Solaris™ Memory Placement Optimization and Sun Fire™ Servers

Technical White Paper

March, 2003

Printed in the United States of America.
4150 Network Circle, Santa Clara, CA 95054 U.S.A

Please
Recycle

Contents

Introduction 1

The latest version of the Solaris™ Operating System has several performance and scalability improvements in a variety of areas: improved threading library, support for multiple page sizes, UFS concurrent direct I/O, advanced page coloring, to name a few.

This document describes one of those improvements, called Memory Placement Optimization, first introduced in update 9/02 of the Solaris 9 Operating System. Its purpose is to improve the placement of memory pages across the physical memory of server, resulting in increased performance.

Chapter 2 describes the Memory Placement Optimization technology, and the specific variables and programming interfaces that were created in the Solaris 9 Operating System.

Chapter 3 demonstrates the benefits of utilizing the Memory Placement Optimization feature, by discussing the results of tests on a variety of workloads and benchmarks.

Appendix A lists code examples that use Memory Placement Optimization commands and variables.

In order to understand how Memory Placement Optimization works, and its value, it is useful to study the design of the Sun Fire™ servers, including their memory latency and bandwidth characteristics. Appendix B in this white paper, as well as References [1] to [6], describe the subject in detail. A brief overview is provided next.

Sun Fire Servers

The UltraSPARC® III-processor-based Sun Fire server family uses common technology and components to create efficient shared memory processing (SMP) systems at different sizes and capacities.

The basic building blocks for these servers are system devices like processors, memory units and I/O controllers. The system devices are connected through the Sun Fireplane system interconnect, which has a crossbar switch design based on a point-to-point protocol. As the number of system devices increases, so does the sophistication and capacity of the Sun Fireplane interconnect implementation.

The Sun Fireplane-based servers can be divided into four categories by the number of interconnect levels required:

1. **Small server**. Four system devices (two processors, one memory unit, and one I/O controller) require only one level of interconnect.
2. **Workgroup server**. 18 system devices (eight processors, eight memory units, and two I/O controllers) require two levels of interconnect.
3. **Mid-size server**. 56 system devices (24 processors, 24 memory units, and eight I/O controllers) require three levels of interconnect.
4. **Large server**. 180 system devices (72 processors, 72 memory units, and 36 I/O controllers, or 106 processors, 72 memory units and 2 I/O controllers) require four levels of interconnect.

The Sun Fire servers are constructed by placing system devices into physical components, like boards or assemblies. Processors and memory devices share a single board, called a Uniboard in the case of the Sun Fire 3800-6800 (mid-size) servers and Sun Fire 12K and 15K (large) servers. A Uniboard contains up to 4 UltraSPARC III processors, and up to 32 GB of memory distributed into 16 logical banks. A unique feature of the UltraSPARC III processor and Sun Fireplane interconnect design is that the memory control units are located inside the processors. Each processor controls 4 logical banks.

The three-level interconnect in a server like the Sun Fire 6800 combines up to 6 Uniboards, and up to 4 I/O assemblies, in a 10x10 crossbar switch.

For a server like the Sun Fire 15K, the four-level interconnect combines up to 18 Uniboards (the same used for the Sun Fire 3800-6800 servers) and up to 18 I/O assemblies in a 18x18 crossbar. There is also an option to exchange I/O

assemblies for boards called MaxCPU, which contain only two UltraSPARC III processors, with no associated memory. In that case, the Sun Fire 15K server can support up to 106 processors (distributed into 18 Uniboards and 17 MaxCPU boards) if only one I/O assembly is installed.

Memory Latency and Bandwidth

In terms of memory performance, the important characteristics of a server are latency and bandwidth. Latency measures the speed in which data can be brought from memory to the processors (low latency means fast access), and bandwidth measures how much data can be moved in a certain interval (high bandwidth means large data-transfer rates).

The design target for the Sun Fireplane interconnect was a balanced system in which available bandwidth increased with the server size, while latencies were kept low across the system. Those are essentially the qualities that define an efficient SMP architecture. The Sun Fireplane interconnect builds on the success of the Sun Gigaplane™ and Sun Gigaplane-XB™ designs which led the previous generation of Sun data center servers, the Sun Enterprise™ 3500-6500 and 10000 systems, to be recognized as industry-leading SMP servers.

SMP implementations generally demonstrate some memory locality effects, which means that when a processor requests access to data in memory, that operation will occur with somewhat lower latency if the memory bank is physically close to the requesting processor. In the case of Sun Fire systems, the UltraSPARC III processor presents an innovative design that places the memory controller inside it, allowing for a fast, efficient, and scalable memory coherency implementation. Such a design causes a processor to experience faster access to the memory banks that are connected to it.

Small and workgroup Sun Fire servers are so compact that memory locality effects are basically negligible. The architecture of the mid-size and large servers, like the Sun Fire 3800-6800 and 12K/15K systems, present memory locality characteristics that can be explored in order to extract higher performance from the SMP architecture.

The mid-size and large Sun Fire servers provide industry leading performance using the Solaris 8 Operating System, which is not aware of memory locality effects. Nevertheless, Sun has invested in optimizing further the operating system's method for memory allocation. The Memory Placement Optimization feature allows the Solaris 9 Operating System to intelligently place memory

pages and processes throughout the SMP server in order to improve memory latencies and bandwidth, providing more performance and value to the Sun Fire products.

Memory Placement Optimization 2

Sun Fire servers offer a scalable, efficient architecture that enables high performance in a shared memory programming model. However, there are substantial performance gains to be realized if the Solaris Operating System takes advantage of the smaller memory latencies, and higher memory bandwidth, obtained with careful placement of memory pages.

As seen in the previous chapter, Sun Fire mid-size and large servers have some elements of locality in memory access. The Memory Placement Optimization (MPO) feature, introduced in the Solaris 9 9/02 Operating System, takes advantage of such locality in order to improve the performance of user applications.

MPO technology, in general, increases the performance of individual applications over that seen on previous releases of the Solaris Operating System. In addition to improving individual job performance, it increases overall system throughput for multiple applications.

With the addition of MPO technology to the Solaris Operating System kernel, the system is better able to explore the bandwidth within different levels of the system interconnect. The kernel is also able to take advantage of the fact that memory located near a processor can be accessed more quickly than memory which is located across the system's Sun Fireplane interconnect. The end result: higher throughput on a wide variety of workloads, from business computing, like decision support and transaction processing, to high performance technical computing (HPTC) applications.

Although such gains can be substantial, they are highly dependent on the workload characteristics. This chapter will discuss how the MPO feature is implemented in the Solaris Operating System, and the next chapter will show some examples of workloads that benefit from it.

Implementation

The MPO overall strategy is to optimize for performance through latency and bandwidth, attempting to ensure that memory is *as close as possible* to the processors that access it, while still maintaining enough balance within the system to avoid the introduction of bottlenecking hotspots.

Locality Groups

Before making any policy decisions to explore memory locality, the operating system needs some way to model the underlying structure of the hardware in order to determine which processors and memory are close to one another. In the Solaris Operating System, this structure is represented by locality groups, or `lgroups`.

An `lgroup` is a subset of a machine in which all components can access one another within a bounded latency interval. Currently, `lgroups` include only processors and memory that are considered to be co-local to one another. In other words, cache-to-cache transfers between any two processors in the group will take roughly the same amount of time, and that time will be less than the time required to retrieve data from the cache of a processor in a different `lgroup`. In addition, all of the memory in the `lgroup` can be accessed in roughly the same amount of time from each processor within the group, and that time is less than the time required to access memory in any other `lgroup`.

In the simplest case, all of the processors and memory in the system are in a single `lgroup`. This case applies to Sun's pre-Sun Fire servers, all of which have architectures with negligible locality effects.

In the next simplest case, the Sun Fire 3800 to 6800 servers, an `lgroup` consists of the CPUs and memory on a single Uniboard. The system may contain multiple Uniboards, and consequently multiple `lgroups`. In this case, all cache-to-cache transfers take the same amount of time, as do all memory-to-cache transfers.

A more complicated case occurs for the Sun Fire 12K and 15K servers. In this design, an `lgroup` may be just a single Uniboard, thus offering the same characteristics as an `lgroup` on one of the smaller servers. The system will in general contain multiple Uniboards, and consequently multiple `lgroup`s. However, when a Sun Fire 12K or 15K server is configured with MaxCPU boards, an `lgroup` will consist of all the processors and memory on both boards in the same expander boardset (Uniboards, IO assemblies and MaxCPU boards are all connected to expander boards; a combination of a Uniboard plus IO assembly, or Uniboard plus MaxCPU board, is called a boardset – see Appendix B for details). In this case, the latencies within a single `lgroup` will not be strictly uniform, but they are not great enough to warrant the creation of two different `lgroups` for the same expander boardset.

Scheduler

When the Solaris Operating System creates a new light-weight process (LWP), it is assigned a “home `lgroup`”. This home `lgroup` is used to minimize the frequency with which a thread moves from one board to another. The selection of a home `lgroup` is based upon the number of LWPs in the process, the size and relative loads of each `lgroup` in the system, and which `lgroups` that process is spread across. The LWP will have an affinity for its home `lgroup` and will tend to run and allocate its memory there.

The dispatcher will always try to run the LWP on its home `lgroup` if possible. If all of the CPUs on the home `lgroup` are busy and running higher priority threads, it will try to find the best place to run the LWP outside of the home `lgroup`. Running the LWP on an `lgroup` other than its home is referred to as “running remotely”. Even if an LWP runs on a remote `lgroup`, its home `lgroup` will remain unchanged. The next time the thread is scheduled, it will try to return to its home `lgroup` if a CPU is available.

Dispatching an LWP to its home locality group as often as possible serves several purposes. It reduces the number of times the LWP has to access memory on a remote board. In addition, locality-aware scheduling reduces the number of inter-board cache-to-cache transfers. Avoiding remote cache transfers provides for a faster ramp-up time should an LWP be “migrated” from one CPU’s run queue to another. This sort of CPU migration occurs frequently in transaction processing type workloads, which run with thousands of LWPs that frequently sleep waiting on I/O.

There are two ways in which an LWP's home `lgroup` can change. The most obvious way an LWP's home `lgroup` can change is if all the processors in its home `lgroup` are removed from the system, either through off-lining or a dynamic reconfiguration operation. The other way that an LWP's home `lgroup` can change is when that LWP is bound to a processor in a different `lgroup`. In that case, the new processor's `lgroup` becomes the LWP's new home. Even if the LWP is subsequently unbound, it will retain this new home `lgroup`.

Note that scheduling affinity is not done for realtime threads, since the implementation is POSIX conformant. Hence, jobs should be placed into timeshare (TS), interactive (IA), fixed priority (FX), or fair share (FSS) scheduling classes in order to benefit from MPO.

Memory Allocation

In the Solaris Operating System, memory allocation is a two step process. The first step assigns virtual memory. This step occurs when an application calls `brk()` to extend its heap or when it maps in a file. The second step is to assign physical memory to back the virtual memory. The assignment of physical memory does not occur until the application first tries to read or write to the new virtual address. At that point, the Solaris Operating System will select a physical memory page and create a mapping from the application's virtual address to this physical page.

The key to delivering the best performance on systems with sizeable memory locality differences is to ensure that physical memory is allocated close to the threads that are expected to access it. This allows for both lower latency and higher bandwidth. Obviously, when a process allocates memory, the operating system cannot predict with certainty how that memory will be used in the future. However, there are several different assumptions that the operating system can make that hold in many cases.

The simplest policy that one can adopt when allocating memory for locality awareness is "first touch". This simply means that memory is allocated from the home `lgroup` of the LWP that first tries to access that memory. This approach assumes that whichever thread first accesses the memory is likely to be the thread that will access that memory most frequently in the future. This assumption obviously holds for single-threaded applications, but it also holds for many multi-threaded applications as well. "First touch" is the default memory allocation policy used for private memory. Note that memory is

allocated from the LWP's home `lgroup`, even if the LWP is running remote from its home at the time the memory is allocated. This behavior reflects an assumption that the LWP will primarily be scheduled to run on its home `lgroup`.

Shared memory (e.g., Intimate Shared Memory (ISM), `MAP_SHARED` `mmap()`'ed files, etc.) is, by definition, likely to be accessed by multiple threads. Assuming that some significant number of those threads will be running in different `lgroups`, the Solaris Operating System allocates shared memory using a random memory placement policy by default. This policy optimizes for bandwidth while trying to minimize average latency for the threads accessing it throughout the server. It spreads the memory across as many memory banks as possible, distributing the load across many memory controllers and bus interfaces, preventing any single component from becoming a performance-limiting hotspot. In addition, random placement improves the reproducibility of performance measurements, by ensuring that the relative locality of threads and memory remains roughly constant across multiple runs of an application.

System Variables

Most of the locality-related optimizations introduced with MPO rely on some fairly simple heuristics, to provide good performance for most applications. It is possible that some applications that do not behave as expected will experience some performance problems with this new functionality. In case of any such issues, the values of the MPO internal system variables can help explain the system behavior, while the controlling APIs described in the next section can help provide a solution.

Important:

The description of the MPO system variables is provided here solely with the purpose of explaining the MPO implementation. Changes to these variables are not supported, and customers experiencing any problems may be required to change the variables back to their default values for proper diagnostics.

Users should keep in mind that these variables are all internal Solaris Operating System variables, and do not constitute a formal interface. Although the commands and variables below are implemented into current releases of the Solaris 9 Operating System, these variables may change or disappear over

time. In addition, since these are internal variables, there may be no error detection should they be changed to unexpected values. Their default values have been carefully chosen to work well together.

