Sun Systems and their Caches

Sales Tactical Engineering
June 15, 1990

Table of Contents

Appendix

1.0 What This Document Covers

This document:

- Describes the different kinds caches found on Sun computers.
- Discusses performance implications of the different caches.
- Demonstrates the effects of moving data in various ways on systems with a cache.

You should have a good understanding of either the Sun-3 or Sun-4 architecture before reading this document. A good overview of the Sun-3 architecture is described in the *Sun-3 Architecture: A Sun Technical Report.* All Sun sales offices have copies of that document.

2.0 What is a Cache?

A typical cache is very high-speed memory that sits between the CPU and the system's main memory (RAM). The cache contains copies of ''frequently requested'' data from main memory.

When the CPU requests to read data from main memory, the cache first checks to see if it contains the requested data. If so, it very quickly satisfies the request. If the cache does *not* have the requested data, it retrieves the data from main memory, passes the data on to the CPU, and saves the data in case that same data is requested again soon.

Similarly, on a CPU write, the data is stored in the cache very quickly and the CPU is allowed to continue executing further instructions. This takes much less time than it would have if the CPU had waited for the data to be written to main memory.

So why doesn't *all* of main memory use the fast cache memory?

- **Cost**: This faster cache memory is very expensive. If all of main memory were as fast as our cache memory, our systems would be **much** more expensive. With a 64k or 128k-byte cache, systems are much faster, but not very much more expensive.
- **Space:** The faster cache memory chips are static-rams (SRAMs), while the main memory chips are dynamic-rams (DRAMs). While SRAMs are much faster than DRAMs, they store much less data per chip. There is just not enough physical space on the CPU boards and memory boards to hold enough SRAMs for the amount of memory used on Sun systems.
- **Power:** The faster SRAMs consume much more power than the DRAMs. If only SRAMs were used on a system, less power would be available for other uses on the system, or a larger power supply would be needed.

3.0 Cache Flavors

There are many different flavors of caches that can be found in systems. For example, when choosing a cache architecture, it might be one or more of the following:

- a **write-through** cache

- a **write-back** cache
- a **physical address** cache
- a **virtual address** cache
- a **central** or **CPU cache**
- an **I/O** cache

3.1 Write-through and Write-back caches

A cache can be either a **write-through** or a **write-back** cache.

When a cache is a **write-through** cache, data written to the cache is immediately written to main-memory. In this type of cache, data in the cache and main memory will always have the same values.

If a cache is a **write-back** cache, data is written to main memory only under certain circumstances (described later). In this type of cache, data in the cache and main memory will frequently contain differences.

The negative side of **write-through** caches is that data is transferred between the cache and main memory very often which slows down system performance. Many times when the CPU requests to write the data from the cache, it must wait for the previous cache write request to complete before the current cache write is satisfied. This happens when the CPU requests "back-to-back" writes to the cache. The delay caused by back-to-back writes to the **write-through** cache can be helped by increasing the complexity of the cache, and adding one or more **write buffers**. For example, if a cache has one write buffer, there would be no delay after the CPU performed two successive writes. However, if there is a third write request immediately after the second successive write, it is likely that the CPU will be delayed until the first cache-to-memory write is completed.

On the positive side of a **write-through** cache, the kernel needn't be concerned with the cache and main memory remaining consistent -- it will always be consistent.

The negative side to **write-back** caches is that the kernel must make sure the cache is *flushed* at various times to retain consistency between the cache and main memory. The positive side to **write-back** caches is that the CPU doesn't have to wait for the cache to update main memory after every write as in a **write-through** cache. This increases the system speed significantly.

3.2 Physical and Virtual address cache

A cache can be either a physical or virtual address cache. The type of cache determines if the addresses used by the cache are physical or virtual addresses. All Sun SPARC-based machines use a virtual address cache. An explanation of virtual caches is given in a following section.

3.3 Central Caches and I/O caches

The most common type of cache is the **Central** or **CPU** cache. This is the cache that sits between the CPU and main memory. In some cases, there is more than one central cache. For example, the Motorola 68030 contains a 256-byte cache and the Sun-3/400 series computers also has a 64K-byte physical address cache.

The I/O cache was designed and patented by Sun. It is a new, innovative way to use the cache concept which produces significantly higher performance. In this case, instead of the cache being between the CPU and main memory, the cache sits between all DVMA devices (devices on the VMEbus and ethernet devices), and main memory. This will be described in detail in a following section.

3.4 Other cache characteristics

Other elements of a cache design include:

- The size of the cache.
- The length of a cache line (cache lines are described at the beginning of the next section).
- The speed of communication between the cache and CPU, and between the cache and main memory.
- On **write-through** caches, the number of write buffers the cache has.

The Appendix of this document includes a list of current Sun systems and their characteristics.

4.0 The Central Cache Overview

The central cache fits neatly between the CPU and the Memory-Management-Unit (MMU). *All* read and write requests go through the cache *before* main memory. This includes both requests from the CPU, and DVMA from a device on the VMEbus and ethernet.

- Each cache block[†] on most Sun systems store 16 bytes. This means the least number of bytes that the cache will move between itself and main memory on reads or writes will always be 16 bytes. For example, if a program requests to read 4 bytes from memory, the cache will actually read 16 bytes (a complete cache block) from memory, instead of only the 4 bytes that were requested.
- Each cache block has some *tags,* or information that describes the cache block with which it is associated.
- Only Sun private/onboard memory is cached (type 0 in the MMU). VMEbus memory is *not* cached.
- The SPARCstation 1+ and SPARCstation SLC have 64k byte central caches. The SPARCserver 300 and SPARCserver 400 have 128k byte caches. At this point, all Sun systems have 16 byte cache-lines, except for the SPARCserver 400, which has 32 byte cache-lines.
- On most Sun systems, DVMA data goes through the cache. However, on the SPARCstation 1+ and SPARCstation SLC, DVMA data bypasses the cache and goes directly through the MMU to main memory.

