastructure.

Beyond packaging, these issues drive to the very technology used to design processors and systems. Traditional high-frequency, single-threaded processors are increasingly producing diminishing returns. Unfortunately, ever-higher clock rates are yielding only small improvements in real-world application performance. At the same time, these high-frequency processors produce higher costs in the form of higher levels of power consumption, and significantly higher levels of heat load that must be addressed by large and expensive HVAC systems. With economic and competitive pressures at an all-time high, most understand that significant change is needed.

### Throughput Computing and Chip Multithreading (CMT) Technology

Faced with these challenges, Sun's drive has been to reduce complexity and management costs by engineering processors and systems that provide orders of magnitude higher throughput, while at the same time drastically changing the assumptions for power and cooling. Sun's Throughput Computing initiative is a wide-ranging and systemic initiative focused on the performance of key workloads, not just the frequency of the processor:

- Chip multithreaded (CMT) processor designs such as the new UltraSPARC T1 processor with CoolThreads technology provide truly massive amounts of thread-level parallelism (TLP) and increased application throughput, with very attractive power and cooling profiles.
- The innovative and secure Solaris™ 10 Operating System effectively delivers the resources of CMT processors to applications, facilitating fine-grained system virtualization and very high levels of utilization while staunchly upholding Sun's binary compatibility guarantee.
- Sun's compilers, development tools, APIs, and systems-level perspective leverage the considerable resources of CMT processors, resulting in virtually unprecedented performance improvements for real-world applications.

### The Diminishing Returns of Complex Processor Design

While optimistic marketing statements constantly call attention to presumably impressive multiple-gigahertz frequencies for new generations of processors, corresponding small gains in real-world system performance and productivity continue to frustrate IT professionals. Throughput Computing, along with Sun's focus on optimizing real workload performance is designed to help resolve these divergent trends. This approach provides higher levels of *delivered* performance and computational throughput while greatly simplifying the data center. Understanding the importance of throughput computing requires a look at how both processors and systems have been designed in the past, and the trends that are defining better ways forward.

The oft-quoted tenant of Moore’s Law states that the number of transistors that will fit in a square inch of integrated circuitry will approximately double every two years. For over three decades the pace of Moore’s Law has held, driving processor performance to new heights. Traditional processor manufacturers have long exploited these chip real estate gains to build increasingly complex processors, with instruction-level parallelism (ILP) as a goal. Today these traditional processors employ very high frequencies along with a variety of sophisticated tactics to accelerate a single instruction pipeline, including:

- Large caches
- Superscalar designs
- Out-of-order execution
- Very high clock rates
- Deep pipelines
- Speculative pre-fetches

While these techniques have produced faster processors with impressive-sounding multiple-gigahertz frequencies, they have largely resulted in complex, hot, and power-hungry processors that don’t serve many modern applications or fit the constraints of modern data centers. In fact, many of today’s data center workloads are simply unable to take advantage of the hard-won ILP provided in these processors. As shown in Table 1, applications with high shared memory and data requirements in particular are typically more focused on processing a large number of simultaneous threads (TLP) rather than running a single thread as quickly as possible (ILP).

*Table 1. Attributes of common commercial workloads favor thread-level parallelism over instruction-level parallelism*

Workload Attributes	Web-Centric		Application-Centric		Data-Centric	
	Web (SPECweb99)	Application (SPECjAppServer2002)	SAP-SD 2Tier	Data (TPC-C)	SAP-SD 3Tier (DB)	DSS (TPC-H)
<b>Application category</b>	Web server	Server, Java™	ERP	OLTP	ERP	DSS
<b>Instruction-level parallelism</b>	Low	Low	Medium	Low	Low	High
<b>Thread-level parallelism</b>	High	High	High	High	High	High
<b>Instruction/Data working set</b>	Large	Large	Medium	Large	Large	Large
<b>Data sharing</b>	Low	Medium	Medium	High	High	Medium

(SPEC, SPECweb, and SPECjbb are registered trademarks of the Standard Performance Evaluation Corporation (SPEC). SAP is a registered trademark of SAP AG in Germany and other countries. TPC-C and TPC-H are trademarks of the Transaction Processing Performance Council (TPC). For more information see [www.spec.org](http://www.spec.org), [www.sap.com/benchmark](http://www.sap.com/benchmark), and [www.tpc.org](http://www.tpc.org).)

Complicating matters, the disparity between memory access speeds and processor speeds means that memory latency dominates performance, erasing even very impressive gains in clock rates. While processor speeds continue to double every two years, memory speeds have typically doubled only every six years. This growing disconnect is the result of memory suppliers focusing on density and cost as their design center, rather than speed. Unfortunately, this relative gap between processor and memory speeds leaves ultra-fast processors idle as much as 85 percent of the time, waiting for memory to return required data. Ironically, as processors get faster and more complex, the effect of memory latency grows—fast, expensive processors spend more cycles doing nothing. It’s easy to see that frequency (gigahertz) is truly a misleading indicator of real performance.

Figure 1 illustrates how even doubling processor performance often provides only a small relative increase in application performance. In this example, though the compute time is reduced by half, only a small overall improvement in execution time results due to the constant and dominant influence of memory latency.