First Group

`lgrp_mem_default_policy`

This variable reflects the default memory allocation policy used by the Solaris Operating System. This variable is an integer, and its value should correspond to one of the policies listed in `<sys/lgrp.h>`. On Sun Fire 3800-6800 servers, this value is `LGRP_MEM_POLICY_NEXT`, starting with the Solaris 9 9/02 Operating System, indicating that memory allocation will default to “first touch”. On Sun Fire 12K and 15K servers, this value is:

- `LGRP_MEM_POLICY_RANDOM` in the Solaris 9 9/02 Operating System, indicating that one defaults to random allocation;
- `LGRP_MEM_POLICY_NEXT` starting with the Solaris 9 12/02 Operating System, indicating that one defaults to “first touch” allocation. However, on Sun Fire 12K and 15K servers without the hardware pre-requisite installed, all processors and memory will be placed in a single `lgroup`, essentially disabling the MPO feature.

See the section entitled “MPO in the Solaris Operating System” for details on MPO availability and hardware pre-requisites on Sun Fire servers.

`lgrp_shm_random_thresh`

As described above, large shared memory regions are allocated randomly rather than using “first touch”. This variable provides control over how large can a region be before we switch to random allocation. The default is 8MB, which is large enough to allow communication buffers such as those used by MPI programs to be local to one of the ends of the communication pipe; yet, it is small enough that memory regions which are likely to become hot spots will be spread across the system’s memory controllers.

This variable is an unsigned 64-bit integer, which may be modified at runtime using a kernel debugger, or via `/etc/system`.

lgrp_mem_pset_aware

If a process is running within a user processor set (see `psrset(1M)`), this variable determines whether “randomly” placed memory for the process is selected from among all the `lgroups` in the system or only from those `lgroups` that are spanned by the processors in the processor set. This value defaults to zero, indicating that the Solaris Operating System will select memory from all the `lgroups` in the system. This default is appropriate for systems where processor sets are not used or are only used to isolate applications from operating system threads. If processor sets are used to isolate applications from one another, then setting this value to one will likely lead to more reproducible performance.

lgrp_expand_proc_thresh

This variable controls how quickly a process’ LWPs will spread across multiple `lgroups`. If the lowest load among all the `lgroups` across which the process is spread exceeds this threshold, that suggests that our current `lgroups` are all approaching or exceeding their capacity. Thus, we will consider placing the next LWP on a new `lgroup`.

This value reflects the fraction of an `lgroup`’s capacity that is being used. To allow the Solaris Operating System to evaluate loads using only integer arithmetic, this value is an unsigned 32-bit integer that is set to `INT16_MAX` times some fractional capacity.

On Sun Fire 12K and 15K servers, this value defaults to `(INT16_MAX*3)/4`, indicating that we will not consider spreading a process to a new `lgroup` until each of its existing `lgroups` is at least 75% loaded. On Sun Fire 3800 to 6800 servers, this value defaults to (`INT16_MAX/4`), indicating that we will consider spreading to a new `lgroup` if our existing `lgroups` are at least 25% loaded. The different values arise from the differences in architecture between the two servers. On Sun Fire 12K and 15K servers, the remote latency is significantly higher than the remote latency on Sun Fire 3800-6800 servers, and, conversely, the available bandwidth is much greater. Thus, these values reflect an attempt to manage load to minimize an application’s latency on a Sun Fire 12K/15K server and maximize an application’s bandwidth on Sun Fire 3800-6800 servers.

Second Group

lgrp_privm_random_thresh

As described above, by default, private memory is always allocated by “first touch”. This variable makes it possible to allocate large private memory regions using random placement rather than “first touch”. By default, this value is `ULONG_MAX`.

This variable is an unsigned 64-bit integer, which may be safely modified at runtime using a kernel debugger, or via `/etc/system`.

lgrp_expand_proc_diff

Once we have decided to spread a process out to a new `lgroup`, there is no point in spreading it to a new `lgroup` that is just as loaded as the `lgroups` we are already running on. This variable uses the same capacity units as `lgrp_expand_proc_thresh`, and it indicates how much lower the load must be on a new `lgroup` before we will assign a new LWP to that `lgroup`. On both Sun Fire 3800-6800 and 12K/15K servers, this value defaults to `(INT16_MAX/4)`, or a 25% difference in load.

lgrp_loadavg_tolerance

As with system load, an `lgroup`’s load is calculated using a decaying average function, which tends to be more useful than the “instantaneous” load measurement which can fluctuate widely and quickly. Thus, the load value for an `lgroup` is really only an estimate, which is constantly changing. When this value is actually used to decide which `lgroup` a new thread should be placed on, `lgrp_loadavg_tolerance` is used as a “fudge factor”. If the current estimated loads on two `lgroups` are within `lgrp_loadavg_tolerance` of one another, we treat those `lgroups` as being identically loaded, and choose randomly between them. The value is specified using the same units as the other load variables. The default value is `0x10000`, which leads to good performance results for a variety of database and mixed workloads. Our tests have shown that HPC workloads frequently benefit from a lower value, such as `0x1000`.

Informational and Controlling APIs

In order to help the developer optimize an application's performance by exploring MPO technology, several new APIs have been added to the Solaris Operating System.

Informational APIs

It is not always easy to identify potential memory-locality-related problems simply by studying an algorithm in isolation. Furthermore, the use of auto-parallelizing compilers can introduce memory locality problems that do not exist in the serial algorithm. The APIs in this section allow an application to dynamically determine how its threads and virtual memory have been assigned to processors and physical memory by the Solaris Operating System. The following is a high-level description of each of the new APIs. The full details of each can be found in the `man` pages starting with the Solaris 9 Operating System.

`getcpuid(3C)`

This routine will return the `cpuid` on which the calling thread was running when it executed the call. Unless a thread is bound to a CPU, the Solaris Operating System is free to schedule it on any CPU in the system (but following `lgroup` policies). Hence, there is no guarantee that a thread will still be running on this CPU.

`gethomelgroup(3C)`

This routine returns the `ID` of the home `lgroup` of the calling thread. A thread's home `lgroup` is a much less transitory value than the current CPU `ID`. Once a thread is assigned a home `lgroup`, that `lgroup` will not change unless the thread is explicitly bound to a CPU in a different `lgroup`, or if all the CPUs in the `lgroup` are taken offline. Note that this permanence may not continue to be true in future releases of the Solaris Operating System. It is possible that, in the future, threads will eventually migrate from one `lgroup` to another in response to system utilization and migration policies.

meminfo(2)

The meminfo(2) system call allows one to query the operating system about both virtual and physical memory assigned to the calling process. Given a virtual address in the calling process's address space, this call can return the physical address, the lgroup to which that physical address belongs, and the size of the page. Given a physical address, the call can return the lgroup in which the memory exists.

This call is useful for diagnostic and verification purposes. Knowing where a range of memory is physically stored can help explain why accesses to that memory take longer than expected. This information can then be used to determine where, or if, calls to madvise(3C)(see next section) might allow the Solaris Operating System to make better decisions about where memory should be allocated. Once calls to madvise(3C) have been added, the meminfo(2) call can be used to verify that the Solaris Operating System has made the expected changes in its behavior.

Controlling APIs

The goal of MPO technology in the Solaris Operating System is to deliver good performance on servers with memory locality properties, without making any changes to the applications. However, there are some applications that could achieve better performance by improving the operating system's default placement policies.

For example, an application in which one thread allocates and initializes a large dataset from private memory will likely have all of its memory located on a single lgroup. If the application then spawns many new threads to access that data, a significant number of those threads are likely to be running on remote lgroups. Rather than making extensive modifications to an application, the following API provides a relatively easy method for improving such an application's performance. While madvise(3C) is easy to use, using its MADV_ACCESS flags has some overhead. Consequently, optimal performance can be obtained by simply having each thread initialize its own data for this example. This means that, for some applications, it may be the case that optimal performance will require that the application be restructured so that each thread initializes and uses a limited portion of the full dataset.

madvise(3C)

This routine allows an application to provide the Solaris Operating System with hints about how it expects a range of memory to be used. Specifically, it allows an application to indicate whether a range of memory will be used by many LWPs (MADV_ACCESS_MANY) or by the next LWP that touches it (MADV_ACCESS_LWP).

The MADV_ACCESS_MANY hint may be used by an application that creates and initializes a large data structure in private memory, and then creates multiple threads that will all access that data structure. This behavior is typical of many auto-parallelized applications. Since the data structure is created while the application has only a single thread, by default the Solaris Operating System will attempt to allocate it all on a single Uniboard. This hint will prompt the Solaris Operating System to allocate the data structure using a random placement policy, which will offer higher bandwidth to all the application's LWPs.

The MADV_ACCESS_LWP hint is most useful when an application changes how it expects a range of memory to be used. After receiving this hint, if the next LWP to touch a page in the specified range is in a different `lgroup` than the memory, then the Solaris Operating System may migrate the page to that LWP's `lgroup`. This can be useful for applications that have multiple phases, each with distinctly different memory usage patterns. It can also be used for applications that allocate a large ISM segment in order to get large pages, but do not intend to share those pages with other threads. Note that migrating memory can be a very time-consuming operation, so MADV_ACCESS_LWP and MADV_ACCESS_MANY should be used with discretion.

madv.so.1

`madv.so.1` is a shared object that interposes on the memory allocation system calls, and allows the user to apply the hints described above without modifying the source code of the application. This functionality is less precise than that offered by the `madvise()` interface, as one cannot choose to apply the advisement to specific address ranges. This object allows one to apply advisement to the whole heap, just ISM or Dynamic ISM (DISM) segments, just private segments and so on. This functionality is most useful for rapid prototyping and for tuning applications for which the source code is not available.

MPO in the Solaris Operating System

The Solaris 9 9/02 Operating System introduced MPO by creating `lgroups`, changing the scheduler to be locality-aware, and implementing “first touch” memory allocation. Also available were the MPO informational APIs and system variables in the “First Group” as defined above.

The Solaris 9 12/02 Operating System made the controlling APIs available to users that desire to fine-tune the MPO functionality. Also available were the system variables in the “Second Group”.

Starting with the Solaris 9 9/02 Operating System, MPO is enabled by default on Sun Fire 3800-6800 servers. There are no hardware pre-requisites.

Default activation of MPO for Sun Fire 12K and 15K servers requires the Solaris 9 12/02 Operating System (or later) and a hardware pre-requisite. On February 2003, Sun announced a “MPO hardware upgrade kit” for existing systems. At that date, Sun also announced the availability of new Sun Fire 12K and 15K server models that are already enhanced for MPO.

Memory Placement Optimization Benefits 3

The main benefits of using Memory Placement Optimization are lower memory latencies and increased bandwidths for memory-intensive applications, resulting in higher performance.

The improvements that can be observed are highly dependent on the application's memory access patterns, and although MPO policies have been developed using a representative set of real workloads, the specific results of MPO introduction may vary significantly. Every effort was made to make sure that, even in the worst case, the default MPO settings would not decrease the performance of an application.

This chapter presents results of the introduction of MPO on a variety of business and scientific workloads, and serves only as a rough indication of the performance gains that can be expected. Users should therefore use caution when estimating any performance gains provided by the MPO feature in their specific workloads; actual tests are recommended for better understanding of each application characteristics.

The tests shown were performed in the Sun benchmarking lab environment, using production and prototype equipment. Baseline results (without the MPO feature) refer to tests using the Solaris 9 Operating System before its 9/02 update. All tests described here used the default values of the MPO system variables.

Business Computing Test Results

Typical business workloads, in Data Warehousing (DW) and On-Line Transaction Processing (OLTP), have been evaluated regarding MPO performance improvements. DW and OTLP workloads depend heavily on Database Management System (DBMS) performance, and all parts of the system (processor, memory and I/O channels and devices) are heavily stressed. Therefore, improvements in memory access performance alone will not necessarily generate substantial increases in overall throughput and speed.

Data Warehousing

Tests were performed for a data warehousing workload comprised of a variety of queries executed on a database of about 3 TB in size. The configuration tested was a Sun Fire 15K server, with 72 1.05-GHz UltraSPARC III Cu processors, 288GB of memory (all memory banks full), and 27.2 TB in 19 Sun StorEdge™ FC-AL disk array units. Introduction of the MPO feature in the Solaris 9 Operating System increased overall query performance by approximately 12%.

Enterprise Resource Planning

Tests were performed with a workload that combined a database and an application server for enterprise resource planning, simulating transactions typical of sales and distribution, which can be characterized as OLTP. The configuration tested was a Sun Fire 15K server, with 76 900-MHz UltraSPARC III Cu processors, 288GB of memory (all memory banks full). About 87% of the processors were running application servers, while the remaining ran the database system, in a hostmode fashion. Introduction of the MPO feature in the Solaris 9 Operating System increased overall transaction throughput by approximately 10%.

Java Application

Tests were performed using a workload that depended heavily on the performance of the Java™ virtual machine (JVM™) software. The configuration tested was a Sun Fire 15K server, with 72 1.05-GHz UltraSPARC III Cu processors, 288GB of memory (all memory banks full). The J2SE™

1.4.0_01 platform, in 64-bit mode, was used. Introduction of the MPO feature in the Solaris 9 Operating System did not produce any significant improvement in performance of the workload.

On-Line Transaction Processing

A study was performed to determine the effects of MPO technology on Oracle9i Database software running an OLTP workload.

The OLTP workload was modeled after the transactions of a wholesale parts warehouse supplier. It simulates a complete environment with terminal operators who execute transactions against a database. These transactions include entry and delivery of orders, recording payments, checking the status of orders, and monitoring the level of the stock of each warehouse.

It is important to understand the basic process and memory architecture of Oracle Database software to determine how MPO affects its performance.