† A cache block is also frequently referred to as a cache *entry* or a cache *line*.

A typical Sun cache:

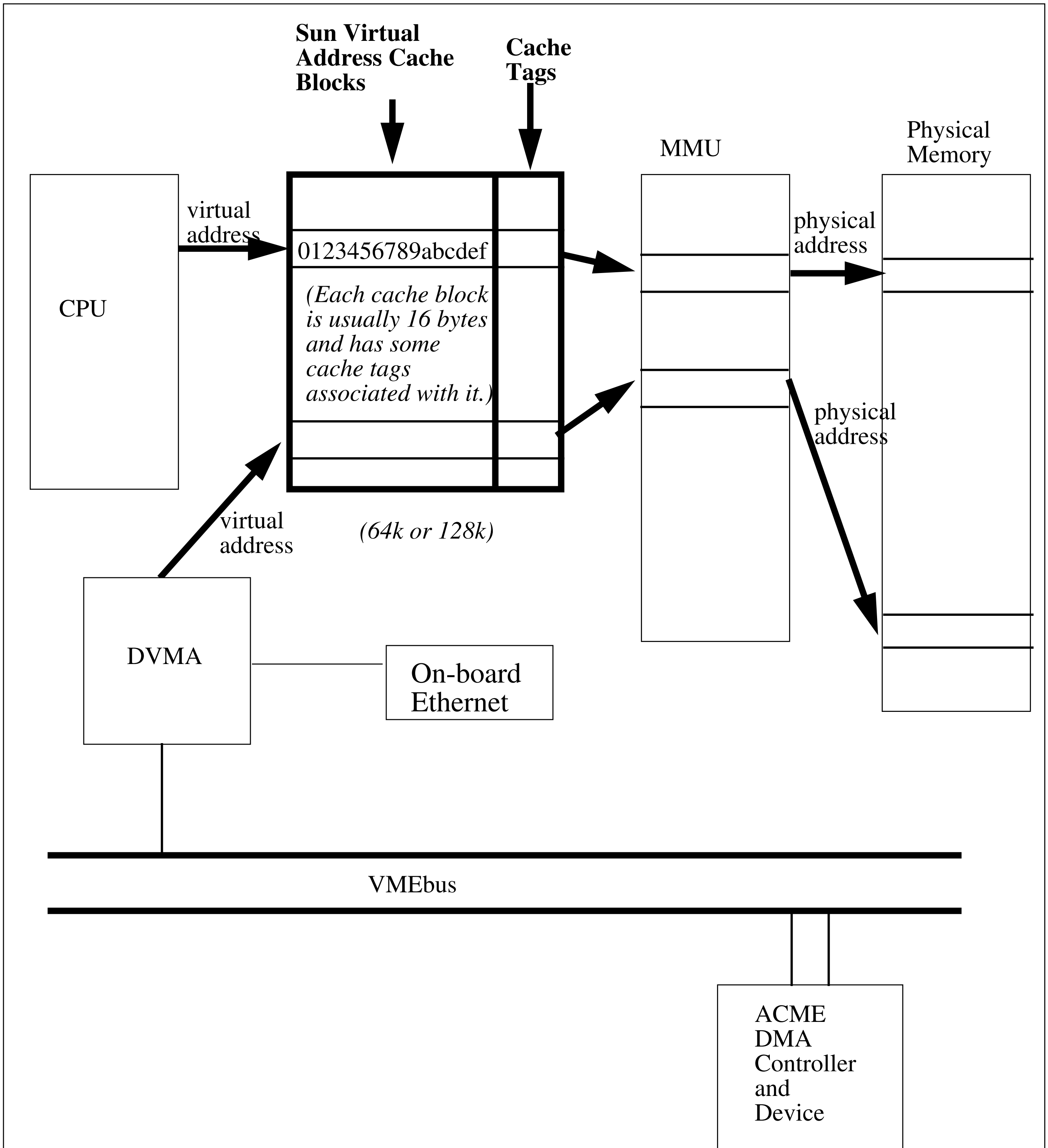


Figure 1: A typical Sun Cache.

5.0 What makes it a *Virtual Address* cache?

The address that comes out of the CPU and from the DVMA circuitry is *always* a *virtual* address -- not a *physical* address. A*ll* read/write requests from the cache go through the MMU (Memory Management Unit) and at *that* point are translated into physical addresses.

It is possible for the MMU to map multiple virtual addresses to the same physical address. Since a virtual address cache contains data from virtual addresses, this means there could be two different entries in the virtual address cache, with different data for the same physical address. This is described more in the next section, **Aliasing**. However, the kernel takes care of data consistency in the cache and main memory.