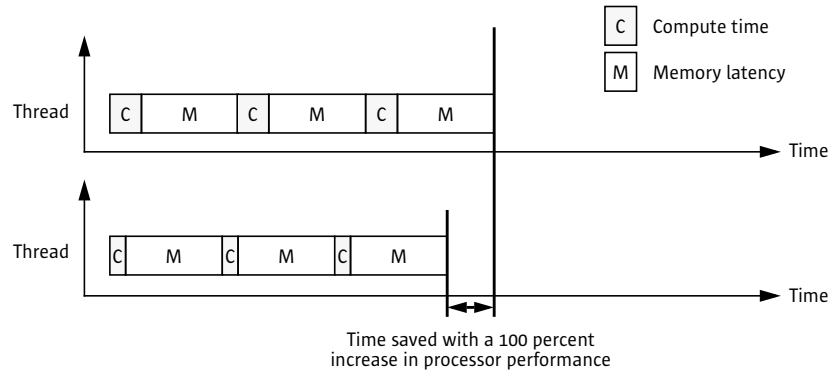


Figure 1. Increasing single-threaded processor performance by 100 percent (a 50-percent reduction in compute-time) provides only a small relative gain in application performance due to memory latency

**Hardware Multithreading**

Dissatisfied with these diminishing returns, Sun’s extensive in-house design team—one of the largest microprocessor design engineering team in the world—has taken a bold new approach to processor design. Sun understands that the network-computing environments found in most modern data centers are inherently multithreaded, where the execution speed of an individual thread is typically less important than overall application throughput. For this reason, Sun is focusing on processors and architecture that are designed to maximize throughput for specific groups of network-computing workloads. These efforts are resulting in new chip multithreading (CMT) processors that leverage the additional gains delivered by Moore’s Law to provide thread-level parallelism rather than instruction-level parallelism.

Unlike traditional single-threaded processors, hardware multithreaded processors allow rapid switching between active threads as other threads block for memory. Figure 2 illustrates a hypothetical CMT processor that is designed to switch between up to four threads as each thread blocks for memory. With this approach, the processor pipeline remains active doing real useful work, even as memory operations for blocked threads continue in parallel.

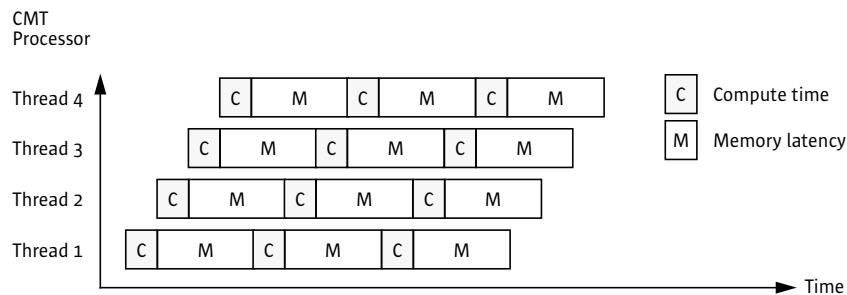
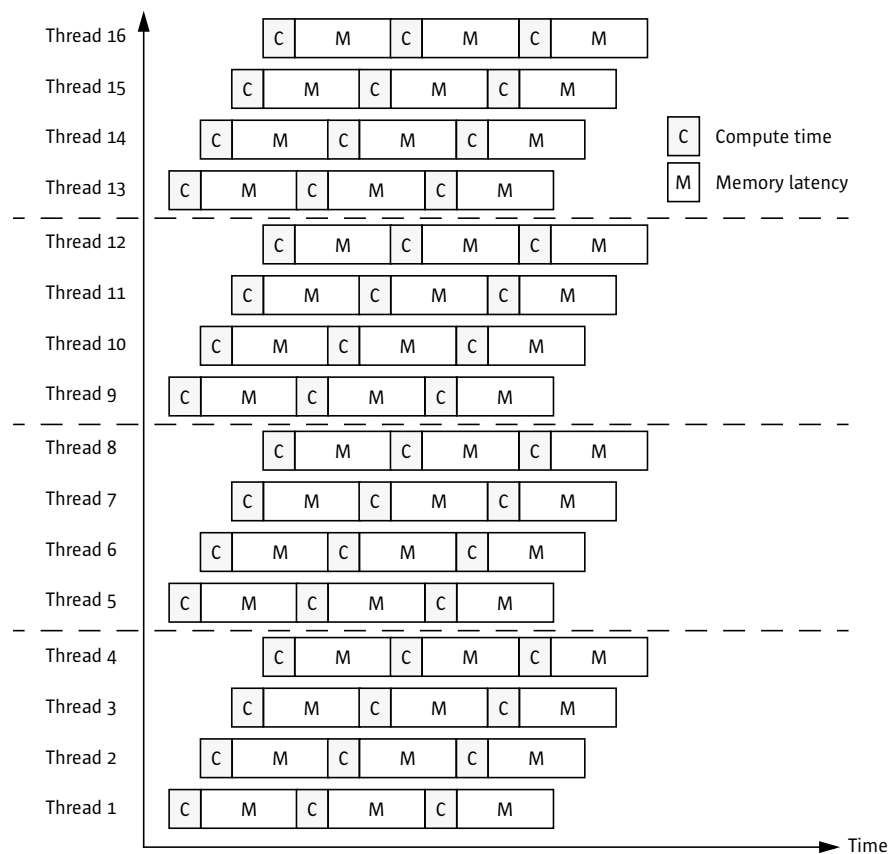


Figure 2. A hypothetical multithreaded processor core switches between a number of threads, doing useful work even while threads block to perform memory-related operations

### Chip Multithreading (CMT)

Many single-threaded processor designs are now available in versions that place multiple processor cores on a single die (so-called chip multi-processing or multi-core processors). This same technique can be used to scale and multiply the benefits of hardware multithreading. Sun calls the result Chip Multithreading (CMT). Unlike complex single-threaded processors, however, CMT processors utilize the available transistor budget to implement multiple hardware-multithreaded processor cores on a single silicon wafer or chip. Because these individual processor cores implement much simpler pipelines, they are also substantially cooler and require less electrical energy to operate. This innovative approach results CoolThread processor technology—multiple physical instruction execution pipelines (one for each core), with several active thread contexts (or strands) per pipeline/core.