Oracle Database Processes

Foreground and background processes are the two basic kinds of processes in the Oracle Database. User processes communicate with the Oracle Database processes to make requests. To process a user request, a number of Oracle Database processes are involved:

- Foreground processes: An Oracle Database foreground or “shadow” process is created to handle requests on behalf of the user. It communicates with the user process and interacts with the Oracle Database to carry out requests for the user.
- Background processes: This set of processes is created for each instance of the Oracle Database to consolidate functions needed to be done for all user processes. Several processes may be used by an Oracle Database instance. However, the following are the most relevant for this workload:

  a. Database Writer: The database writer process writes modified data from the database buffer cache to data files. One or more database writers may be configured for any given instance, depending on whether much data needs to be written;

b. Log Writer: The log writer process writes entries from the redo log buffer located in the System Global Area (SGA) of memory into disk. There is only one log writer process per Oracle Database instance.

Oracle Database Memory Usage

Oracle Database processes use private and shared memory. This is reflected in the basic memory structures in the Oracle Database which are the Program Global Area (PGA) and System Global Area (SGA).

The PGA is part of the memory heap in the Oracle Database server processes and contains data and control information. The PGA is part of the private memory used by Oracle Database processes along with other data from the data, heap, and stack sections of each process. All Oracle Database processes use their private memory.

The Oracle Database foreground and background processes such as the database writers and log writer access the System Global Area (SGA). The SGA is a shared memory region which contains control information and data for one Oracle Database instance. It is allocated when an Oracle Database instance is started and deallocated when the instance is shutdown.

The SGA is shared among users currently connected to the Oracle Database instance. It is used to maximize the amount of data in memory and avoid disk I/O through a caching mechanism, so it is usually made as large as possible for optimal performance. The SGA contains memory structures such as a database buffer cache of most recently used data, a redo log buffer for logging changes to the database, and other data structures shared among the Oracle Database processes.

Results

Tests were performed on a Sun Fire 15K server with 48 900-MHz UltraSPARC III Cu processors, 384 GB of memory, and 36 Sun StorEdge A5200 disk arrays. The results were obtained using the Solaris 9 Operating System, with and without MPO enabled.

The experiment yielded a 9.1% improvement in throughput (transactions per minute) from the baseline to the updated environment using MPO.

In order to see why there is an improvement, one must study how the Oracle Database processes use memory and then evaluate how the MPO feature optimizes memory accesses for locality. Most Oracle Database processes use both private and shared memory for this workload.

It is helpful to look at the distribution of load misses on the processors' E-cache to estimate how often Oracle Database processes access private and shared memory. As seen in Table 3-1, the (normalized) percentage of load misses to E-cache is about 43% for private memory (i.e. Data, Heap, Stack) and 57% for shared memory (i.e. SGA). These numbers were obtained through simulation.

Table 3-1 Distribution of Load Misses to E-cache, evaluated through simulation.

Memory Location	Percentage of Total Memory	Normalized Percentage
Data	9.46%	10%
Heap	25.64%	26%
Stack	6.52%	7%
SGA	56.29%	57%
Total	97.91%	100%

For shared memory, the Solaris 9 Operating System behaves roughly the same way if MPO is present or not. The shadow processes, database writers, and log writer access the SGA across the entire system, and the Solaris 9 Operating System allocates the physical memory for the shared memory pages randomly across the system. This policy effectively levels the cost of accessing the shared memory when it is accessed from processors everywhere on the server, and is essentially what the Solaris Operating System, previous to the 9/02 update, does for all types of memory.

While the MPO feature in the Solaris 9 Operating System does not change the characteristics of shared memory, it distinguishes itself in what it does with private memory allocation. First, the MPO feature introduces locality aware scheduling to ensure that the Oracle Database processes run within its home locality group, giving them the opportunity to benefit from locality. Second, it tries to allocate the physical memory for their private memory pages from their home locality groups.

If the MPO feature manages to make all the private memory of processes be local, one would expect to see a reduction in remote memory accesses equal to the number of accesses to private memory. From the distribution of cache-misses on Table 3-1, the estimated percentage of accesses to private memory was 43%. Thus, MPO should reduce the remote memory accesses by roughly the same amount if it provides good locality to private memory for the Oracle Database processes.

However, this assumes that the Solaris 9 Operating System without MPO provides no locality whatsoever for private memory. This is not true because random memory allocation is used for all memory. Thus, one would expect that it would allocate a fraction of the memory locally, more precisely equal to 1 divided by the number of system boards in the server. On this specific server, that would mean that random placement would result in 1/12 of the memory (~8%) being allocated locally. Consequently, MPO should ideally reduce remote memory accesses by about 39%, which is the portion of private memory which is remote.

In order to verify this, one can use the local and remote E-cache miss counters available on the Sun Fire 15K server through `cpustat(1M)` to measure the percentage of user cache misses to remote memory. The miss rates given in Table 3-2 show that introduction of the MPO feature in the Solaris 9 Operating System clearly provided better locality for Oracle Database processes: E-cache misses to remote memory dropped by 31%, which is close to the ideal percentage of 39% from the analysis above using simulation data.

Table 3-2 Changes in measured E-cache Load Misses and private vs. shared memory distribution

Test	Percentage of User Remote E-Cache Load Misses (Measured)		Percentage of Memory that is Remote (Simulated)	Percentage of Memory that is Local (Simulated)	Total
No MPO	93.1%	Private Memory	39%	4%	43%
		Shared Memory	52%	5%	57%
With MPO	62.2%	Private Memory	0	43%	43%
		Shared Memory	52%	5%	57%
Difference	30.9%	Difference	39%		

One concludes that MPO has done a good job providing locality for Oracle Database processes in this case. The locality aware scheduling and memory allocation of MPO reduced E-cache misses to remote memory significantly for the private memory of Oracle processes. As mentioned earlier, this improvement in locality allowed a 9.1% increase in transaction throughput for this OLTP workload

Other applications may experience similar benefits from the added locality provided by MPO. The UltraSPARC III hardware performance counters may be used to measure the change in E-cache misses to remote memory, and help quantify the improvement in locality that is achieved through MPO.

High Performance Technical Computing Test Results

High-Performance Technical Computing (HPTC) workloads have heavy emphasis in scientific, floating-point-type calculations, and are very memory intensive in general.

Our tests demonstrated that, in HPTC workloads, the effects of MPO are larger, and can vary more, than in the case of business workloads. Some of the following tests demonstrate large gains in performance when MPO is enabled, but such performance differences are highly dependent on the workload. The discussions in the following sections shed some light into the application characteristics that lead to improved performance due to the MPO feature.

Memory-Copy Microbenchmark

A synthetic microbenchmark was constructed, to measure the performance of a vector floating-point operation, in the form:

```
for (i = 0; i < N; i++)
    a(i) = b(i)
```

In today's servers, the time required to perform mathematical operations can be frequently outweighed by the time required to read the operands in from memory. Thus, this benchmark functions as a test of the performance of a machine's memory subsystem.

Sun Fire 15K Server Results

Tests were performed on a 18-board Sun Fire 15K server, with 72 processors and memory distributed evenly across all memory banks. Table 3-3 shows results obtained using the Solaris 9 Operating System, with no MPO implementation, compared to those obtained with MPO active. Each test was run using 4MB pages for the application's heap. Table 3-4 lists the breakdown of the number of 4MB memory pages that was placed in each of the `lgroup`s, as reported by the `meminfo(2)`command - see Appendix A.

Table 3-3 Memory-copy results for a Sun Fire 15K server.

Test	Description	Memory Copy Rate (MB/s)
1	Multi-Threaded, no MPO	14,139
2	Multi-Threaded, with MPO	58,154

Table 3-4 Breakdown of array data locality for Sun Fire 15K server tests.

Test	`lgroup` distribution of memory pages		
1	`lgroup` 00: 1728 pages		
2	`lgroup` 00: 96 pages	`lgroup` 01: 95 pages	`lgroup` 02: 96pages
	`lgroup` 03: 96 pages	`lgroup` 04: 96 pages	`lgroup` 05: 96 pages
	`lgroup` 06: 97 pages	`lgroup` 07: 95 pages	`lgroup` 08: 96 pages
	`lgroup` 09: 96pages	`lgroup` 10: 96 pages	`lgroup` 11: 96 pages
	`lgroup` 12: 98 pages	`lgroup` 13: 94 pages	`lgroup` 14: 96 pages
	`lgroup` 15: 96 pages	`lgroup` 16: 96 pages	`lgroup` 17: 97 pages

In Test 1, we measured the performance of a multithreaded version of the memory-copy microbenchmark. The code was automatically parallelized by the compiler and was configured to run with 72 threads. In the absence of MPO, all the memory was allocated from a single `lgroup`, and that `lgroup` included all the memory in the server. So, even though `meminfo(2)` reported that all memory was in `lgroup 0`, that did not reveal anything about where the memory was physically located in the machine. There is no easy procedure to know exactly how much memory was local to the processors running the

microbenchmark, and how much was actually on a remote board, but because the default allocation for this case (without MPO) is random, we know that only a fraction of memory was local for each processor.

Test 2 runs the same multi-threaded code, with 72 threads, with MPO active. Unlike the previous example, we know that all of the benchmark's data was allocated by "first touch" by default. The `lgroup` breakdown indicates that memory was allocated uniformly across all 18 `lgroups` in which the benchmark was running. Therefore, all memory was actually local to each processor on which the benchmark was running, resulting in performance that was significantly higher than in the case without the MPO feature. The performance gain was 311% for this specialized microbenchmark, but such large gains should not be considered typical for real-world applications.

Table 3-5 presents a side-by-side comparison of the performance of the memory-copy microbenchmark as the number of threads grows. The results for 36 threads or less were obtained on a Sun Fire 12K server.

Table 3-5 Multi-threaded memory-copy performance for Sun Fire 12K and 15K servers, in MB/s. Speed-up is defined as the percentage of improvement in the memory-copy rate.

Threads	No MPO	With MPO	Speed-up
24	11,134	19,421	74%
36	12,332	29,178	137%
72	14,139	58,154	311%

Sun Fire 6800 Server Results

Tests were performed on a 6-board Sun Fire 6800 server, with 24 processors and memory distributed evenly across all memory banks. Table 3-6 shows results obtained using the Solaris 9 Operating System, with no MPO implementation, compared to those obtained with MPO active. Table 3-7 lists the breakdown of the number of memory pages that was placed in each of the `lgroups`.

Table 3-6 Memory-copy results for a Sun Fire 6800 server

Test	Description	Memory Copy Rate (MB/s)
1	Single-Threaded, no MPO	2,134
2	Single-Threaded, with MPO	2,303
3	Multi-Threaded, no MPO	8,795
4	Multi-Threaded, with MPO	9,360

Table 3-7 Breakdown of array data locality for Sun Fire 6800 server tests.

Test	`lgroup` distribution of memory pages		
1	`lgroup` 0: 24 pages		
2	`lgroup` 3: 24 pages		
3	`lgroup` 0: 576 pages		
4	`lgroup` 0: 96 pages	`lgroup` 1: 96 pages	`lgroup` 2: 96 pages
	`lgroup` 3: 96 pages	`lgroup` 4: 96 pages	`lgroup` 5: 96 pages

In Test 1 there is only one processor running the benchmark. Since all of the system's memory was managed as a single pool, the application's memory was distributed fairly evenly across the server boards.

In Test 2, the system had 6 `lgroups` (one per board). All of the microbenchmark's data was allocated on a single board and memory was actually local to the processor on which the benchmark was running. Ensuring that every access goes to local memory did not lead to a significant change in performance. This reflects the fact that the difference between local and remote latency for a Sun Fire 6800 system is low.

Test 3 measures performance of a multi-threaded version of the microbenchmark without MPO, using 24 threads. Memory was distributed fairly evenly throughout the system. While this avoids creating a single hotspot that could limit performance, it also implies that each thread will have to go off-board to access the bulk of its memory.

Test 4 repeats the multi-threaded test with MPO enabled. The `lgroup` breakdown indicates that memory was allocated on each of the 6 `lgroups` in which the benchmark was running, so that all of the memory was local to the each processor running the benchmark. The increased locality reduced average latency for each memory access but, as we saw in the single-threaded case, latency alone does not appear to be a significant limiter of performance for the Sun Fire 6800 server. By reducing the number of off-board accesses, the increased locality probably led to less contention for the bandwidth available on the system's Sun Fireplane Interconnect, thus enabling higher aggregate bandwidths.

Table 3-8 presents a side-by-side comparison of the performance of the memory-copy microbenchmark and the speed-up achieved.

Table 3-8 Multi-threaded memory-copy performance for a Sun Fire 6800 server, in MB/s. Speed-up is defined as the percentage of improvement in the memory-copy rate.

Threads	No MPO	With MPO	Speed-up
1	2,134	2,303	8%
24	8,795	9,360	6%

Finite-Differences Microbenchmark

In this test, MPO influence on a finite-difference application was evaluated. This microbenchmark was chosen because of its ease of execution, high computational throughput, simple setup, and nice spatial locality. The benchmark, written in FORTRAN, consists of iterating *N* times over three sets of calculations performed on thirteen matrices of dimensions *N1*x*N2*.

The model itself consists of three sets of differential equations that are executed by *M* CPUs across thirteen grids in parallel. Because each iteration effectively swamps the external processor cache, and the access pattern is such that data prefetch is relatively ineffective, this benchmark is constrained mostly by memory latency rather than memory bandwidth.

Using the OpenMP parallel programing model, each `DO` loop is split in such a way as to break each *N1*x*N2* matrix into *M* grids which are iterated on by *M* threads. Because of this, each active CPU issues memory accesses only to its own grid while performing computations, eliminating remote-board data transfers when data is local.

During initialization, the fourteen matrices (one is only used for initialization) are split by OpenMP parallel programming into the exact same grids as those during computation. The default heap placement policy of “first touch” places the data close to the CPU making the initial reference. Since the microbenchmark is partitioned in such a way that further references from that thread will also occur to the same grid, remote-board transfers are eliminated, memory latency is minimized, and available memory bandwidth is maximized.