6.0 Aliasing

Virtual addressing allows *aliasing*: the possibility of multiple virtual addresses mapping to the same physical address. When using aliasing, the kernel must ensure the cache remains consistent. There are a number of ways to do this. First, look at how the cache determines which location data will be placed:

```
(the virtual address of the data) MODULO (the size of the cache)
```

In other words, for the SPARCserver 400's 128k-byte cache, bits 5-16 of the virtual address determine where the data will be placed in the cache. Bits 17-31 of the virtual address, along with the context of the process, are stored in the cache block's *tags* field:

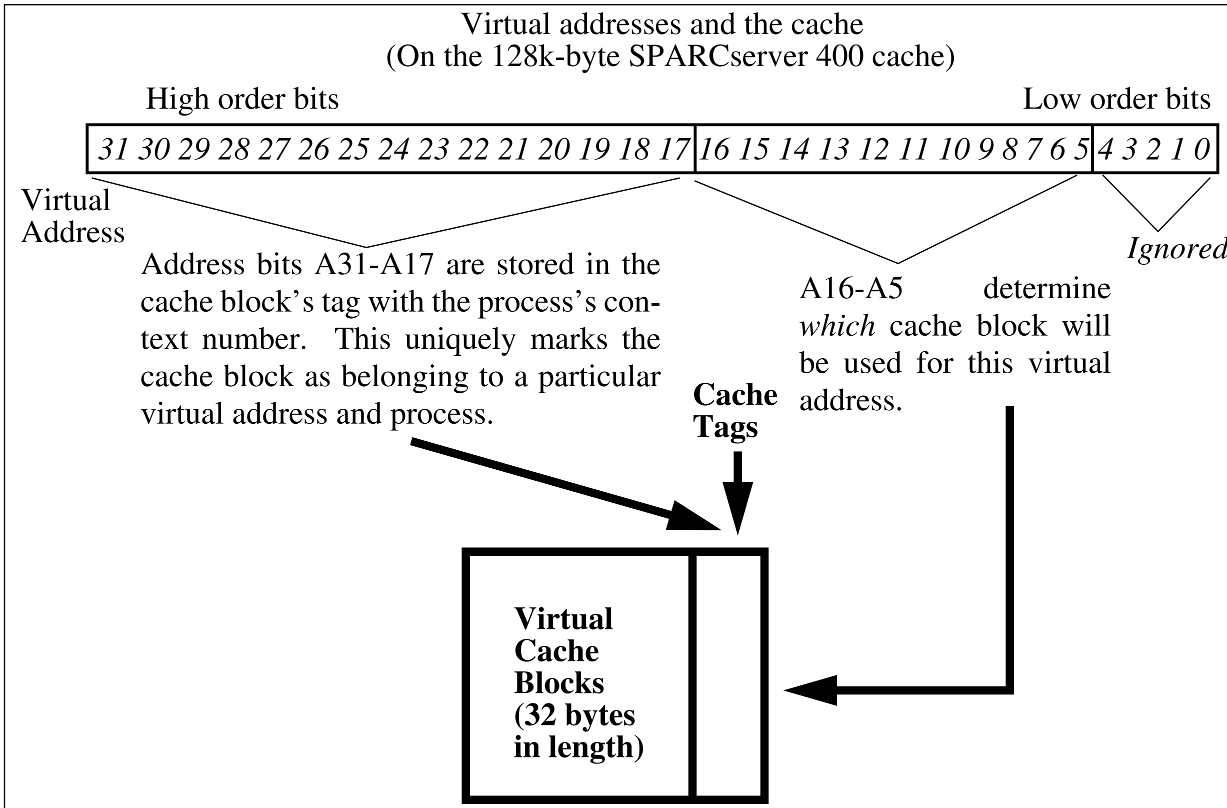


Figure 2: Virtual Addresses and the cache.

The reason bits 0-4 are ignored when the SPARCserver 400 is determining the cache location to use for a particular address is because each cache line holds 32 bytes, so if two virtual addresses are identical in bits 5-16, it doesn't matter if the low-order 5 bits are **00000** (binary) or **11111** (which is 31, decimal) or anywhere in between: the same cache line would be used and would store 32 bytes in that cache line starting at a 32-byte boundary.

The diagram shown next is an example of how the system could look with two virtual addresses (aliases) mapping to the same physical address.