Application throughput is greatly improved with CoolThreads, as is the utilization of pipeline resources. Thread-rich applications common in commercial workloads benefit greatly from this model, whether comprised of larger multithreaded applications, or large numbers of smaller single-threaded applications. The number of simultaneous threads that can be accommodated is quite large, and a wide range of processor designs are possible. Figure 3 illustrates a hypothetical example where four hardware multithreaded cores are combined into a single CMT processor, supporting up to sixteen active threads.



## Throughput Computing and Performance

CMT architectures have a number of implications for application performance that are worth noting. The first important factor is that the impact of memory latencies is mitigated greatly over that of single-threaded processors. Because of the processor's ability to rapidly switch between threads, the impact of long memory latencies is effectively masked. Not only are memory latencies less visible, but the processor pipeline in each multithreaded core is utilized much more fully since it is kept busy working on threads that are ready to run. The result is much greater efficiency, since considerably less energy is wasted powering effectively idle processors.

At the same time, it is important to remember that the goal of Throughput Computing is to maximize the overall system throughput, not necessarily the performance of a single thread of execution. As a result, the performance of a single software thread may actually be lower than if it were run on a single-thread processor (depending on the implementation of the CMT processor). Because threads may not have dedicated access to all of the core's resources, but rather share them with other strands, an individual task may ultimately take more processor cycles to complete. In most CMT implementations, the pipeline is also simpler, so the performance of a single software thread can be behind that of a single-threaded processor core, even if the thread (strand) has been given exclusive access to the core.

Because of shared structures like caches, the performance of a thread will also be affected by other threads running on the same core. This effect can be positive or negative:

- If a thread is stalled waiting for memory, its cycles can be directly and immediately used to process other threads that are ready to run.
- If many threads are running the same application (a common occurrence in today's deployments), they can actually benefit from constructive sharing of text and data in the Level-2 cache.
- Though it is a very rare scenario in modern commercial applications, if all of the threads running on the four strands of a core have little or no stall component, then all will receive approximately one quarter of the CPU cycles available within the core.
- If one thread is thrashing a cache or translation lookup buffer (TLB) on a core, other threads on that core can be adversely affected. This scenario is known as "destructive sharing".

## Ideal Applications for CMT Based Systems

CMT processors are designed to scale well with applications that are throughput oriented—namely those that seek to accomplish a large amount of work in aggregate—particularly web and transaction services. A number of classes of applications benefit directly from the ability to scale throughput with CMT processors:

- ***Multithreaded native applications*** — Multithreaded applications are characterized by having a small number of highly-threaded processes. These applications scale by scheduling work through threads or Light Weight Processes (LWPs) in the Solaris Operating Environment. Threads often communicate via shared global variables. Examples of threaded applications include Lotus Domino or Siebel CRM.
- ***Multi-process applications*** — Multi-process applications are characterized by the presence of many single-threaded processes that often communicate via shared memory or other Inter Process Communication (IPC) mechanisms. Examples of multi-process applications include the Oracle database, SAP, and PeopleSoft.

- **Java applications** — Java applications embrace threading in a fundamental way. Not only does the Java language greatly facilitate multithreaded applications, but the Java Virtual Machine (JVM™)<sup>1</sup> is a multithreaded process that provides scheduling and memory management for Java applications. Java applications that can benefit directly from CMT resources include application servers such as Sun's Java Application Server, BEA's Weblogic, IBM's Websphere, or the open-source Tomcat application server. All applications that use a Java™ 2 Platform, Enterprise Edition (J2EE™ platform) application server can immediately benefit from CMT technology.
- **Multi-instance applications** — Even if an individual application does not scale to take advantage of a large number of threads, it is still possible to gain from CMT architecture by running multiple instances of the application in parallel. If multiple application instances require some degree of isolation, Solaris Containers technology can be used to provide each of them its own separate and secure environment.

In the Solaris OS, both software threads and processes are implemented as Light-Weight Processes (LWPs)—the actual entity scheduled by the Solaris OS. These terms are used interchangeably throughout this document. The term strand, on the other hand, refers to the hardware thread contexts that are implemented and scheduled within CMT processors. On systems based on CMT processors, the Solaris OS schedules LWPs (software threads and processes) onto strands (hardware thread contexts).

It is important to note that some applications do not scale well with CMT technology. Long-running single-threaded codes will take longer to complete (unless there is benefit to running multiple instances). Batch applications that have a single thread of execution and severe elapsed time constraints are also not ideally suited to CMT architecture.

1. "The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.

## Chapter 2

# Overview of the UltraSPARC T1 Processor with CoolThreads Technology

Sun's innovative UltraSPARC T1 processor represents a radical new processor design that leverages CMT technology to both drastically improve computational density, and greatly reduce power density. Unlike many disruptive technological breakthroughs of the past, however, the UltraSPARC T1 processor is designed to work seamlessly with existing applications and operating environments. When systems based on the UltraSPARC T1 processor are released, the multithreaded Solaris OS will immediately be ready to take advantage of the additional processor resources while providing high levels of utilization. Existing SPARC applications will simply run without modification due to Sun's Binary Compatibility guarantee.