Without MPO, a process’ memory is scattered randomly across the machine all the time. This results in the majority of memory accesses being off-board, encountering extra latency. Since this microbenchmark is perfectly separable, it accesses 100% of its memory local to the CPUs performing computations when MPO is enabled. With this in mind, one would expect the speed-up with MPO to be proportional to the ratio of local to remote memory latency. This was indeed the observation when running with and without MPO on both the Sun Fire 6800 and 15K servers.

Sun Fire 15K Server Results

On a Sun Fire 15K server, the local-to-remote latency ratio is about 1-to-1.8 for the working set size of this benchmark, so a speedup of 30-40% was anticipated when running with MPO enabled.

The test configuration had 72 900-MHz UltraSPARC III Cu processors, and 256GB of memory. The run parameters were as follows, creating a working data set of about 1.4 GB:

```
NO. OF THREADS = 72, MAX NO. OF ITERATIONS = 400, N1=N2=3802
```

The decrease in computational time per iteration was about 41%, as seen in Table 3-9, which compares results obtained running the Solaris 9 Operating System with and without MPO enabled.

Table 3-9 Finite-difference microbenchmark performance (in seconds per iteration) in a Sun Fire 15K. Speed-up is defined as the percentage reduction in the time required per iteration.

No MPO	With MPO	Speed-up
0.317 s / iteration	0.187 s / iteration	41%

Sun Fire 6800 Server Results

On a Sun Fire 6800 system, the local-to-remote latency ratio is about 1-to-1.2, so a speedup of 10-20% was anticipated for the microbenchmark when running with MPO enabled.

The test configuration had 24 900-MHz UltraSPARC III Cu processors, and 192GB of memory. The run parameters were as follows, also creating a working data set of about 1.4 GB:

```
NO. OF THREADS = 24, MAX NO. OF ITERATIONS = 100, N1=N2=3802
```

The decrease in computational time per iteration was about 16%, as seen in Table 3-10, which compares results obtained running the Solaris 9 Operating System with and without MPO enabled.

Table 3-10 Finite-difference microbenchmark performance (in seconds per iteration) in a Sun Fire 6800. Speed-up is defined as the percentage reduction in the time required per iteration.

No MPO	With MPO	Speed-up
0.735 s / iteration	0.615 s / iteration	16%

Additional Tests

A number of HPTC applications were tested to gather data on the potential gains of MPO usage. The data shown in Table 3-11 was obtained on a Sun Fire 6800 server with 24 processors. The speed-up obtained by introducing MPO varied from 0 to 9%, illustrating how MPO performance improvements are highly dependent on the characteristics of the application, like: properties of the computational problem, specific algorithm employed, and implementation details.

Table 3-11 Performance gains due to MPO in several applications for a Sun Fire 6800 server. Performance data is in seconds of execution (lower is better). Speed-up is defined as the percentage reduction in the time required to run the test.

Test	Solaris 9 OE (no MPO)	Solaris 9 12/02 OE (with MPO)	Speed-up
Finite-difference solver	690 s	629 s	9%
Finite-volume solver	174 s	168 s	3%

Table 3-11 Performance gains due to MPO in several applications for a Sun Fire 6800 server. Performance data is in seconds of execution (lower is better). Speed-up is defined as the percentage reduction in the time required to run the test.

Test	Solaris 9 OE (no MPO)	Solaris 9 12/02 OE (with MPO)	Speed-up
Finite-element solver A	258 s	238 s	8%
Finite-element solver B	477 s	448 s	6%
Multiple ODE solver	864 s	858 s	1%
Eigenvalue solver	402 s	403 s	0

Summary 4

The Memory Placement Optimization feature in the Solaris 9 Operating System can improve application performance significantly, and users should keep that in mind when considering their adoption plans for Sun's latest operating system. The MPO feature was designed especially to explore the benefits of the Sun Fire server architecture, and demonstrates how tight integration of operating system and server architecture can create high performance and computational efficiency.

The achievable performance improvements from MPO utilization are heavily dependent on workload characteristics, and on specific server configurations. High-Performance Technical Computing workloads should in general present larger benefits than Business Computing workloads. In the same fashion, larger Sun Fire 12K and 15K server configurations should benefit more from MPO than Sun Fire 3800-6800 servers.

As Sun introduces new technologies into its microprocessor and server lines, technology improvements like MPO should help users reach the highest performance levels on their applications.

Appendix A - Code Examples

Detecting MPO Status

There are certain commands that will help a system administrator determine if the Solaris 9 Operating System running in a server domain has the MPO feature enabled:

Step 1:

The number of `lgroups` should be greater than 1. Verify this through the `nlgrps` variable (equal to 3 in this example):

```
# echo nlgrps/X | mdb -k
nlgrps:
nlgrps:         3
```

Step 2:

The default memory allocation policy should be “next”. Verify this by checking that variable `lgrp_mem_default_policy` is 1:

```
# echo lgrp_mem_default_policy/X | mdb -k
lgrp_mem_default_policy:
lgrp_mem_default_policy:        1
```

Step 3 (only for Sun Fire 12K and 15K servers):

The “lpa” variables should be different than zero. Choose one Uniboard and one CPU inside the domain and verify its variables; the following example shows non-null values for CPU 0 on system board 1:

```
# cfgadm -x passthru -o showlpa SB1::cpu0
showlpa SB1::cpu0 portid 32, base pa 22000000000, bound pa
24000000000
```

Users should always make sure that all required and recommended patches are applied to software installed on the system domains and system controller. Presently, proper default MPO activation requires application of patch number 112488-10 to the System Management Software (SMS 1.2) for the Sun Fire 12K and 15K system controllers, or upgrade to SMS 1.3 (recommended).

Important:

The MPO feature is enabled by default when all proper software and hardware is present in the server (see Section “MPO in the Solaris Operating System” for more details). Use the above commands just to verify the status of the MPO feature. Once enabled, control of the MPO feature should be exercised through the `madvise(3C)`command and `madv.so.1` object.

Controlling MPO

The code sample shown below demonstrates how the `madvise(3C)` and `meminfo(2)` commands can be used to influence how the Solaris Operating System allocates physical memory, and to extract locality information from the Solaris Operating System at runtime. Both APIs have options not shown in this example, but which are described in their `man` pages.

The `dump_lgroups()` routine illustrates how we determined the number of pages allocated on each lgroup when running the benchmarks described in Chapter 3.

Code Sample: Obtaining information and controlling the MPO feature

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

```
/* Defined in <sys/lgrp.h>, but hidden by #ifdef _KERNEL  */
#define NLGRPS_MAX 18

/*
 * Given a buffer and a size, report how many pages in the buffer are
 * backed by memory allocated from each lgroup.
 */
void
dump_lgroups(void *buf, uint64_t bytes)
{
    const unsigned int info = MEMINFO_VLGRP;
    unsigned int    i, v, page_cnt[NLGRPS_MAX];
    uint64_t        addr, last_addr, lgrp;

    for (i = 0; i < NLGRPS_MAX; i++) {
        page_cnt[i] = 0;
    }

    addr = (uint64_t) buf;
    last_addr = addr + bytes - 1;

    while (addr <= last_addr) {
        /*
         * Query Solaris for the lgroup on which this page is
         * allocated.
         */
        if (meminfo(&addr, 1, &info, 1, &lgrp, &v)) {
            perror("meminfo()");
            return;
        }

        if (v) {
            page_cnt[lgrp]++;
        }

        /*
         * We could also use meminfo() with MEMINFO_VPAGESIZE to
         * verify that we aren't using large pages.
         */
        addr += PAGESIZE;
    }

    for (i = 0; i < NLGRPS_MAX; i++) {
        if (page_cnt[i]) {
        printf("    lgroup %d: %d pages", i, page_cnt[i]);
```

```
        }
    }
    printf("\n");
}

int
main(int argc, char **argv)
{
    const unsigned int elements = 1000000;
    double          *first_touch_array, *random_array;
    uint64_t        bytes;
    int             i;

    bytes = elements * sizeof(double);

    /*
     * Allocate the arrays on page boundaries.
     */
    first_touch_array = memalign(PAGESIZE, bytes);
    if (!first_touch_array) {
        fprintf(stderr, "Failed to allocate first-touch array.\n");
        exit(1);
    }
    random_array = memalign(PAGESIZE, bytes);
    if (!random_array) {
        fprintf(stderr, "Failed to allocate random array.\n");
        exit(1);
    }

    /*
     * Advise Solaris on how physical memory should be allocated for
     * each array.
     */
    if (madvise((caddr_t)first_touch_array, bytes,
        MADV_ACCESS_LWP)) {
        perror("madvise(MADV_ACCESS_LWP)");
        exit(1);
    }

    if (madvise((caddr_t)random_array, bytes, MADV_ACCESS_MANY)) {
        perror("madvise(MADV_ACCESS_MANY)");
        exit(1);
    }

    /*
     * Touch the first byte on each page of each array.  This forces
```

```
         * Solaris to allocate physical memory to back them.
         */
        for (i = 0; i < elements; i += (PAGESIZE / sizeof(double))) {
            first_touch_array[i] = 0;
            random_array[i] = 0;
        }

        printf("lgroup breakdown for the first-touch array:\n");
        dump_lgroups((void *) first_touch_array, bytes);

        printf("\nlgroup breakdown for the random array:\n");
        dump_lgroups((void *) random_array, bytes);
        exit(0);
}
```

Appendix B - Sun Fire Servers

The heart of a shared-memory processor (SMP) system is the interconnect, and Sun has a long history of designing efficient interconnects that were used to create servers with outstanding scalability (see Ref [1]).

This appendix discusses the basics for designing Sun Fire servers based on the Sun Fireplane interconnect. The current family of UltraSPARC III-based servers are described in detail, and their memory access characteristics are highlighted. References [1] to [6] contain more detail on the subject.

Introduction to Cache Coherency

A shared-memory system interconnect has to send memory addresses, maintain cache coherency, and transfer cache-line sized-blocks of data. Each processor has a *cache* in which to keep frequently accessed data blocks, so that they are quickly available. A typical cache block size is 32, 64, or 128 bytes. These blocks are also called cache *lines*.

When a processor cannot find a needed data item in its cache (called a *cache miss*), it asks the interconnect to supply that block. Requests are satisfied from memory unless some system device (a processor or an I/O controller) currently has a modified copy of that block in its cache.

To get permission to modify a block, a processor has to become its *owner*. All other devices then invalidate any copies they have, and the old owner supplies the data. Henceforth, when other processors request to share a read-only copy

of the data, the owning device, not memory, will supply it. Memory becomes the owner again when the owning processor needs to make room in its cache for new data, and it *victimizes* the cache block by writing it back to memory.

This process of finding the up-to-date copy of a cache block is called *cache coherency*. System designers use two main methods to keep each processor's view of memory consistent.

1. In *broadcast (snoopy) coherency*, all addresses are sent to all system devices. Each device examines (snoops) the state of the requested cache line in its local caches. The system determines the overall snoop result a few cycles later. Broadcast coherency provides the lowest possible latency. Sun has constantly increased the rate of broadcast coherency, since this is the method used in Sun's mid-sized servers, such as the Sun Fire 6800 server.

2. In *directory (point-to-point) coherency*, addresses are sent only to those system devices that are known to be interested in that cache block. The hardware keeps a directory in memory or in special RAM to keep track of which system devices share or own each cache block. Since all addresses are not sent everywhere, the total system bandwidth can be much higher than with broadcast coherency. However, the latency is longer and more variable, due to the more complicated protocol.

For more on cache coherency, see chapters six and eight of Ref. [7].

Sun SMP Server Generations

Approximately every four years, Sun has introduced an improved system-interconnect architecture to support new SPARC® processor generations. The Sun Fireplane interconnect described here is Sun's fourth generation of shared-memory interconnect.

So far, these SMP interconnects have been used across Sun's entire system-size spectrum, from two processors on up. Table B-1 and Figure B-1 summarize Sun's four SMP generations.

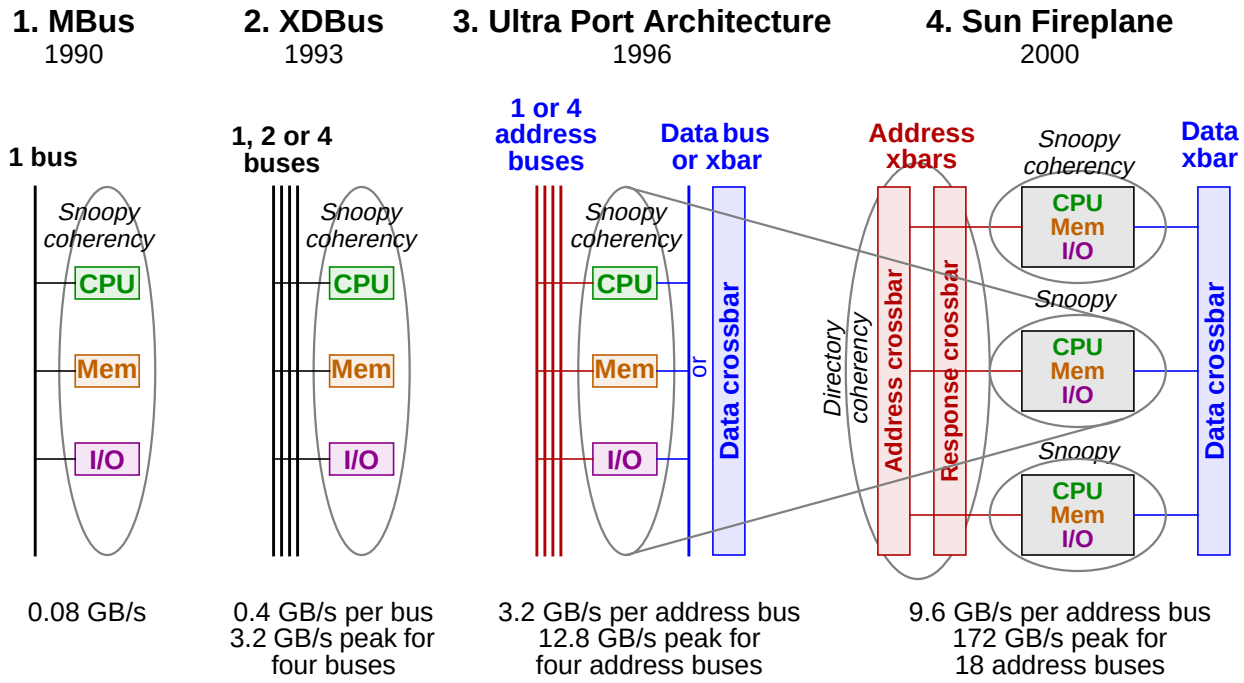


Figure B-1 Sun interconnect generations

Sun Fireplane Interconnect

For its fourth generation, Sun has put the snooping logic and cache tags inside the system devices, so they can snoop a new address every system clock. The system clock has been increased to 150 MHz. Compared to the previous generation, these improvements tripled the data bandwidth of a single-address bus, mid-sized system to 9.6 GB/s (see Figure B-1).