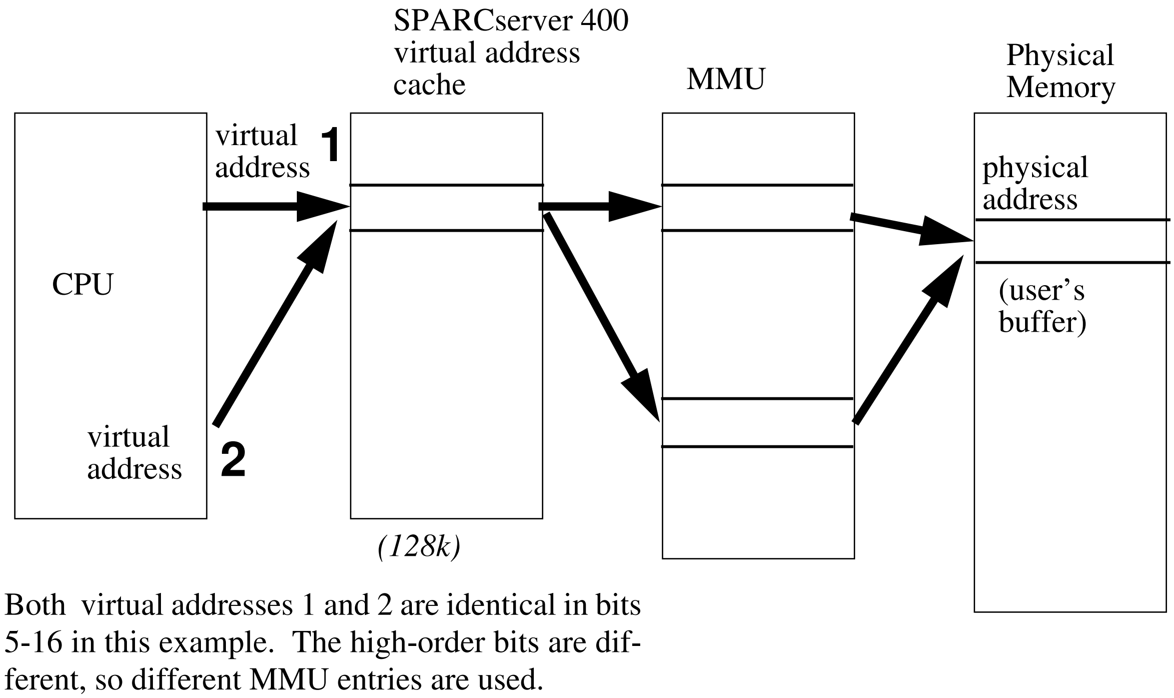


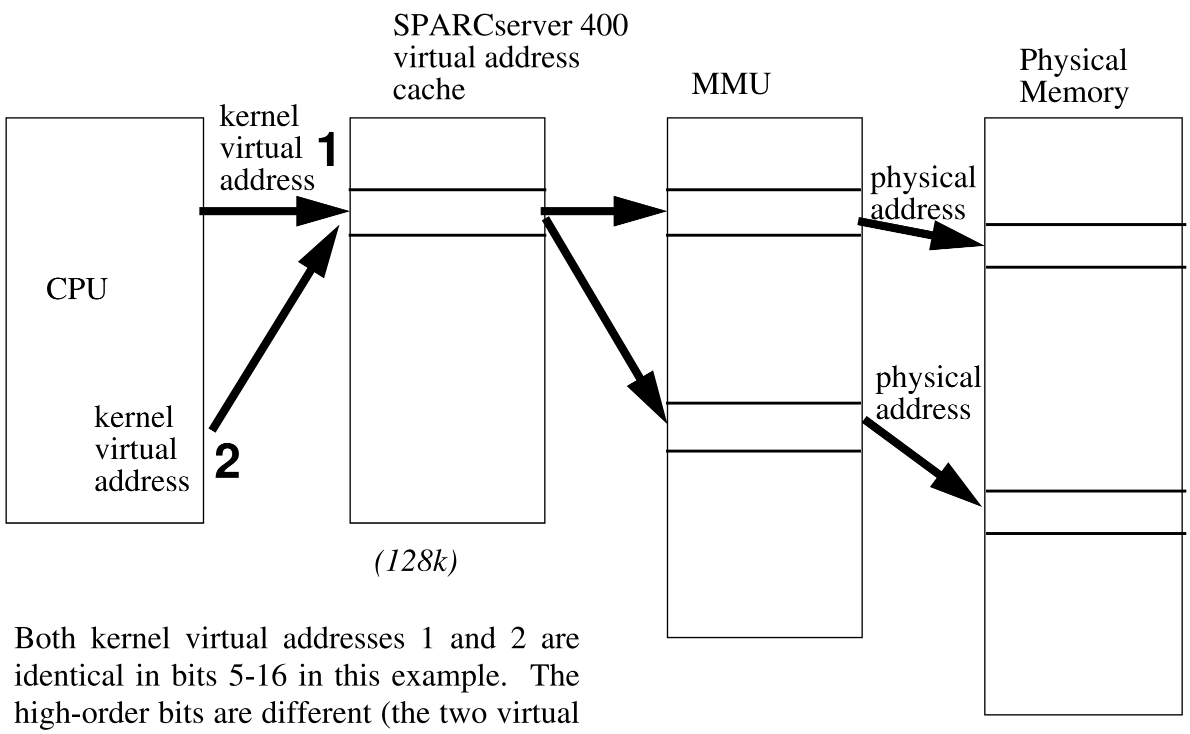
Figure 3: Two Virtual Addresses Mapping to the same cache line.

One example of two virtual addresses mapping onto the same physical address is that of DMA with a raw device driver. In the *raw* interface to device drivers, right before the DMA is done directly to or from a user's buffer, the kernel makes copies of the mappings to the user's buffer and places them in kernel space.

In the diagram above, the low-order bits (A16-A5) of the virtual addresses are identical. In this case, virtual addresses 1 and 2 are a multiple of 128k apart. Consequently, since the cache is 128k, the same cache block is used for the two different virtual addresses.

Since the virtual addresses are different, different MMU entries are used for the two virtual address. If the addresses are aliases of each other, the two virtual addresses end up pointing to the same physical address. For device drivers setting up DMA operations, it is common for the kernel to go out of its way to set up two virtual addresses which map to the same physical addresses, and also map to the same cache line, but use different MMU entries.

Note that it is also possible for two different virtual addresses to map onto the same cache block, but end up mapping to *different* physical addresses:

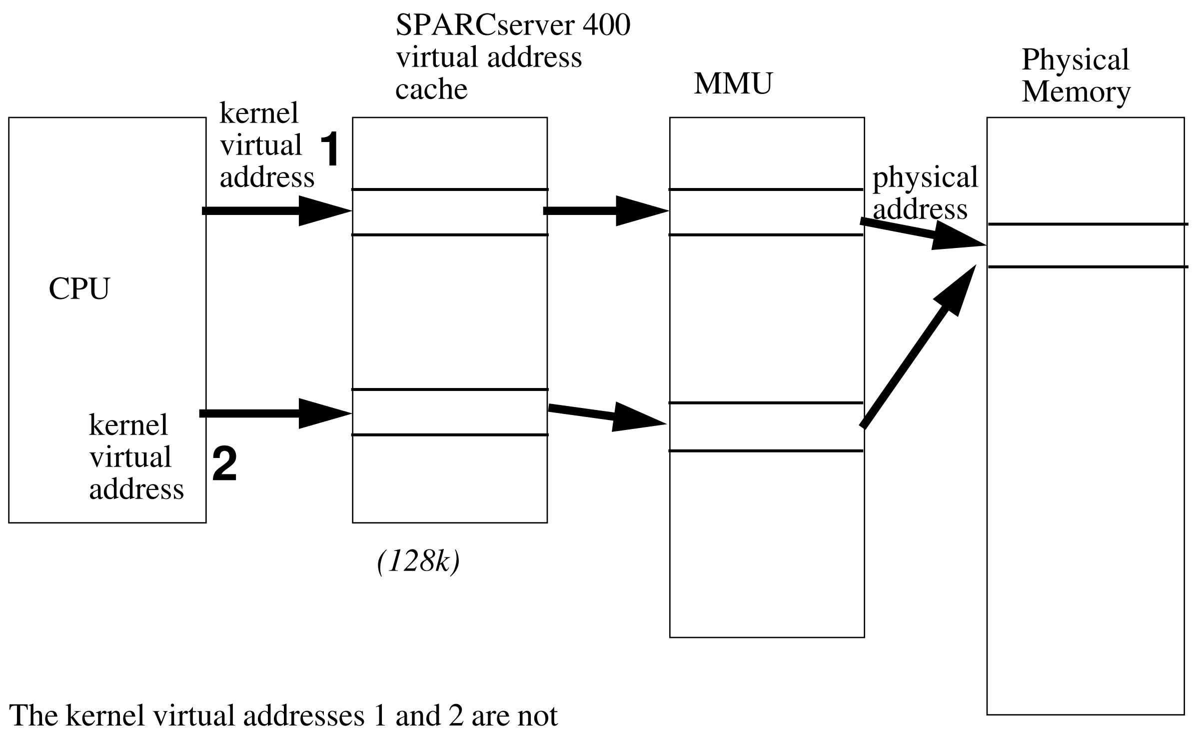


Both kernel virtual addresses 1 and 2 are identical in bits 5-16 in this example. The high-order bits are different (the two virtual addresses are different), so different MMU entries are used.

Figure 4: One cache line that maps to different physical addresses.

Since the low-order bits 5-16 are identical, the same cache block is used for the two different virtual addresses. The Sun cache has special alias detection hardware that ensures the correct cache block will be used with a given virtual address. The cache block will be reused and can be read and written while remaining consistent with physical memory. This case is examined more closely in the section titled **The Cache Write-Back**.

The one other possibility that can occur would probably only pertain to you if you are writing or modifying operating system code: It is also possible to have aliased addresses which map to the same physical memory, but use different cache blocks:



The kernel virtual addresses 1 and 2 are not identical in bits 5-16 in this example. Consequently, different cache blocks are used for the data at these virtual addresses.

Figure 5: Virtual Addresses which map to identical physical addresses.

In the above example, changes made at one cache location are not seen at other cache locations. This could result in potential cache inconsistencies if data is written to one cache block without being seen at the second cache block†. In this case, a second method to guarantee cache consistency can be used by setting the *no-cache* bit in the MMU. This slows system performance and is usually not the best way to ensure cache consistency. Also, note that the *no-cache* bit is available only to the kernel - not to the user.

† However, remember the kernel takes the responsibility of keeping the cache and main memory consistent in these cases.

7.0 The Cache Tags

The cache stores information for *each* cache block. These are called the cache block's *tag*. Below is a description of what a cache tag consists of:

- **Valid** (1 bit): Indicates that the tags and data at the addressed cache block are both valid.
- **Modified** (1 bit): Indicates that the cache block has been modified (written to at least once).
- **Virtual Address** field (from 11 to 15 bits, depending on cache size): Virtual address bits, which, together with the cache address, uniquely identify a cache block in virtual address space. For the 128k byte SPARCserver 400 cache, the Virtual Address field is 15 bits long, and stores address bits A31 through A17. The contents of this field will change every time the particular cache line is allocated to a different address, or is flushed.
- **Protection** (2 bits): *Write Allowed* and *Supervisor Access* protection bits, identical to those in the MMU.
- **Context ID** (CID) field (8 bits on a SPARCserver 400): Identifies to which of the 8 (on a Sun-3) contexts the cache block is associated.

With the above information, the cache can determine whether or not a particular cache block is valid, to which virtual page it refers, and even to which context the cache block belongs.

One more location that cache information is stored is in the MMU. The MMU has one bit known as the *no-cache* bit. If the *no-cache* bit is set for a particular page, then data from that page will not be stored in the cache.

8.0 The Cache Flush

Flushing refers to making the data in the cache and main memory identical.

There are at least three types of cache flushes: a page flush, a segment flush, and a context flush. On a SPARCserver 400, there is an additional type of cache flush called *user flush*, which flushes all cache blocks accessed by a particular user.

On execution, for example, a page flush would check the cache for blocks that contain data from the requested page. Then:

- The matching cache entries are written to main memory *if* they have been modified since the time they were loaded inside the cache. One of the cache tags, the *modified* bit, keeps track of whether or not an block has been modified.
- Those matching entries in the cache are invalidated so that the next access to that virtual address will cause a read from main memory.