### The UltraSPARC T1 Processor at a Glance

Sun's new UltraSPARC T1 processor implements CoolThreads by combining chip multi-processing (multiple cores per processor) and hardware multithreading (multiple threads per core) with efficient instruction SPARC V9 pipelines. The result is a processor design that provides multiple, physical instruction pipelines with several active thread contexts (strands) per pipeline.

In particular, a single UltraSPARC T1 processor features up to eight processor cores, or individual execution pipelines per physical chip (Figure 4). Each core supports up to four strands, or active thread contexts. The result is a single processor that provides support for up to 32 active strands at a time. The Solaris Operating System presents the resources of a single UltraSPARC T1 processor as up to 32 logical processors.

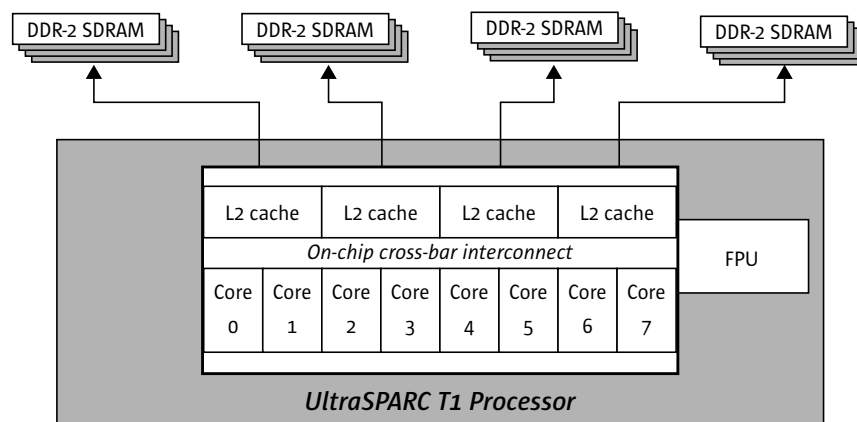


Figure 4. Each UltraSPARC T1 processor features eight processor cores, each with support for four hardware thread contexts (strands)

Each hardware thread or strand is represented by a set of registers that contain the thread's state. On each clock-cycle, the processor is able to switch strands in a round robin ordered fashion, skipping strands that are stalled waiting for a memory access to return. It is the task of the Solaris OS scheduler to schedule LWPs on UltraSPARC T1 hardware strands while the UltraSPARC T1 hardware itself schedules strands within each individual core. As shown

in the illustration, the individual processor cores are connected by a high-speed, low-latency crossbar implemented on the silicon itself. The UltraSPARC T1 processor can essentially be viewed as a symmetric multiprocessing (SMP) system implemented on a single chip, with up to 32 logical processors.

The memory subsystem is implemented as follows:

- Each core has an Instruction cache, a Data cache, an Instruction TLB, and a Data TLB, shared by the four strands.
- Each UltraSPARC T1 processor has a twelve-way associative unified Level 2 (L2) on-chip cache, and each hardware strand shares the entire L2 cache.
- This design results in unified memory latency from all cores (Unified Memory Access (UMA) not Non-Uniform Memory Access (NUMA)).
- Memory is located close to processor resources and four memory controllers provide very high bandwidth to memory, with a theoretical maximum of 20GB per second.

Each core has a Modular Arithmetic Unit (MAU) that supports modular multiplication and exponentiation to help accelerate Secure Sockets Layer (SSL) processing. A single Floating Point Unit (FPU) is shared by all cores, thus the UltraSPARC T1 processor is not an optimal a choice for applications with intensive floating point requirements. It is important to note that there is no coherency circuitry, so a system consisting of multiple UltraSPARC T1 processors running a single Solaris kernel image is not currently possible.

## Solaris 10™ Features for Throughput Computing

One of the most attractive features of systems based on the UltraSPARC T1 processor is that they appear as a familiar SMP system to the Solaris OS and the applications it supports. The Solaris 10 OS has incorporated many features to improve application performance on CMT architectures:

- ***CMT awareness*** — The Solaris 10 OS is aware of the UltraSPARC T1 processor hierarchy so that the scheduler can effectively balance the load across all the available pipelines. Even though it exposes each of the up to 32 individual strands as a logical processor, the Solaris OS understands the correlation between strands and cores.
- ***Fine-granularity manageability*** — The Solaris 10 OS has the ability to enable or disable individual processors. In the case of the UltraSPARC T1 processor, this ability extends to enabling or disabling individual hardware strands. In addition, standard Solaris OS features such as *processor sets* provide the ability to define a group of processors (or strands), and schedule processes or LWPs on them.
- ***Binding interfaces*** — The Solaris OS allows considerable flexibility in that processes and individual LWPs can be bound to either a processor or a processor set, if required or desired.
- ***Solaris Containers*** — Solaris Containers provide fine-grained partitioning, virtualization, and allocation of resources within a given Solaris instance. For example, the resources of a single UltraSPARC T1 processor can be easily partitioned into multiple containers, with each securely supporting a separate Web or application server.
- ***The Hypervisor and the sun4v kernel sub-architecture*** — The UltraSPARC T1 processor features a new hypervisor and identifies itself through the Solaris OS as the *sun4v* kernel subarchitecture. Because the Sun4v kernel subarchitecture is new, applications that query for the system details should recognize *sun4v* as a valid architecture, fully compatible with *sun4u*.