All systems have been implemented with point-to-point electrical signals. All systems use a data crossbar rather than a data bus. Dynamic System Domains, introduced on the Sun Enterprise 10000 server in the previous generation, have been extended down to mid-range servers. There are more common components between mid and large sized systems.

Table B-1 Sun system interconnect generations.

	1. MBus [8]	2. XDBus [8]	3. Ultra Port Architecture (UPA) [9]	4. Sun Fireplane [1]
First system shipments	1990	1993	1996	2000
Processor	CypressSPARC®	SuperSPARC®	UltraSPARC I/II	UltraSPARC III
Processor clock	40 MHz	40–60 MHz	167–400 MHz	750–1200 MHz
Interconnect clock	40 MHz	40–55 MHz	83–100 MHz	150 MHz
System sizes	1–4 processors	1–64 processors	1–64 processors	1–106 processors
Cache-coherency mechanism	Broadcast			Broadcast + point-to-point
Packet protocol	Circuit switched	Packet switched		
Address and data	Multiplexed on same wires		Separate wires	
Cache coherency line size	32 bytes	64 bytes		
System clocks per address	16	11	2	1
Address rate per address bus	2.5 million/s	4.5 million/s	50 million/s	150 million/s
Data bandwidth per address bus	0.08 GB/s	0.29 GB/s	3.2 GB/s	9.6 GB/s
Maximum number of address buses	1	4	4	18
Maximum address-limited data bandwidth	0.08 GB/s	1.28 GB/s	12.8 GB/s	172 GB/s
Datapath width	8 bytes		16 bytes	32 bytes
Electrical implementation	Bused	Bused	Bused & switched	Switched
Note: 1 GB/s (gigabyte per second) = 10^9 bytes per second				

The protocol is improved so that address and data packets need only to traverse the shortest path between source and destination. In the previous generation, packets always had to go all the way to the outermost level of the interconnect, even if they were going to a destination on the same board. The current implementation lowers the latency for “close” transfers.

The coherency protocol was extended to do directory coherency between multiple snooping coherency domains — allowing the interconnect bandwidth to increase beyond the limit imposed by broadcast coherency.

The global address interconnect of the largest architecture, the Sun Fire 15K server, was implemented using an address crossbar and a response crossbar to connect 18 snooping coherency domains. This increased the peak interconnect bandwidth over the previous generation by 13-fold to 172 GB/s, and increased the centerplane bisection bandwidth by 3.3x to 43 GB/s. The maximum processor count was increased to 106 processors.

The Sun Fireplane interconnect protocol is used in all current UltraSPARC III-based systems, which range from 2 to 106 processors.

Number of Interconnect Levels

The Sun Fireplane interconnect is implemented with up to four levels of interconnect logic. The number of interconnect levels required is approximated by $\log_4 n$, where n is the total number of system devices (processors, memory units, and I/O controllers). The total amount of interconnect logic needed for n components is approximately $n\log_4 n$. The Sun Fireplane-based servers can be divided into four categories by the number of interconnect levels required:

1. **Small server**. Four system devices (two processors, one memory unit, and one I/O controller) require only one level of data switch interconnect.
2. **Workgroup server**. 18 system devices (eight processors, eight memory units, and two I/O controllers) require two levels of interconnect.
3. **Mid-size server**. 56 system devices (24 processors, 24 memory units, and eight I/O controllers) require three levels of interconnect.
4. **Large server**. 180 system devices (72 processors, 72 memory units, and 36 I/O controllers, or 106processors, 72 memory units and 2 I/O controllers) require four levels of interconnect.

The increased number of levels needed for larger systems explains why they have longer average latencies than smaller systems.

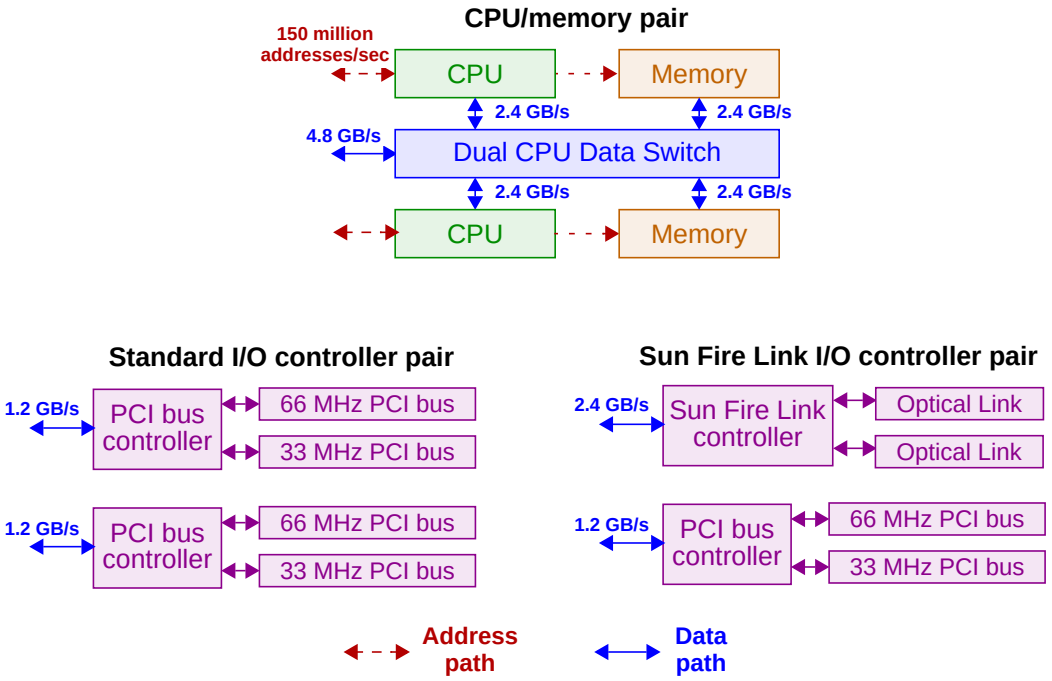


Figure B-2 Sun Fireplane active system devices

Address Interconnect

Levels one and two of the Sun Fireplane address interconnect perform snoopy coherency using a tree-structure of Address Repeater ASICs (Application Specific Integrated Circuits). A new address can be broadcast at a peak rate of one every interconnect clock cycle (1/150MHz = 6.67 ns). All system devices get a particular address at the same time — six clocks after it was originated.

The memory access is started as soon as the address is received by the CPU that controls the requested location. About half the memory access time is overlapped with the nine clocks required to compute the snoop result and broadcast it to the devices. If a system device owns the requested block, then it supplies the data to the requester, and the data from memory is not sent.

Data Interconnect

The Sun Fireplane data interconnect is implemented from a hierarchy of crossbar switches. Board-level connections are 32 bytes wide to CPU boards, and 16-bytes wide to I/O boards. All datapaths are bidirectional, and can move data in one direction or the other every system clock.

Active System Devices

Active system devices originate transactions. Currently there are three such system devices: the UltraSPARC III processor, the PCI I/O controller, and the Sun Fire Link I/O controller. These are the building blocks of all Sun Fireplane-based systems. Both CPUs and I/O controllers are interconnected in pairs, as shown in Figure B-2.

Processor

The UltraSPARC III processor used in the mid-size and large servers currently has clock rates of 900 MHz, and 1.05 GHz. See Figure B-3 for its block diagram.

Instruction Execution

The instruction issue unit can initiate up to four in-order instructions per clock from a 32 KB, four-way associative instruction cache. Instructions may complete out of order. The integer unit can perform up to four operations per clock: two ALU operations, one load/store, and one branch. The floating-point unit can initiate a multiply-type operation and an add-type operation every clock.

On-chip Caches

The data cache unit includes a 64 KB, four-way set associative, data cache, plus a 2 KB write cache for merging stores, and a 2 KB prefetch cache for buffering data from software prefetch instructions and automatic hardware prefetching from the external cache.

External Cache

An 8 MB, two-way associative second level cache is provided using high-speed external SRAMs. The tags for the second-level cache are located inside the processor chip to allow high-speed snooping.

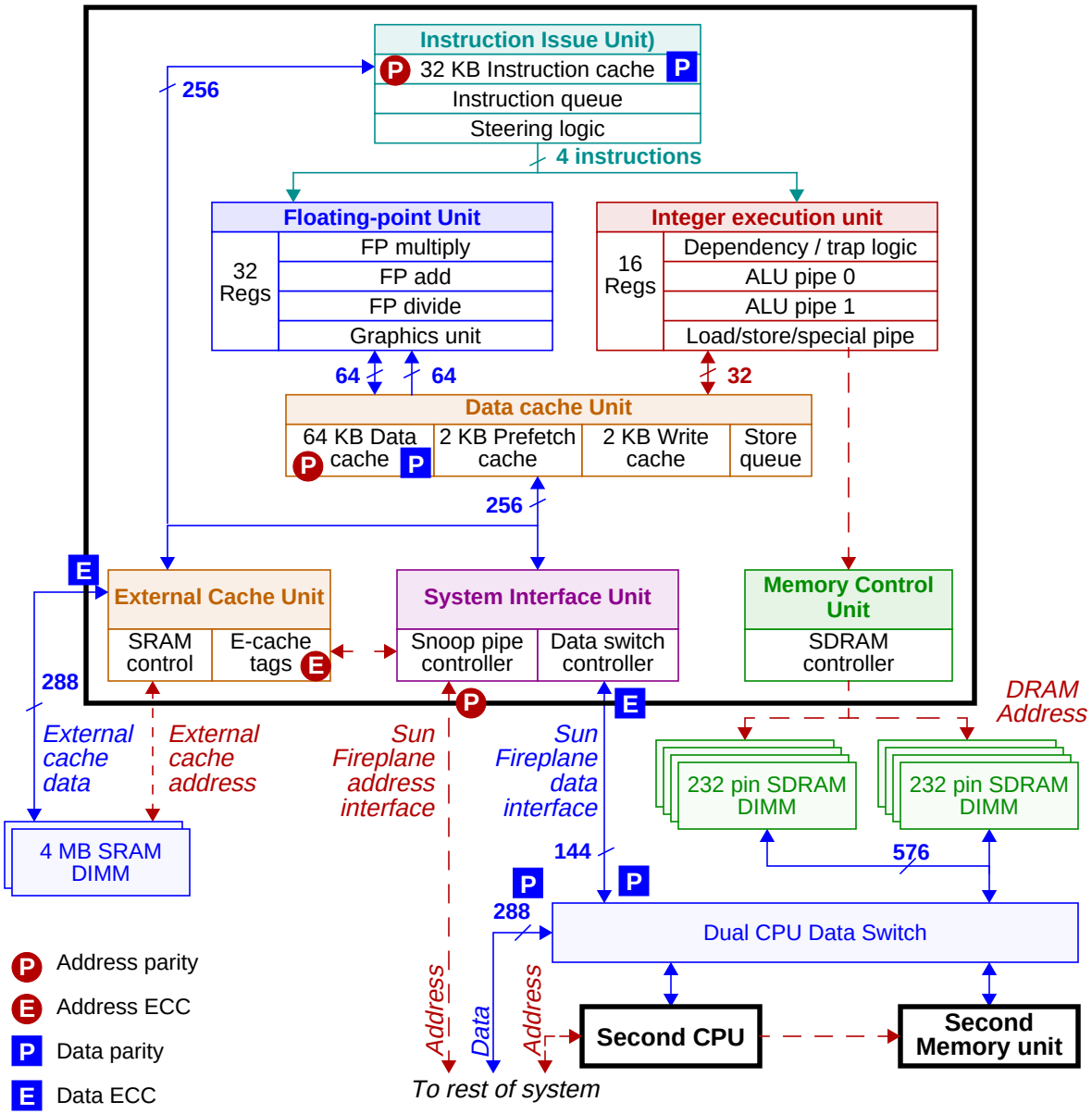


Figure B-3 UltraSPARC III processor subsystem

Memory Controller

To reduce chip count and latency, and to provide scalable memory bandwidth, each processor has an on-chip DRAM controller. This provides addresses and control signals for two groups of four dual-banked SDRAM DIMMs. Currently, the maximum DIMM size is 1 GB, so each CPU can configure up to 8 GB of memory.

System Interface

The processor has a 150 MHz Sun Fireplane address interface. It can snoop a new address every system interconnect clock cycle. The processor has a 2.4 GB/s data interface, which goes to the Dual CPU Data Switch. This switch is implemented from 8 bit-sliced ASICS, and connects together two processors and two memory units, and has a port to higher levels of the data interconnect. It is shown in the lower part of Figure B-3.

For more information on the UltraSPARC III processor, see Ref. [10].