9.0 A Cache Example

Below is a possible scenario of what could happen with an empty cache. Assume all of the requests below are made by the same process running in the same context:

1. The CPU requests to *read* the contents of location 0x2345.
2. It is a cache ''miss'' -- the contents of virtual address 0x2345 are not in the cache, since we are assuming in this example that the cache starts out containing no data from main memory.
3. The cache reads the contents of virtual address 0x2345 from main memory, sends the result to the CPU, and stores it.
4. A little later, the CPU once again requests to *read* the contents of location 0x2345.
5. This time it is a cache ''hit'' -- the cache contains the data at this virtual address and the cache tags for this cache block match virtual address 0x2345 for this context. The cache very quickly sends the contents of location 0x2345 to the CPU.
6. The CPU requests to *write* the contents of that same location 0x2345.
7. It is a cache ''hit'' again. The value from the CPU is written to the cache very quickly, but the cache does *not* write the value to main memory (because this is a *write-back* cache). The *modified* cache tag for this cache block is set. The cache and main memory have different values now.
8. The CPU once again requests to *write* the contents of location 0x2345.
9. It is another cache ''hit'' and only the cache contents are changed. The *modified* cache tag for this block is already set.
10. The CPU requests to *read* from location 0x2345 once again.
11. It is a cache ''hit''--the value that was last written to the *cache* is sent to the CPU. The cache does not access main memory.

Remember that the address 0x2345 referred to above is a *virtual* address, not a *physical* address. Also remember that if a process running in a different context were to request the same virtual address, it would be a cache *miss* since in this case, the cache tags would not match.

10.0 The Cache Write-Back

Modified (sometimes referred to as ''dirty'') data values in the cache are not written to main memory until necessary. Sometimes the kernel explicitly requests part of the cache to be written back to main memory. Other times the cache itself will decide to write some entries to main memory. When a cache block is being written to main memory, the cache is said to be performing a **write-back** (or *flushing* that block).

There is one situation where the cache will decide on its own that it must write an block to main memory. Below is an example of this with a diagram of this example on the next page:

1. The CPU *reads* 4 bytes which causes the cache to be loaded with the requested data from main memory (labeled in the diagram as virtual and physical address **1**). Assume these four bytes contained the characters **TREE**.
2. The CPU then does a *write* to modify the data that was just loaded into the cache. For example, assume **MARE** is the new data the CPU writes to the cache. This would cause **TREE** to be replaced with **MARE**.

3. The CPU then requests to *read* from (writing would have the same effect) a *different* virtual address (labeled in the diagram as virtual address **2**), which by coincidence, *uses the same cache location that the read and write in the above two steps used.* Remember that the cache is only a small percentage of the size of main memory, so reads and writes from **many** different virtual addresses will map onto the *same* cache location. For example, on SPARCserver 400's with a 128k-byte cache, *all* virtual addresses that happen to have the same low-order 17 bits (bits 0-16) will map onto the same cache block.
4. The cache realizes that this requested cache block is *dirty*, so before it fills that cache block with the newly-requested data, it saves the old cache block temporarily in a reserved buffer. Then the requested data from step 3 is read from main memory into the cache (**HUNT** in the diagram). While the CPU is continuing on, the cache then takes the time to write the old temporarily saved cache block to main memory.

After the 4 steps are finished, the cache and memory would look like this:

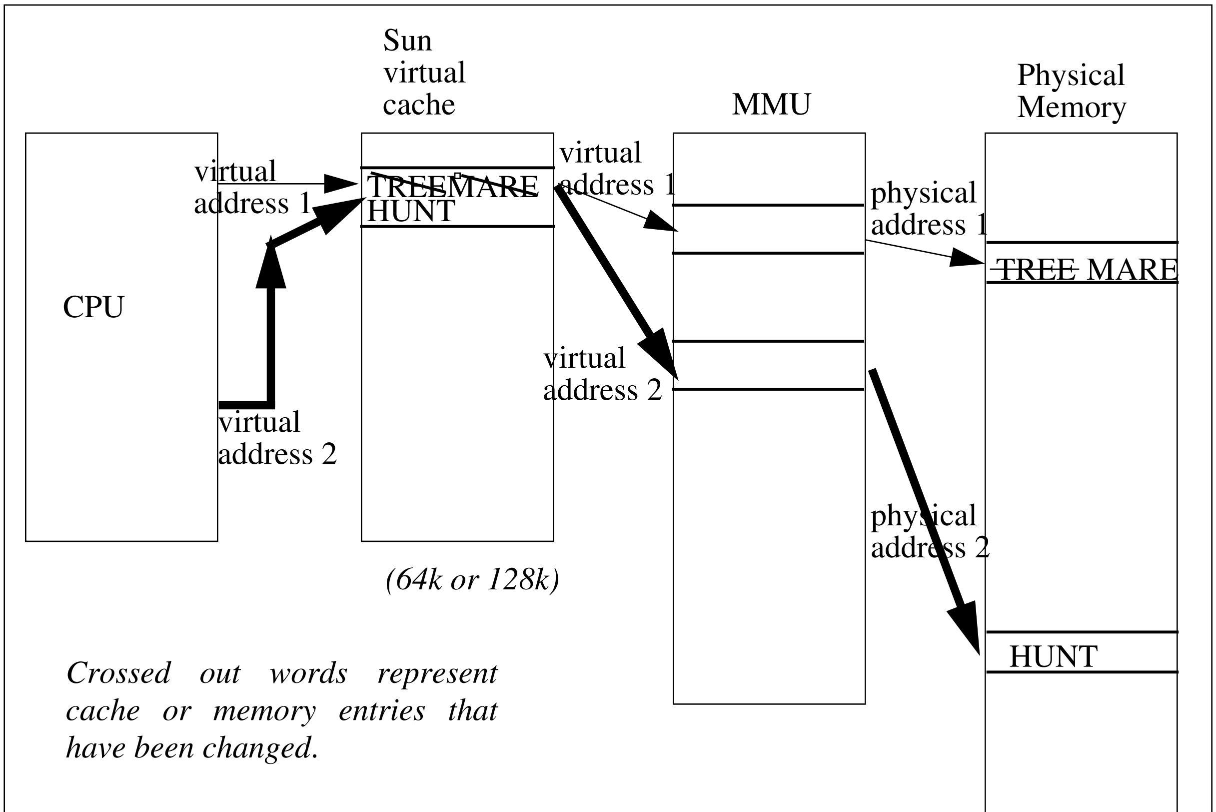


Figure 6: A cache example.