## Applications and the UltraSPARC T1 Processor

Perhaps the best news for application developers is that the introduction of radical CMT technology through the UltraSPARC T1 processor will have a minimal impact on existing applications and the development processes.

- ***Existing applications run without modification*** — CMT technology is simply transparent to applications, and the UltraSPARC T1 processor is fully SPARC V9 compliant. To applications, the underlying platform looks no different than a traditional SMP system, even though its processor resources reside on a single chip. The UltraSPARC T1 processor takes care of the scheduling of the hardware strands onto the cores, presenting each strand as a logical processor to the operating system. The Solaris OS then schedules LWPs (software threads or processes) accordingly. Systems based on the UltraSPARC T1 processor are fully binary compatible with previous SPARC/Solaris environments, and applications will run without modification — *there is no need to make source code changes or even recompile.*
- ***No special software qualification required*** — In spite of the innovative and radically-different architecture of CMT processors, no special software qualification process is required. Because Sun stands behind its binary compatibility guarantee, software vendors need only qualify to the Solaris 10 OS. Solaris 10 qualified applications will then run on any SPARC system that supports the OS, including systems based on the UltraSPARC T1 processor.
- ***Sizing and tuning for maximum scalability*** — At the same time, it is important that applications don't have any inherent limitations to vertical scalability in order to take full advantage of the resources available with the UltraSPARC T1 processor. For example, applications that have only been tested on two- or four-processor systems may reveal existing bottle-necks when run in an environment with up to 32 strands. Sizing and tuning may also be required to take full advantage of CMT architectures. Scalable applications should allow a flexible mechanism to increase the amount of runtime parallelism, either by allowing the number of processes or software threads to be increased, or by facilitating the creation of multiple instances of the application on the same host.

## Chapter 3

# Optimizing Applications for Throughput Computing and CMT Processors

Though existing applications require no re-coding or even recompilation to run on systems based on the UltraSPARC T1 processor, individual application scalability may be improved to leverage the resources available with these systems. This section provides specific advice for removing bottlenecks and improving scalability, applicable to both Throughput Computing in general, and the UltraSPARC T1 processor specifically.

### Sun Studio Compiler Optimization

The UltraSPARC T1 processor is fully binary compatible with all SPARC architectures that Sun supports (namely SPARC V7, V8 and V9). Again, applications do *not* need to be recompiled or modified in any way to run on systems based on the UltraSPARC T1 processor. Developers do not need to support a specific binary for these systems, and binaries created with older versions of the Sun Studio compiler will also work well.

All of the currently-available Sun Studio compiler optimizations will also work well on systems based on the UltraSPARC T1 processor. In general, Sun recommends using profile feedback to improve optimization, taking advantage of a representative workload by gathering run-time information. The `-xprofile=collect` compiler flag followed by `-xprofile=use` can provide substantial performance improvements in most applications for all SPARC based platforms. Profile feedback is described at <http://developers.sun.com/solaris/articles/sol-perf/sol-app-perf.pdf> and the *Sun Studio 10, C User's Guide* also offers details (available from <http://docs.sun.com>). Table 2 shows recommended optimization flags for various versions of the Sun Studio Compilers in case the application is recompiled for the UltraSPARC T1 processor:

*Table 2. Recommended compiler flags for various Sun Studio compiler versions*

Binary type	Compiler	Flags
32-bit	Sun Studio 7, 8, 9, 10	-x04 or -fast -x04 -xtarget = generic
32-bit	Sun Studio 10, Update 1	-x04 -xtarget=ultraT1 or -fast -x04 -xtarget=generic
64-bit	Sun Studio 7, 8, 9, 10	-x04 -xtarget=generic64 or -fast -x04 -xtarget=generic
64-bit	Sun Studio 10, Update 1	-x04 -xtarget=ultraT1 -xarch=generic64 or -fast -x04 -xtarget=ultraT1 -xarch=generic64

The UltraSPARC T1 processor has a relatively small per-core instruction cache, and optimizations that compact the executable are worth investigating. If feedback optimization is already being used, post-optimization via the `-xlinkopt` option is the most effective mechanism to achieve this compaction (details are available at [http://docs.sun.com/source/819-0494/cc\\_ops.app.html](http://docs.sun.com/source/819-0494/cc_ops.app.html).) A `mapfile` can also be generated using the Sun Studio Performance Analyzer (described later in this document).

## JVM Optimizations

A number of optimizations have been added to the Sun JVM that provide advantages for CMT processors and UltraSPARC T1 based systems in particular. The first released JVM built with these optimizations is J2SE 5.0\_06, and it can be downloaded from <http://java.sun.com>. The optimizations are also available today in the JDK 6.0 Binary Snapshot Releases available at <http://www.java.net/download/jdk6/binaries>. While some may be reluctant or unable to move to the latest release of the JVM, every effort should be made to move to version 1.4.2 at a minimum, since it offers parallel garbage collection and the `AggressiveHeap` option.

## Locking Mechanisms

Hot locks represent the most common reason that applications fail to scale to a large number of threads. Locks are needed to protect shared data, and developers tend to be conservative when placing locks around critical sections to reduce the risk of data corruption. Though hot locks might not be encountered on systems with a small number of processors, they may become exposed when running the application on a larger system. The reason for this disparity is that though multiple threads may be enabled by the application, real contention for the lock only takes place when larger numbers of threads actually run in parallel.