PCI Controller

The PCI Controller implements two 64-bit-wide PCI buses, one running at 33 MHz and one at 66 MHz. The 33 MHz PCI bus has from one to three PCI slots, depending upon the particular server. It has a peak bandwidth of about 0.25 GB/s. The 66 MHz PCI bus has one PCI slot, and has a peak bandwidth of about 0.5 GB/s. The PCI Controller has an 8-byte wide (1.2 GB/s peak) connection to the Sun Fireplane data interconnect. On all but the smallest systems, the PCI Controllers are configured in pairs (see bottom of Figure B-2).

Sun Fire Link Controller

The Sun Fire Link Controller implements a proprietary cluster interconnect that enables fast memory-to-memory transfers [11]. Each controller provides two bi-directional optical links, with 1.2 GB/s peak bandwidth each. The Sun Fire Link Controller has an 2.4 GB/s peak connection to the Sun Fireplane data interconnect. The Sun Fire Link controller is always paired with a PCI Controller (see bottom of Figure B-2).

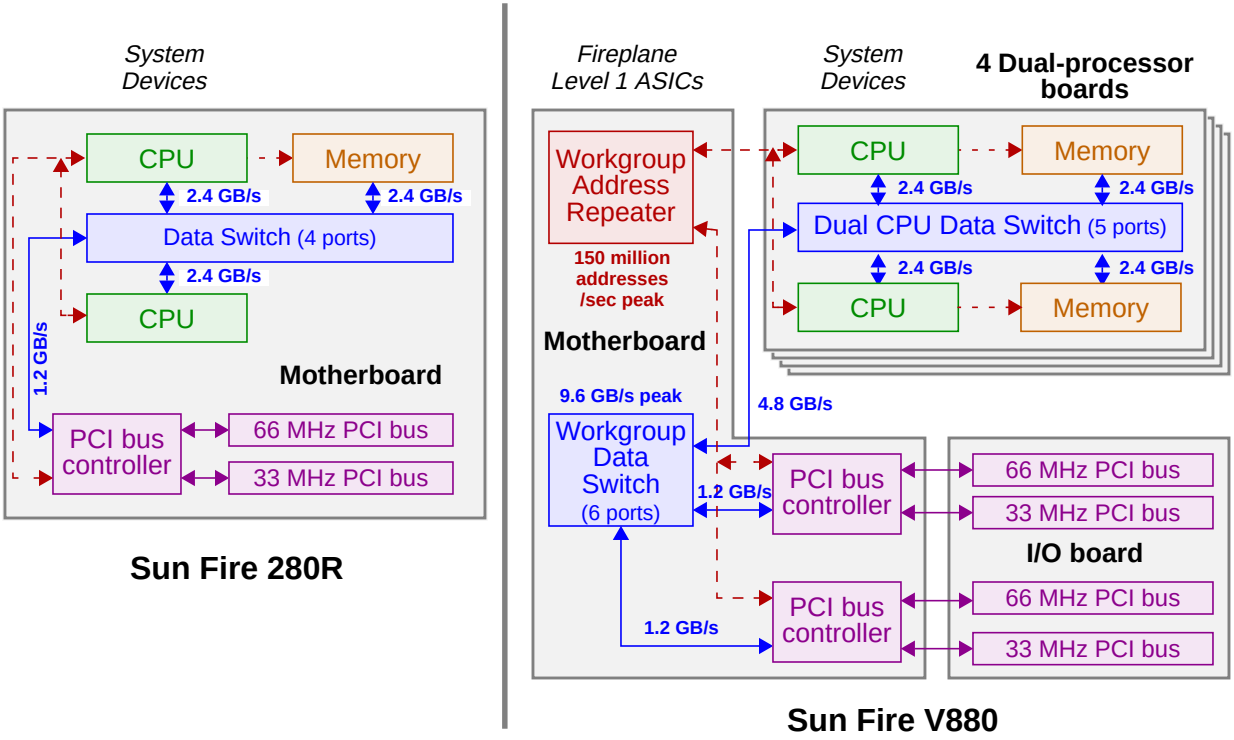


Figure B-4 Small and workgroup interconnects (Sun Fire 280R and V880 servers)

Small Sun Fireplane Interconnect

A small server like the Sun Fire 280R needs only a minimal amount of interconnect for four devices (two processors, one memory unit, and one I/O controller) — see Figure B-4. The two processors and one I/O controller connect directly together via a Sun Fireplane address bus. A four-port data switch (implemented from six bit-sliced ASICs) connects together the four system devices.

Workgroup Sun Fireplane Interconnect

A workgroup server like the Sun Fire V880 needs one level of Sun Fireplane interconnect for up to 18 devices (eight processors, eight memory units, and two I/O controllers) — see Figure B-4.

Dual Processor Board

The Dual-processor board used on workgroup servers holds a pair of processors and a pair of memory units. The two processors share a Sun Fireplane address bus connection to the motherboard. The four devices connect to the motherboard via a Dual CPU Data Switch, which is implemented from 8 bit-sliced ASICs. These same ASICs are used on all servers with more than two processors.

Sun Fireplane Level 1: Motherboard

Level 1 is located on the system motherboard. The workgroup Address Repeater ASIC has five address bus ports, one for each pair of system devices. A peak of one address can be broadcast every system clock (150 MHz) on the address interconnect. This address rate determines the peak data bandwidth of 9.6 GB/s, since an address is required to initiate each 64-byte data transfer.

The workgroup Data Switch is implemented with six bit-sliced ASICs that provides a 6x6 crossbar between four 32-byte wide ports for the Dual-processor boards, and two 8-byte wide ports for the I/O controllers.

Mid-Range Sun Fireplane Interconnect

Mid-range servers like the Sun Fire 3800-6800 need two levels of Sun Fireplane interconnect for up to 56 devices (24 processors, 24 memory units, and eight I/O controllers) — see Figure B-5.

Sun Fireplane Level 1: System Boards

Level 1 is implemented on two types of boards, one type for processors and memory, and the other type for I/O.

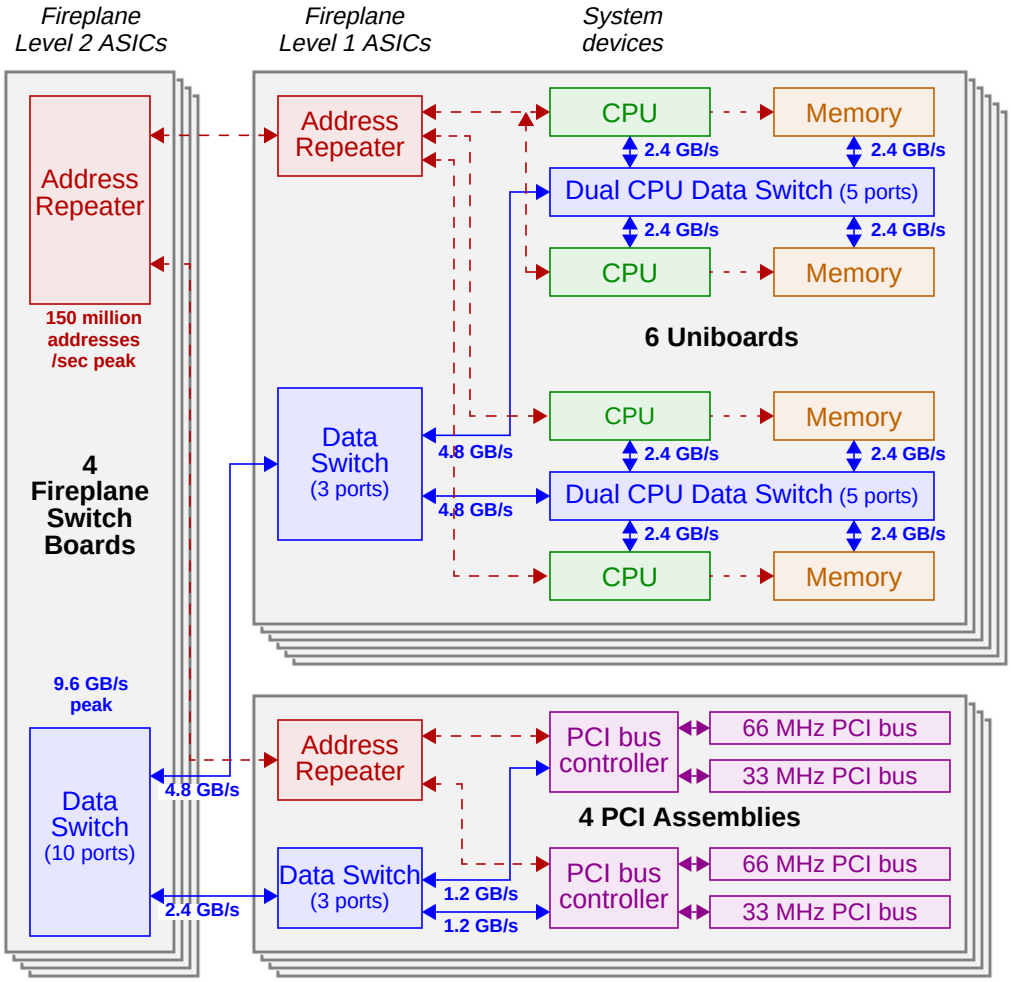


Figure B-5 Mid-range interconnect (Sun Fire 3800-6800 servers)

Uniboard

A Uniboard holds two pairs of processors and two pairs of memory units. The Uniboard block diagram is shown in Figure B-5, and Uniboard physical layout is shown in Figure B-7. The Uniboard is used in all medium and large Sun Fire servers.

The Address Repeater ASIC collects address requests from the four processors, and forwards them up to the level-2 Address Repeater. It also broadcasts addresses that come down from level-2.

Eight Dual CPU Data Switch ASICs connect together a pair of processors and a pair of memory units. Four Data Switch ASICs connect the two halves of the board to the level 2 data interconnect via a 32-byte wide, 4.8 GB/s path.

I/O Assemblies

An I/O assembly holds a pair of I/O controllers. The Address Repeater ASIC collects address requests from the two I/O controllers, and forwards them up to the level-2 Address Repeater. It also broadcasts addresses that come down from level-2. Two Data Switch ASICs connect the two I/O controllers to the level-2 data switch via a 16-byte wide, 2.4 GB/s path.

So far, there are four types of I/O assemblies for the mid range. All standard PCI assemblies have the same architecture, and provide two 33 MHz PCI buses, and two 66 MHZ PCI buses. They have different numbers and types of PCI slots:

- Six compact PCI slots (Sun Fire 3800 server).
- Four compact PCI slots (Sun Fire 4800-6800 servers).
- Eight standard PCI cards (Sun Fire 4800-6800 servers).

The Sun Fire Link I/O assembly for the mid-range provides two bidirectional optical links and two cPCI slots.

Sun Fireplane Level 2: Switch Boards

Level 2 connects multiple Uniboards and I/O assemblies. Level 2 is the top of the address-broadcast tree, and so encompasses a snooping coherency domain. The maximum data bandwidth inside a snooping coherency domain is determined by the 150 MHz address rate × 64-byte cache block = 9.6 GB/s.

Sun Fire 3800 Server

Level 2 is implemented on the motherboard. The Sun Fire 3800 server can hold two Uniboards (8 processors) and two six-card Compact PCI assemblies.

Sun Fire 4800-6800 Servers

Level-2 is implemented on hot-swappable Sun Fireplane switch boards. Two Sun Fireplane switch boards are used in the Sun Fire 4800-4810 servers to interconnect three Uniboards and two I/O assemblies, and four are used in the Sun Fire 6800 server to interconnect six Uniboards and four I/O assemblies. A Sun Fireplane switch board contains one Address Repeater ASIC and two Data Switch ASICs.

These systems use the four-slot Compact PCI assembly, the eight-slot regular-PCI assembly, and in the case of the Sun Fire 6800 server, the Sun Fire Link I/O assembly.

Segments

The mid-range level-2 interconnect can optionally be split into two *segments*, where half the Sun Fireplane switch boards are in one segment, and half are in the other. This creates two independent systems in one box which do not share any logic components. Configuring a mid-range system into two segments doubles the address-limited bandwidth of the box to 19.2 GB/s, but halves the Sun Fireplane switch bandwidth, since it must then be run in double-pumped mode.

If most traffic is local to each board, then the reduction in data switch bandwidth may have little effect, and the aggregate bandwidth of a segmented system can approach twice that of an unsegmented system. Memory Placement Optimization can particularly benefit Sun Fire mid-range servers in this segmented mode, since by keeping memory accesses local to each board the effects of diminished interconnect bandwidth are minimized.

When the segments are configured along the half-cabinet power-grid boundaries of a Sun Fire 6800 server, then there is no electrical connection between the two halves, and they function as a totally isolated, two-way cluster-in-a-box.

Domains

A segment is configured into one or two Dynamic System Domains. Each domain runs a separate instance of the Solaris Operating System, and has its own boot disk. A domain must have at least one Uniboard and one I/O assembly, so there can be up to two domains in a Sun Fire 3800-4810 server, and four in a Sun Fire 6800 server. A domain is isolated from other domain's Uniboard and I/O assembly hardware faults, as well as any software faults.

When there are two domains in a segment, they share the Sun Fireplane switch boards of that segment, and split between them the address bandwidth. A system board failure in one domain will not affect the other domain, but a switch board failure will bring down both domains in a segment.

High-End Sun Fireplane Interconnect

A large server like the Sun Fire 15K needs three levels of interconnect for up to 180 devices (72 processors, 72 memory units, and 36 I/O controllers, or 106 processors, 72 memory units, and 2 I/O controllers) — see Figure B-6.

Sun Fireplane Level 1: System Boards

Uniboard

This is the same Uniboard as on all mid and high-end servers.