The cache remembers the high-order bits of the virtual address (they are part of the cache *tags).* When a second request comes in and ''hits'' the same cache block, the cache checks to see if the data for that block is the same virtual address as the last time a ''hit'' at that cache location occurred.

For example, notice in the figure how one cache block can refer to different locations at different times. The first few times the cache is referenced in this example, it will point to the **TREE** and **MARE** data location (labeled as virtual address **1** in the diagram). After the write-back of this location completes, and virtual address **2** is requested, that same cache location will then begin referring to **HUNT**, since the cache tags for that cache block will be changed.

The kernel occasionally explicitly requests to flush portions of the cache. For example, when a process is about to be swapped out of memory to disk, the kernel requests to flush all cache entries that are related to that process. *T*he reason this is done before the process is swapped out has to do with the fact that the cache assumes *every* valid virtual cache block refers to a valid virtual *and* physical memory page[†].

† If the kernel allowed the cache to *write-back* to a page that was paged-out on disk, it would fail and receive a *write-back* error. The cache does not work if a block is dirty and the physical page it refers to is not valid.

11.0 The I/O Cache

The SPARCserver series computers incorporate the new I/O cache. The I/O cache sits between the I/O devices and main memory:

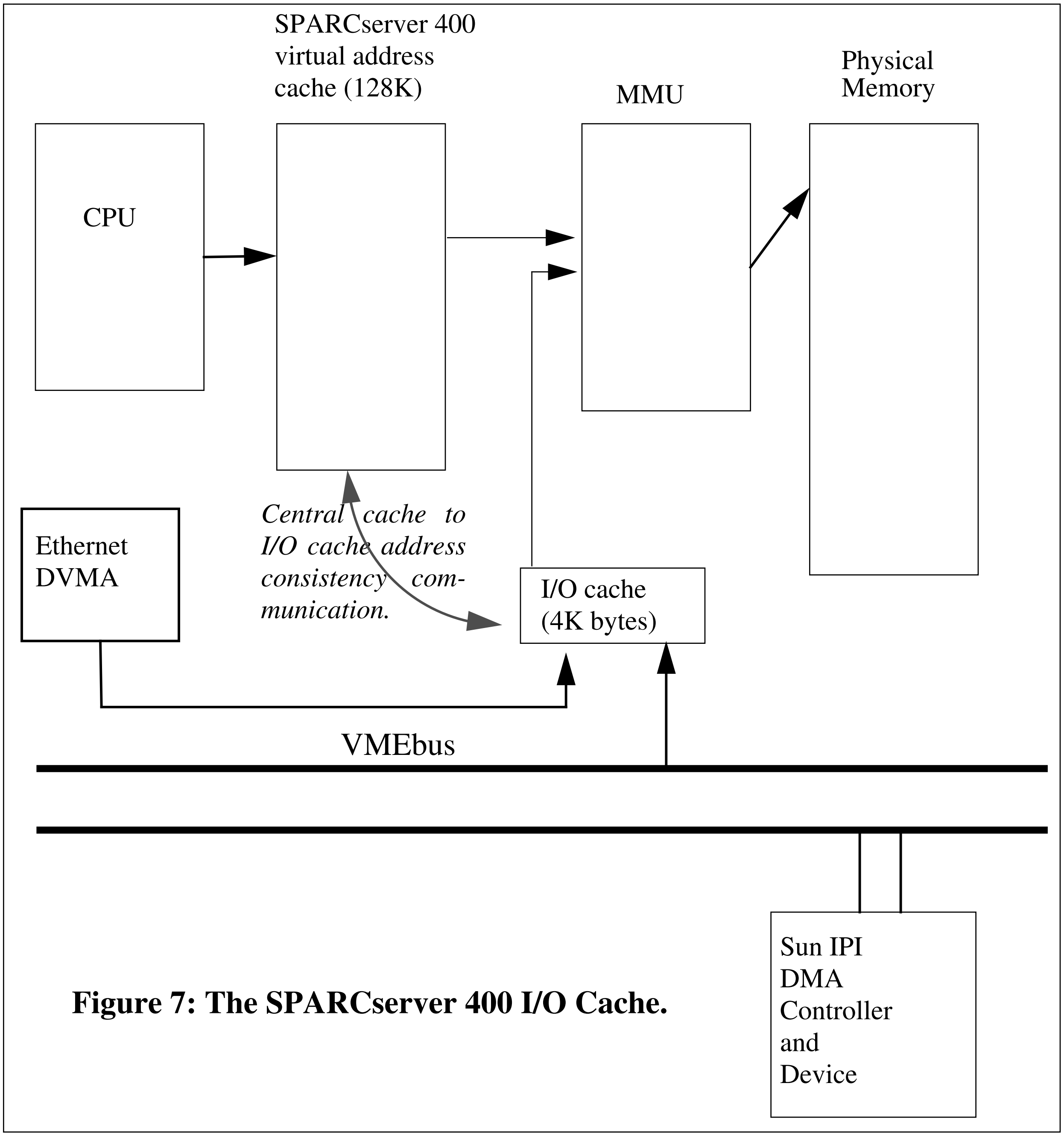


Figure 7: The SPARCserver 400 I/O Cache.