With their larger number of available hardware threads and the resulting increased levels of concurrency, a CMT processor such as the UltraSPARC T1 processor can expose previously unknown hot locks in existing applications. However, though it may help expose hot locks, the UltraSPARC T1 processor also has the advantage of faster lock acquisition compared to larger SMP systems, because locks typically reside in the UltraSPARC T1 processor's unified Level-2 cache.

### Tuning Locks

As applications go through multiple tuning cycles, hot locks are identified and typically broken up to protect smaller sections of code. Applications that use their own implementation of locks often provide statistical information about their access, helping performance characterization. For applications making use of Solaris locks, Solaris 10 includes a new utility called `plockstat (1M)`, that reports user-level lock statistics.

Locks are usually implemented as spin-mutexes. When a thread wishes to acquire a lock it first tests (usually via a compare and swap instruction), and if it fails to acquire the lock, the thread spins for some time. Spinning is used to avoid having the thread be context switched off the processor—an expensive operation. On a classic SMP systems, spinning was used to try to increase the utilization of the processor in question, because it was better to burn CPU cycles in the hope the lock would soon become available. On systems powered by the UltraSPARC T1 processor, however, spinning steals cycles from other runnable strands on the same core. In this light, spinning has to be weighed against the disruption that a context switch will have on the other strands being processed by a core.

Setting the right spin count for locks on CMT platforms is an iterative process that is influenced by many different factors, and ultimately depends on the target architecture. Ideally, applications should provide a mechanism to tune the spin count at deployment time, and applications that provide spin and lock statistics greatly facilitate the tuning effort. An application's default setting is usually a good starting point, and in most cases it will be sufficient. If testing shows that system utilization increases with the load, but the throughput remains flat, then the application could be doing excessive spinning, and the spin count should be decreased. When decreasing the

spin count, look for changes in the involuntary context switch rates displayed by `mpstat(1)`. In multithreaded native applications, when locks will not be shared across processes, they should be declared as `PTHREAD_PROCESS_PRIVATE` (or `USYNC_THREAD` for Solaris threads), so that a more efficient and light-weight mechanism is used.

“Thundering herds” is another lock-related issue that can present on systems equipped with the UltraSPARC T1 processor with 32 active threads (and substantially more that are ready to run). Thundering herds occur when the holder of a hot lock relinquishes control and a number of other threads are already spinning and waiting to acquire the same lock. With the lock held in unified Level-2 cache (because it was just freed) the waiting threads all see the freed lock together—responding in unison. This event can lead to unforeseen deadlock situations that are often resolved using an unoptimized backoff codepath. Unfortunately, if all the of the waiting threads are backed off a similar amount of time, they can all return looking for the lock again at the same time (again, as a thundering herd). The effect is periods of reduced throughput on the system.

In these cases, it may be necessary to break the lock into smaller locks, or assign random backoff periods to avoid thundering herds. On the other hand, sometimes removing a lock that is protecting a hot structure can require a significant architectural changes that simply are not possible. For example, the lock in questions might be embedded in a third-party library where the developer has no access or control. In these cases, deploying multiple application instances can often help achieve significant scaling.

### Observing Hot Java Locks

For Java applications, locking primitives are handled differently in various versions of the JVM. From a user perspective the effect of lock contention manifests itself as:

- No increase, or actual decrease in throughput as more load is applied
- Increasing application response time as more load is applied

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()` systems 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(1)`.

If many threads are contending for the same lock, the effect will be more pronounced on systems based on the UltraSPARC T1 processor (effective a large SMP system) as all of the hardware strands will enter the kernel. To preserve CPU cycles for other processes, processor sets can be used to isolate such an application. This approach necessarily restricts the number of threads contending for the lock to those available to the processor set.

From JVM 1.5 onwards, locking has been moved to the Virtual Machine itself. A backoff mechanism has been implemented for hotly-contended locks that translates to more idle time on the system. In JVM version 1.5, therefore, idle levels may increase as more load is applied. There is much ongoing work in Java observability via the Java Virtual Machine Profiler Interface (JVMPPI) and the JVM Tool Interface (JVMTI). These interfaces are only available in later versions of the JVM. This work has been linked with `dtrace` as well. Visit <http://blogs.sun.com/roller/page/ahl> and <https://solaris10-dtrace-vm-agents.dev.java.net/> for more information. The Sun Studio Performance Analyzer also uses this framework in its later releases.

## Large Pages

Processes work with virtual memory in the Solaris OS. When a virtual memory reference needs to be translated to a physical address, a hardware cache called a Translation Lookaside Buffer (TLB) is used to make the translation. As the number of hardware threads increases, so does contention for TLB entries. A miss on the TLB cache is expensive, because several additional steps are needed to map the address.

The unit of translation is the page size, determining the span of memory a single TLB can translate. The larger the page size, the larger the range of memory covered, thus reducing contention on TLB entries. It is possible to use different page sizes for different types of memory segments (e.g. text, heap, stack, anon, shared memory, etc.)

### Large Pages for Native Solaris Applications

The Solaris 10 OS will attempt to use the largest page size possible, so in most cases no specific action is needed to get the benefit of large pages. In any case, it is always a good practice to monitor TLB activity to make sure it is not inhibiting performance. The `trapstat(1M)` utility offers detailed TLB statistics, broken down by strand and page size. In addition, `pmap(1)` can be used to observe the current page size for all memory segments in a process.