I/O Assembly

The hot-swap PCI I/O assembly has the same architecture as for the mid-range systems, but is different physically to fit into the *slot-1* form factor of the expander board. It accommodates up to four standard PCI cards, which are mounted in hot-swappable cassettes. A Sun Fire Link I/O assembly is also available, providing two bidirectional optical links, and accommodating up to two standard PCI cards, mounted in the same hot-swappable cassettes as in the standard PCI I/O assembly.

MaxCPU board

When not all of the 18 type-1 slots are needed for I/O connectivity or I/O bandwidth, then the remainder of the type-1 slots can be populated with MaxCPU boards.

A MaxCPU board has two processors, but no on-board memory. These processors use the memory on other boards. The off-board bandwidth per processor is 1.2 GB/s, the same as for the Uniboard.

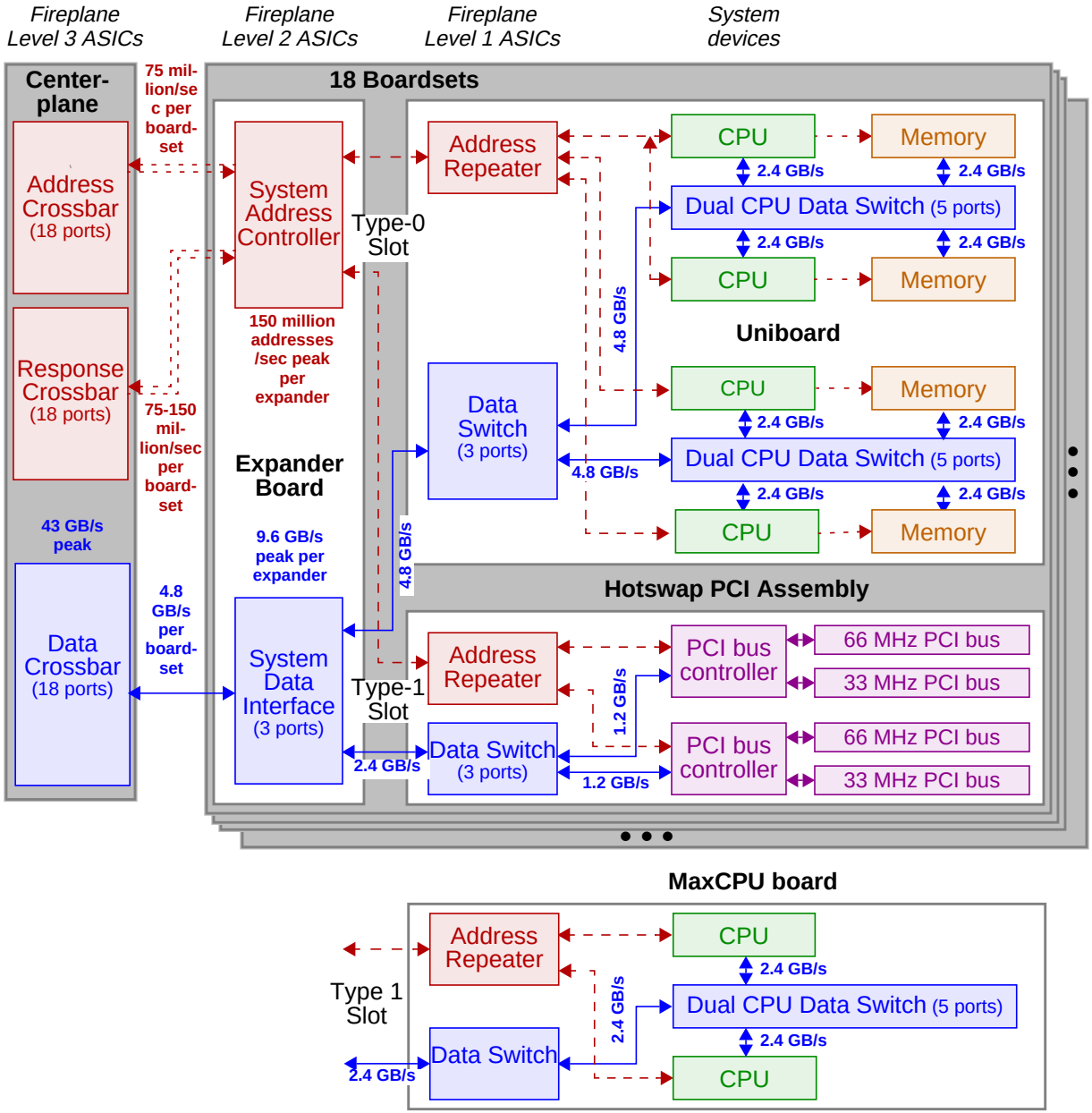


Figure B-6 High-end interconnect (Sun Fire 12K / 15K servers)

Sun Fireplane Level 2: Boardset

Level 2 is implemented on an *Expander board*. The Expander board has one type-0 slot for a Uniboard, and one type-1 slot for an I/O assembly or MaxCPU board. The three-board combination of an Expander board, a Uniboard, and slot-1 board is called a *boardset*. Each of these boards is hot-swappable

For transfers within a boardset, the System Address Controller acts as a level-2 Address Repeater, and the six System Data Interface ASICs act as a level-2 data switch. A boardset by itself with one Uniboard and one I/O assembly behaves identically to a similarly configured Sun Fire 3800-6800 server.

Sun Fireplane Level 3: Centerplane

Level 3 connects together the 18 boardsets. Each boardset has separate input and output connections to the address crossbar and to the response crossbar. Address requests are sent across the address crossbar, and take two clocks. Replies are sent back on the response crossbar, and take one or two clocks. Each boardset has a 4.8 GB/s peak data connection to the 18x18 centerplane data crossbar.

The centerplane address crossbar is implemented by four ASICs, the response crossbar by two ASICs, and the data crossbar by 14 ASICs. In the event of an ASIC failure, each of these crossbars can be individually reconfigured into double-pumped mode which operate at half the usual rate.

Directory Coherency

Each boardset is a separate snooping coherency domain. When a processor requests a memory address in a different snooping coherency domain, the *scalable-shared memory* (SSM) agent in the system address controller ASIC on the requester's expander board notices the request, and sends it across the address crossbar to the *home* SSM agent in the system address controller ASIC on the home boardset. Home is the boardset where the memory is physically located. The home agent keeps track of which snooping coherency domain any sharers are in, and where the current owner is.

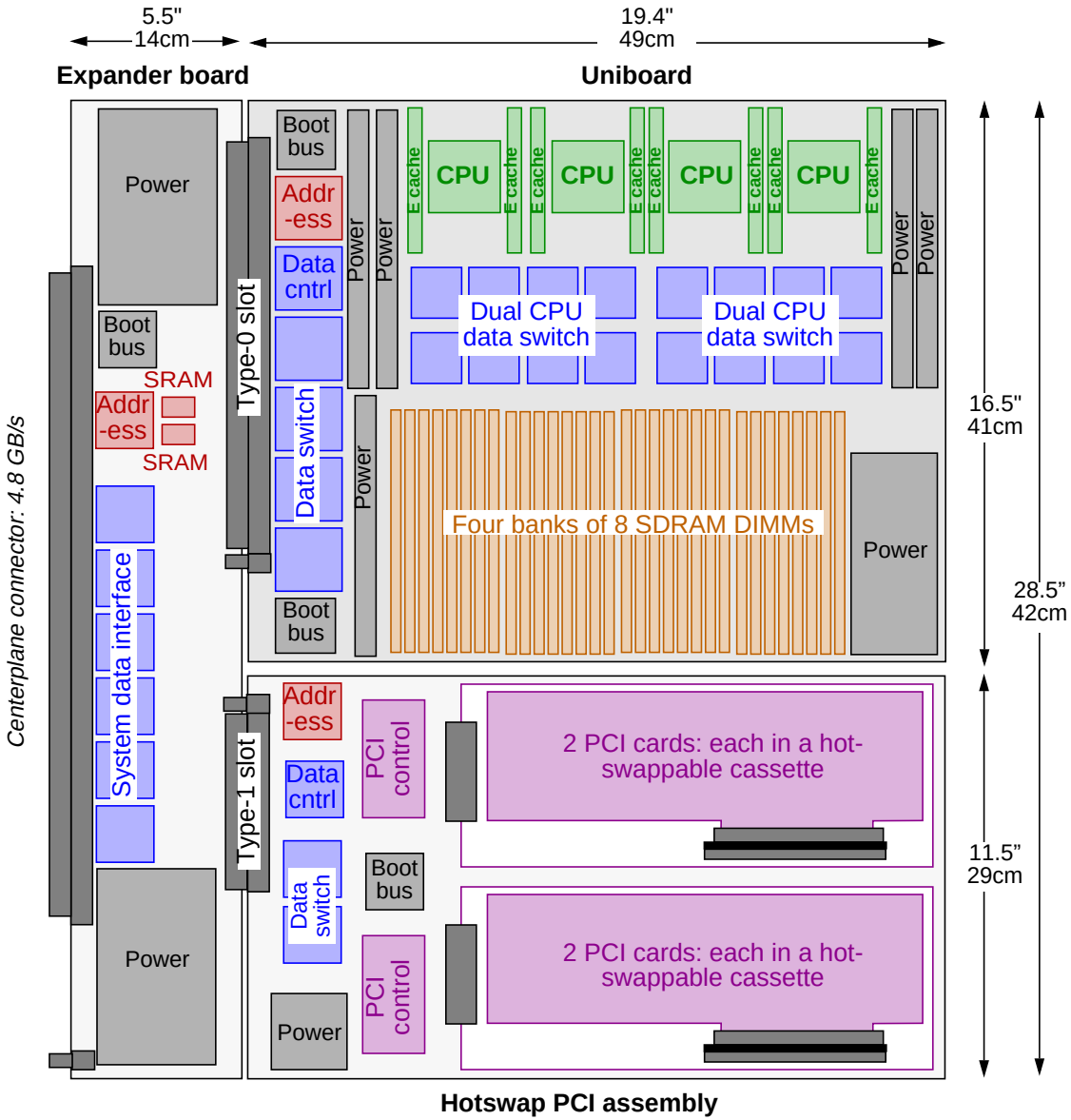


Figure B-7 Sun Fire 12K/15K server boardset

The home agent broadcasts the request on the home address bus, and if the block is not owned by a cache on the home boardset, then memory supplies the data. The data is moved through the home boardset data switch, the centerplane data crossbar, and the requesting boardset's data switch to reach the requester.

If the requested location is currently owned by a system device, then the home agent sends the address over the address crossbar to the SSM agent on the owning boardset. The owning agent broadcasts the address on its address bus, and the owning device supplies the data.

In all cases, the requesting agent reruns the address transaction on the requester's address bus, to establish the request's place in the global memory order.

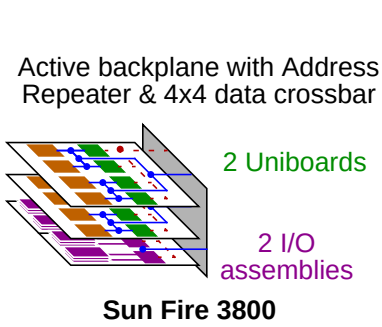
The SSM protocol uses three tag bits stored in memory (the *Mtags)* to specify the global coherency state of a memory block. The Mtag state is in addition to the snoopy Modified, Owned, Exclusive, Shared, Invalid (MOESI) state of a line.

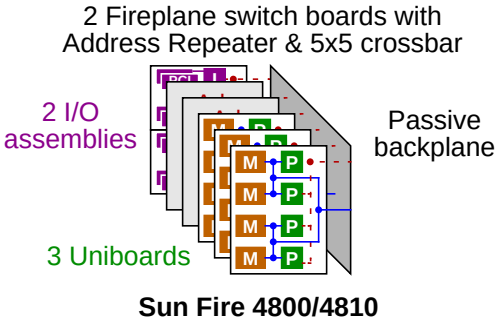
The Mtags are stored in the eight-bytes of error-correcting code (ECC) information that goes with each 64-byte memory block. Memory is used because the number of snoop tags that would have been required to represent all the data cached in other snooping coherency domains would have been too large to fit in high speed SRAM. Instead, each SSM agent has an SRAM Coherency Directory Cache (CDC) to give it quick access to the most recent coherency information.

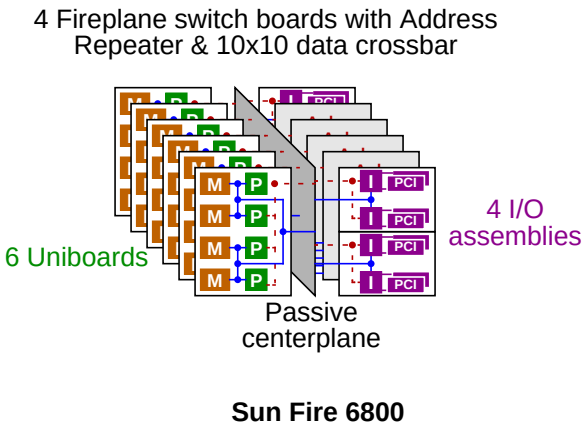
For more detail on the Sun Fireplane SSM coherency protocol, see [1] and [4].