It is also possible to request a specific page size explicitly, either at runtime by using Multiple Page Size Support (MPSS) or `ppgsz`, or during development by using the `memcntl` API. See the `mpss.so.1(1)`, `ppgsz(1)` and `memcntl(2)` manual pages for more information. It is important to keep in mind that such requests do not guarantee an actual page size, since other factors like segment size and alignment come into play. The Solaris OS will, however, attempt the next-best fit if the requested page size is not available.

For many years, the Solaris OS has featured a performance optimization called Intimate Shared Memory (ISM) that allocates on large pages where possible, and locks down memory. The default page size on systems with the UltraSPARC T1 processor is 256MB. ISM has been further enhanced to enable its dynamic reconfiguration, known as Dynamic ISM (DISM). Please see the `shmat(2)` manual page for more details. To enable ISM add `SHM_SHARE_MMU` to the shared memory attach flags. DISM is achieved by adding `SHM_PAGEABLE` to the attach flags. Note that unlike ISM, DISM requires backing swap space.

### Large Pages for Java Applications

From JVM version 1.4.2 onwards, the `-XX:+AggressiveHeap` option has been available. One of the affects of this option is to put the Java heap on 4MB pages. To achieve maximum performance with this option, it is best to pre-allocate the heap using the options `-Xmx<size>m -Xms<size>m`. The following example will attempt to pre-allocate 3.8GB on 4MB pages. As always, check that the pages were allocated with `pmap -xs`.

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

## Critical Sections

On a CMT processor, a thread may take longer to pass through a critical section than on a single-threaded processor, especially if there are other active threads sharing cycles on the core. Critical sections are often protected using locks, so the users may be able to detect longer hold times in the application's lock statistics. Performance gains may also be achieved by shortening the length of each critical section. This goal can be achieved in a number of ways:

- By breaking up the structure and creating a number of locks for each of the sections
- By creating per-CPU structures that then scale the associated locks
- By reducing the amount of global data In native multithreaded applications (with the introduction of Thread Local Storage (TLS), Solaris threads can have high performing per-thread data)

Where possible to enhance threading and synchronization, Java developers are advised to use the `java.util.concurrent` package introduced in J2SE 5.0. Many of the data-structures provided by that package are lock-free and scale well on large systems such as those powered by the UltraSPARC T1 processor. For more information, please reference: <http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html>

Another mechanism to enhance the performance of critical sections can be found in the `schedctl` APIs (see the `schedctl_init(3SCHED)` manual page for details). These calls allow a running thread (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.

## Polling and Asynchronous Operations

Polling is often used in applications to help ensure response time requirements. In this scenario, a thread (LWP) is dedicated to constantly check for incoming requests, responses, and other events. Polling on a traditional processor is an acceptable option, as there is only a single thread of execution for the scheduler to manage. On a traditional processor, other LWPs in the dispatch queue will prevent the polling thread from burning CPU cycles. Polling in a CMT architecture is an undesirable option, as it occupies CPU cycles that could be used by other threads scheduled on a core. In a thread-rich environment the polling thread will also run far more often.

A useful alternative to polling is the Event Ports facility available in the Solaris 10 OS. For more information, see the `port_create(3C)`, `port_get(3C)`, and `port_send(3C)` manual pages. 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. (See also [http://developers.sun.com/solaris/articles/event\\_completion.html](http://developers.sun.com/solaris/articles/event_completion.html))

Similarly, kernel asynchronous I/O (`kaio`) can be used to avoid burning excessive CPU cycles. For more information, see the `aio_read(3RT)`, `aio_write(3RT)`, `aio_waitn(3RT)` and `aio_suspend(3RT)` manual pages. Lists of I/O can also be submitted with `lio_listio(3RT)`. Rather than blocking, I/O operations can be reaped asynchronously.

## Performance Tools

A variety of tools are available to help gain insight into performance-related issues on CMT systems.

### Sun Studio Performance Analyzer

The Sun Studio Performance Analyzer can help assess code performance, identify potential performance problems, and locate the section of the code where problems are occurring. The Performance Analyzer can be used from the command line or from a graphical user interface. The Analyzer consists of two tools:

- **The Collector** collects performance data by profiling and tracing function calls. The data can include call stacks, micro-state accounting information, thread-synchronization delay data, hardware-counter overflow data, MPI function call data, memory allocation data, and summary information for the operating system and the process. The Collector can collect all kinds of data for C, C++, and FORTRAN programs as well as profiling applications written in Java.
- **The Performance Analyzer** itself displays the data recorded by the Collector. The Analyzer processes the data from the Collector and displays various metrics of performance at the level of the program, the functions, the source lines, and the instructions. These metrics are classed into five groups: timing metrics, hardware counter metrics, synchronization delay metrics, memory allocation metrics, and MPI tracing metrics. The Analyzer also displays the raw data in a graphical format as a function of time. See the Sun Studio Performance Analyzer documentation for more details.

### Solaris Dynamic Tracing (DTrace)

When production systems exhibit nonfatal errors or sub-par performance, the sheer complexity of modern distributed software environments can make accurate diagnosis of the root cause extremely difficult. Unfortunately, most traditional approaches to solving this problem have proved time-consuming and inadequate, leaving many applications languishing far from their potential performance levels.