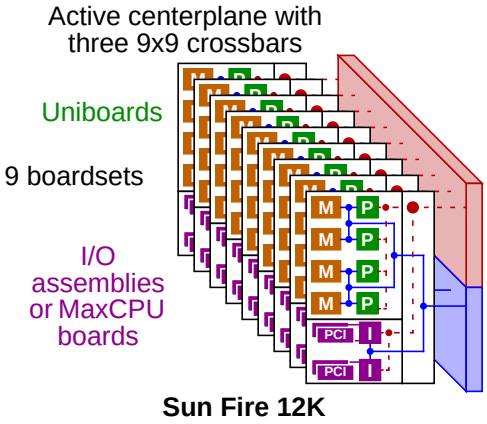
Peak Bandwidth

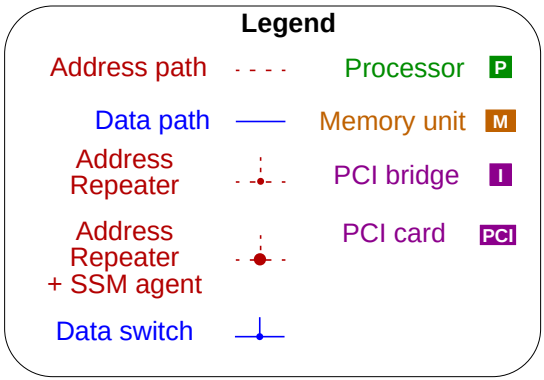
When all accesses in a Sun Fire 15K server are local to each boardset, the peak interconnect bandwidth is the aggregate local interconnect bandwidth of each boardset: 18 boardsets × 9.6 GB/s = 172 GB/s. The peak local memory bandwidth is about 6.7 GB/s per boardset (assuming that memory banks are 16-way interleaved), which gives a peak memory bandwidth of about 120.6 GB/s. When all accesses are to a different boardset than the requester, the peak interconnect bandwidth is the bisection bandwidth of the centerplane: 18 boardsets × 2.4 GB/s = 43 GB/s.











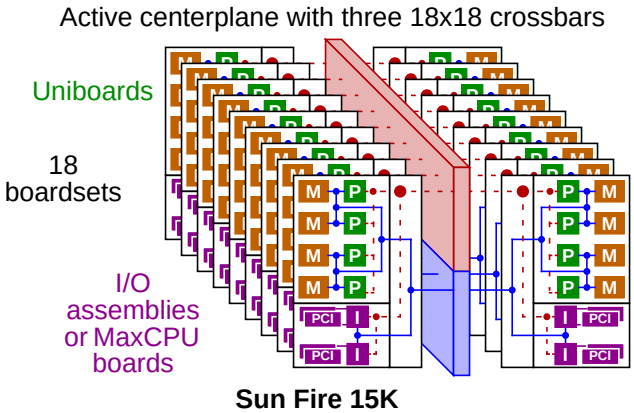


Figure B-8 Sun Fire family interconnect overview

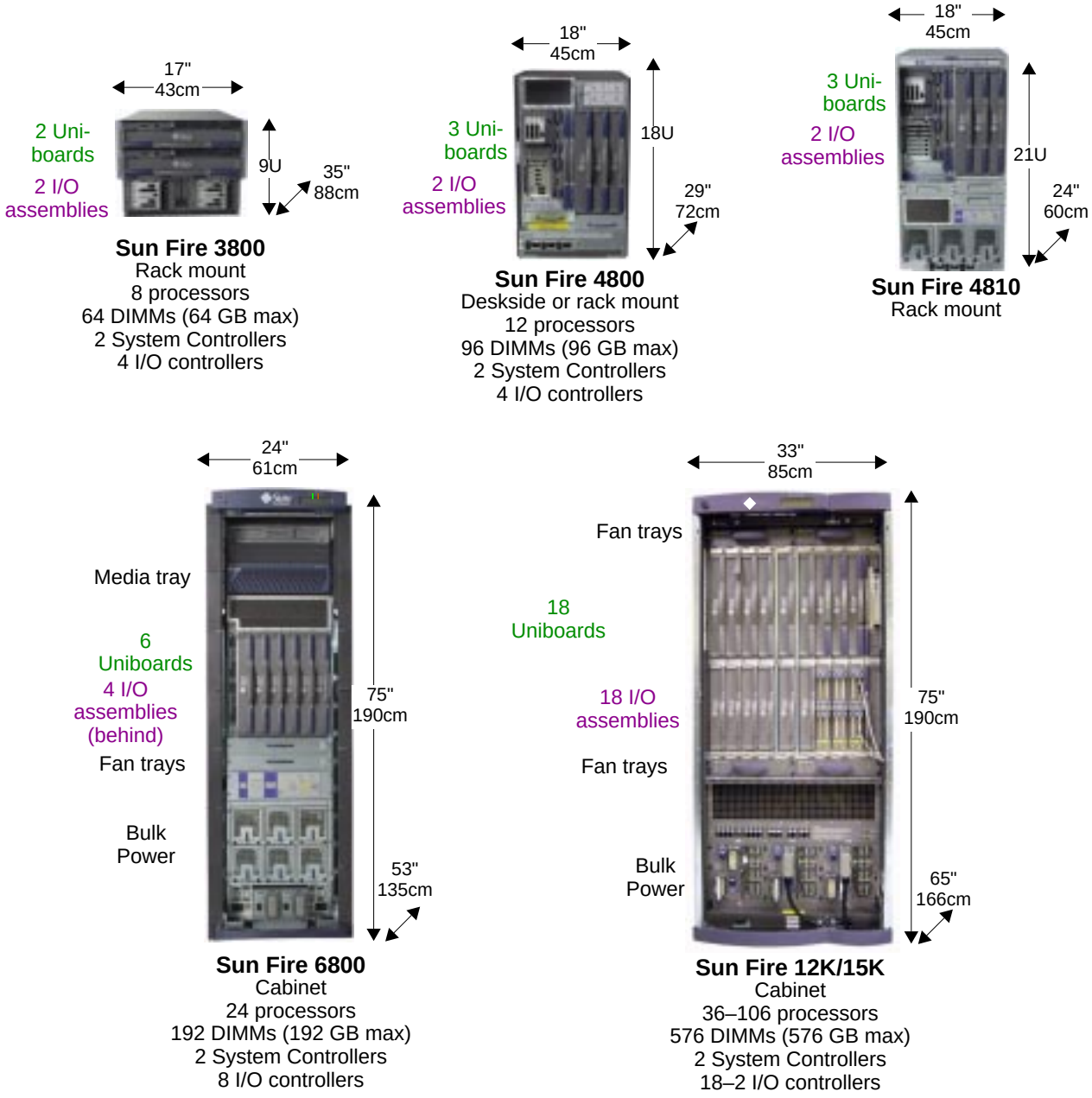


Figure B-9 Mid-range and high-end Sun Fire family cabinets

Domains

The Sun Fire 15K server can be divided into as many as 18 Dynamic System Domains, on a board-level granularity. A Uniboard from a given boardset can be in one domain, and the I/O assembly can be in another, which makes it possible to move CPUs from one domain to another without affecting the I/O connections.

Sun Fire Server Cabinets

Figure B-8 shows an overall picture of the Sun Fireplane interconnect implementation in the various servers, and Figure B-9 shows the family cabinets.

Memory Access Performance

The variations on the Sun Fireplane interconnect design in each of the server models result in differences on memory performance characteristics among Sun Fire servers. In particular, the more powerful and scalable servers will present different memory access speeds depending on the location of the memory page (or cache line) and of the requesting CPU. These performance differences manifest themselves in terms of latency and bandwidth. For the purposes of this discussion, we will focus on requests made by CPU processes for data that is resident in memory, and will exclude requests for cache lines that are actually in CPU caches.

On a small of workgroup server, like the Sun Fire 280R or Sun Fire V880 servers, such differences in memory access speeds due to locality are negligible for all practical purposes. This is due to the fact that, in those server models, the Sun Fireplane interconnect implementation is very compact, using a level-0 or level-1 approach.

On the Sun Fire 3800-6800 server family, the presence of a level-2 logic on the Sun Fireplane interconnect creates small locality effects. Therefore, memory access speeds are faster when the requesting CPU is in the same Uniboard as the memory that contains the requested data. Additional cycles are necessary when the requesting CPU and requested data are in different Uniboards.

On the Sun Fire 12K and 15K servers, in addition to the level-3 logic introduced, the presence of the SSM protocol creates variations in memory access speed depending on the situation (a hit on the directory cache will enable faster memory access) — see Ref. [4].

Table B-2 lists latencies for Sun Fire mid-range and large servers. Pin-to-pin latency is calculated by counting clocks in the interconnect logic design between the address request from a CPU and the completion of the data transfer back into the CPU. Memory latencies for small or workgroup servers are of the order of the lowest (same board) latencies in a Sun Fire 6800 or 15K system.

Table B-2 Pin-to-Pin latencies for data in memory for Sun Fire mid-range and high-end servers

Location of Memory	Sun Fire 3800-6800	Sun Fire 12K / 15K
Same board (requester local memory)	180 ns	180 ns
Same board (available through other CPU on the same dual CPU data switch)	193 ns	193 ns
Same board (other side of data switch)	207 ns	207 ns
Other board	240 ns	–
Other board (CDC hit)	–	333 ns
Other board (CDC miss)	–	440 ns

Similarly, memory bandwidth is also affected by the location of the requesting CPU and the target data. Table B-3 shows maximum memory bandwidth for several types of access. With random distribution of data across the whole system memory, the maximum overall system bandwidth should be somewhere between the “all local” and “all remote” values.

It is easy to see from the latency and bandwidth tables that performance gains can be achieved if the operating system is aware of the locality of memory and can explore it by carefully placing memory pages requested by CPU processes.

Table B-3 Maximum memory bandwidth for Sun Fire servers

Memory Access		Sun Fire 3800-6800	Sun Fire 12K	Sun Fire 15K
Same board as requester	one Uniboard	6.7 GB/s	6.7 GB/s	6.7 GB/s
	system aggregate ("all local board access")	9.6 GB/s	60.3 GB/s	120.6 GB/s
Separate board from requester	one Uniboard	2.4 GB/s	2.4 GB/s	2.4 GB/s
	system aggregate ("all remote board access")	9.6 GB/s	21.6 GB/s	43.2 GB/s

References

[1] Alan Charlesworth, “The Sun Fireplane System Interconnect,” *SC2001 conference paper*, Nov 2001, http://www.sc2001.org/papers/pap.pap150.pdf.

[2] Alan Charlesworth, “SMP Interconnect @ Sun”, *Proceedings of the Spring 2002 SuperG Conference*, 2002, Sun Microsystems, Inc.

[3] Alan Charlesworth, “The Sun Fireplane Interconnect”, *IEEE Micro*,Jan–Feb 2002, pp 36-45.

[4] *Sun Fire™ 15K System - Overview*, Nov 2001, Part No. 806-3509-10 (V2), Sun Microsystems, Inc. http://docs.sun.com/db/doc/806-3509-10?q=806-3509

[5] *Sun Fire™ 6800/4810/4800/3800 Systems Overview*, Apr 2001, Part Number 805-7362-11, Sun Microsystems, Inc. http://docs.sun.com/db/doc/805-7362-11?q=805-7362-11

[6] Alan Charlesworth, “Starfire: Extending the SMP Envelope,” *IEEE Micro*, Jan–Feb 1998, pp 39-49, http://www.sun.com/servers/white-papers/starmicro.pdf?rendition=pdf.

[7] David Culler and Jaswinder Singh, *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco, 1999.

[8] Ben Catanzaro, *Multiprocessor System Architectures*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[9] Kevin Normoyle, Zahir Ebrahim, Bill VanLoo, Satya Nishtala, “The UltraSPARC Port Architecture,” *Hot Interconnects III conference paper*, August 1995.

[10] Tim Horel and Gary Lauterbach, “UltraSPARC III: Designing Third-Generation 64-Bit Performance,” *IEEE Micro*, May–June 1999, pp 73-85, http://dlib.computer.org/mi/books/mi1999/pdf/m3073.pdf.

[11] *Sun Fire™ Link Systems Overview*, Part Number 816-0697, Sun Microsystems, Inc, 2002.

Sales Offices

Africa (North, West and Central): +33 1 30674680
Argentina: +5411-4317-5600
Australia: +61-2-9844-5000
Austria: +43-1-60563-0
Belgium: +32-2-704-80-00
Brazil: +55-11-5187-2100
Canada: +905-477-6745
Chile: +56-2-3724500
Colombia: +571-629-2323
Commonwealth of Independent States: +7-502-935-8411
Czech Republic: +420-2-3300-9311
Denmark: +45 4556 5000
Egypt +202-570-9442
Estonia: +372-6-308-900
Finland: +358-9-525-561
France: +33-01-30-67-50-00
Germany: +49-89-46008-0
Greece: +30-1-618-8111
Hungary: +36-1-202-4415
Iceland: +354-563-3010
India: +91-80-5599595
Ireland: +353-1-8055-666
Israel: +972-9-9513465
Italy: +39-039-60551
Japan: +81-3-5717-5000
Kazakhstan: +7-3272-466774
Korea: +822-3469-0114
Latvia: +371-750-3700
Lithuania: +370-729-8468
Luxembourg: +352-49 11 33 1
Malaysia: +603-264-9988
Mexico: +52-5-258-6100
The Netherlands: +31-33-450-1234
New Zealand: +64-4-499-2344
Norway: +47-2202-3900
People's Republic of China:
    Beijing: +86-10-6803-5588
    Chengdu: +86-28-619-9333
    Guangzhou: +86-20-8755-5900
    Hong Kong: +852-2802-4188
    Shanghai: +86-21-6466-1228
Poland: +48-22-8747800
Portugal: +351-21-4134000
Russia: +7-502-935-8411
Singapore: +65-438-1888
Slovak Republic: +421-7-4342 94 85
South Africa: +2711-805-4305
Spain: +34-91-596-9900
Sweden: +46-8-623-90-00
    German: 41-1-908-90-00
    French: 41-22-999-0444
Switzerland: +41-1-908-9000
Taiwan: +886-2-2514-0567
Thailand: +662-636-1555
Turkey: +90-212-236 3300
United Arab Emirates: +9714-3366333
United Kingdom: +44-1-276-20444
United States: +1-800-555-9SUN OR +1-650-960-1300
Venezuela: +58-2-905-3800
Worldwide Headquarters:
    Sun Microsystems, Inc.
    4150 Network Circle
    Santa Clara, CA 95054 U.S.A
    Phone: 650-960-1300 or 800-555-9SUN
    Internet: www.sun.com