The Solaris DTrace facility provides dynamic instrumentation and tracing for both application and kernel activities—even allowing tracing of application components running in a Java Virtual Machine. DTrace enables developers and administrators to explore the entire system to understand how it works, track down performance problems across many layers of software, or locate the cause of aberrant behavior. Tracing is accomplished by dynamically instructing the operating system kernel to record additional data at locations of interest. Best of all, although DTrace is always available and ready to use, it has no impact on system performance when not in use, making it particularly useful in monitoring and analyzing production systems.

### Solaris Commands

A variety of commands are available for analysis and performance tuning from the Solaris command line. Please reference the respective manual page entries to learn more about the following Solaris commands:

- mpstat(1M)
- vmstat(1M)
- prstat(1M)
- ps(1M)
- cpustat(1M)
- cputrack(1M)
- busstat(1M)
- truss(1)
- trapstat(1M)
- plockstat(1M)
- psrset(1M)
- pbind(1M)
- psradm(1M)
- psrinfo(1M)
- pmap(1)
- kstat(1M)

## Chapter 4

# Conclusion

Sun's Throughput Computing strategy is designed to directly address the needs of organizations for flexible and scalable performance while drastically improving energy consumption profiles in the data center. CMT support in the revolutionary UltraSPARC T1 processor with CoolThreads technology provides massive amounts of thread-level parallelism while effectively masking the dominant effect of memory latency on throughput and workload performance. With well-written applications and execution platforms such as the Solaris OS and Java Virtual Machine, these performance benefits can be delivered directly to applications in terms of greatly-enhanced computational throughput.

As a complete systems vendor with leading processor, operating system, and developer solutions, Sun is in the unique position of being able to make the transition to revolutionary CMT processor technology with virtually no negative impact to existing applications or operations. In fact, existing SPARC/Solaris applications simply run on systems with UltraSPARC T1 processors with no modification of any kind. At the same time, these systems present virtually unprecedented opportunities for scalability, and steps can be taken to help applications take advantage of the considerable performance gains made available through CMT technology.

## Chapter 5

# References

Sun Microsystems posts product information in the form of data sheets, specifications, and white papers on its Internet World Wide Web Home page at: <http://www.sun.com>. Look for the these and other Sun technology white papers:

*“Application Development and Tuning on UltraSPARC T1 CMT Systems”*, white paper, Denis Sheahan, Sun Microsystems, Inc.

## Chapter 6

### Glossary

- **Chip multi-processing (CMP):** Also known as multi-core processors, chip multi-processing places multiple processor cores on a single chip.
- **Chip multithreading (CMT):** Chip multithreading as defined by Sun combines hardware multithreading (multiple threads per processor core) with chip multi-processing (multiple processor cores per chip) to yield hardware support for large numbers of thread contexts.
- **Core:** The circuitry required to implement a processor's execution pipeline.
- **Crossbar:** A interface where everything connected to the crossbar has a direct connection to everything else. The UltraSPARC T1 processor features an on-chip crossbar that connects processor cores to the L2-cache.
- **Data cache (D-cache):** Fast memory (Level-1 cache) close to the processor pipeline for caching data. Each of up to eight cores in the UltraSPARC T1 processor has its own data cache.
- **Floating Point Unit (FPU):** An FPU provides acceleration for the execution of floating-point instructions.
- **Instruction cache (I-cache):** Fast memory (Level-1 cache) close to the processor for caching instructions. Each of up to eight cores in the UltraSPARC T1 processor has its own instruction cache.
- **Level-2 cache (L2 cache):** A second hierarchical level of cache positioned between the Level-1 caches and memory. Each UltraSPARC T1 processor has a 12-way associative unified on-chip L2 cache.
- **Light-Weight Process (LWP):** The schedulable entity in the Solaris OS. Both software threads and processes are implemented as LWPs.
- **Modular Arithmetic Unit (MAU):** An MAU accelerates modular arithmetic, including tasks such as encryption and decryption needed for Secure Sockets Layer (SSL) processing.
- **Non-Uniform Memory Access (NUMA):** A NUMA architecture implies that memory access time is dependent on the memory location. For example, in some multi-processor systems, each processor controls a bank of memory so memory access times are different for the processor to access the memory it controls as compared to memory that another processor controls.
- **Pipeline:** The portion of a processor core that executes instructions.
- **Strand:** Strands represent a hardware thread context, and are scheduled by the UltraSPARC T1 processor.
- **Symmetric Multi-Processing (SMP):** In multi-processor architectures, an SMP system consists of two or more processors that are connected to a single shared system memory. With up to eight processor cores and up to 32 hardware thread contexts, the UltraSPARC T1 processor is an example of an SMP system on a chip (SOC).
- **Translation Lookaside Buffer (TLB):** When a virtual memory reference needs to be translated to a physical address, a hardware cache called a Translation Lookaside Buffer (TLB) is used to make the translation.
- **12-way associative:** A set-associative cache combines direct-mapped and fully-associative cache designs. A 12-way set associative cache is divided into sets where each set contains 12 cache lines. Within each set, the cache is associative.
- **Uniform Memory Access (UMA):** A UMA architecture implies that all CPU pipelines have a uniform distance and access time from memory. Systems built from the UltraSPARC T1 processor represent UMA architectures.



Copyright © 2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A.

All rights reserved.

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.

Sun, Sun Microsystems, the Sun logo, CoolThreads, Solaris, Java, JVM, and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a). 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 HELD TO BE LEGALLY INVALID.