At first glance, it may not appear that adding an I/O cache would change performance. After all, in systems *without* an I/O cache, data coming from I/O devices go through the central cache. How could adding one more cache devoted just to I/O devices help?

11.1 Central Cache Access: I/O versus the CPU

If you closely watch what happens in a system without an I/O cache, you would see that while a controller is doing DVMA to a system, *each and every* DVMA transfer needs exclusive access to the central cache. It makes sense that the central cache would allow only one device to access it at a time: either the CPU or DVMA requests. Consequently, during the DVMA transfer to the central cache, CPU requests to the cache are temporarily suspended. When the DVMA transfer completes, the CPU is once again allowed access to the central cache.

For example, without an I/O cache, if a controller requests to make 8 *word* (4-byte) transfers (for a total of 32-bytes transferred), the CPU would be "taken off" the central cache bus all 8 times for the duration of each transfer. That turns out to be a significant delay and interruption. With the Sun I/O cache on the SPARCserver 400, this is reduced from 8 interruptions and delays into only one, dramatically improving both CPU and I/O performance.

11.2 Central Cache Allocation: I/O versus the CPU

Another significant benefit of the I/O cache is the reduction in the number of cache writebacks and flushes. Without an I/O cache, all DVMA data effectively competes with the CPU for *allocation* of central cache lines.

For example, if a DVMA device requests to *write* to CPU memory on a system without an I/O cache, the cache line may already be in use. Consequently, that cache line would have to be written to memory before loading it with the DVMA data. During this *cache writeback,* the CPU would be delayed. Later, the CPU may want to use that same cache line, but due to the DVMA data using that cache line, the CPU would have to be delayed again while waiting for the cache line to be written to main memory. With the Sun I/O cache, the DVMA data is cached in the I/O cache, and from there, the data is eventually moved directly to main memory, at high speed, bypassing the central cache.

11.3 Performance numbers: With and without the I/O Cache

Systems with I/O caches allow the CPU to spend *much* less time on I/O operations. A SPARCserver 400 system without an I/O cache, performing 4.2 megabytes/second I/O transfers would stall the CPU a *minimum* of 26.1% of the time for 32-bit transfers and 44.5% of the time for 16-bit transfers. In contrast, the SPARCserver 400 with the I/O cache, performing at the 4.2 megabytes/second I/O rate, drops that "CPU minimum stall-rate" to only 5.21% of the time![†]

11.4 I/O Cache Overview

How does the I/O cache do this? The SPARCserver 400 I/O cache has 32-byte cache lines. When a DVMA device requests to *write* to the SPARCserver 400 memory, the I/O cache receives the request and stores the data while remembering (via its I/O cache tags) where the data was targeted to be written. This happens without any interruption to the CPU. This continues until finally a 32-byte cache line is full. At this point, the I/O cache interrupts the CPU and performs a high-speed data transfer directly to main memory. To insure data integrity with the central cache, the I/O cache also communicates with the

† Hsieh, Wei and Van Loo, *A Cached System Architecture Dedicated for the System IO Activity on a CPU Board,* Proceedings of the International Conference on Computer Design, October 2, 1989, pp. 518 - 522.

central cache and informs it about this transfer. For example, if the cache-line for the DVMA'ed addresses are in the central cache, the CPU cache invalidates its corresponding cache line.

Note that while the I/O cache is performing the high-speed transfer to main memory, it can simultaneous accommodate new DVMA requests. This is an additional feature that also helps increase system performance.

Similarly, for DVMA requests from controllers to *read* from the SPARCserver 400 memory, the I/O cache logic causes a complete 32-byte cache-line to be read into the I/O cache. During the DVMA requests to read those 32 bytes, the DVMA controller will not interrupt the CPU.

11.5 I/O Cache Implementation

On the SPARCserver 400 series computers, the I/O cache is 4K, while each I/O cache-line is 32 bytes in length. There are 128 cache-lines in the I/O cache, and the DVMA address space is 1 megabyte so the CPU boards' DVMA circuitry will respond to all VMEbus requests in range 0 to 1 megabyte.

The I/O cache evenly divides up the VMEbus DVMA space:

```
1024k DVMA space / 128 cache-lines = 8K DVMA addresses per cache line
```

What this means is that each 8k page in the DVMA address space is allocated one cache-line. For example, DVMA addresses in the range 0-7k will use cache-line 0, DVMA addresses in the range 8-15k will use cache-line 1, etc. If a disk controller is doing an 8k DVMA transfer (assuming the transfer is on an 8k-boundary), the same I/O cache line is used repeatedly. The I/O cache has cache tags, similar to the central cache, which keep track of the full address that a particular cache-line is referring to.

The assignment of unique DVMA page addresses to unique I/O cache blocks is important for the following reason. Each DVMA device is assigned to use page addresses unique from those of any other DVMA device. Consequently, an arbitrary configuration of DVMA devices can transfer data in an arbitrary sequence to unique I/O cache blocks. There is never any interference between devices within a given I/O cache block.

The ethernet DVMA is a special case and uses cache line 127 for DVMA reading, and cache line 126 for DVMA writing. In addition, the ethernet interface is granted the highest priority when its request is competing with other pending I/O devices. Once the ethernet interface gains access to the I/O cache, it will be allowed to fill up or unload its on-chip FIFO to minimize the possibility of packet underrun or overrun.

Below is a diagram of how the I/O cache is allocated:

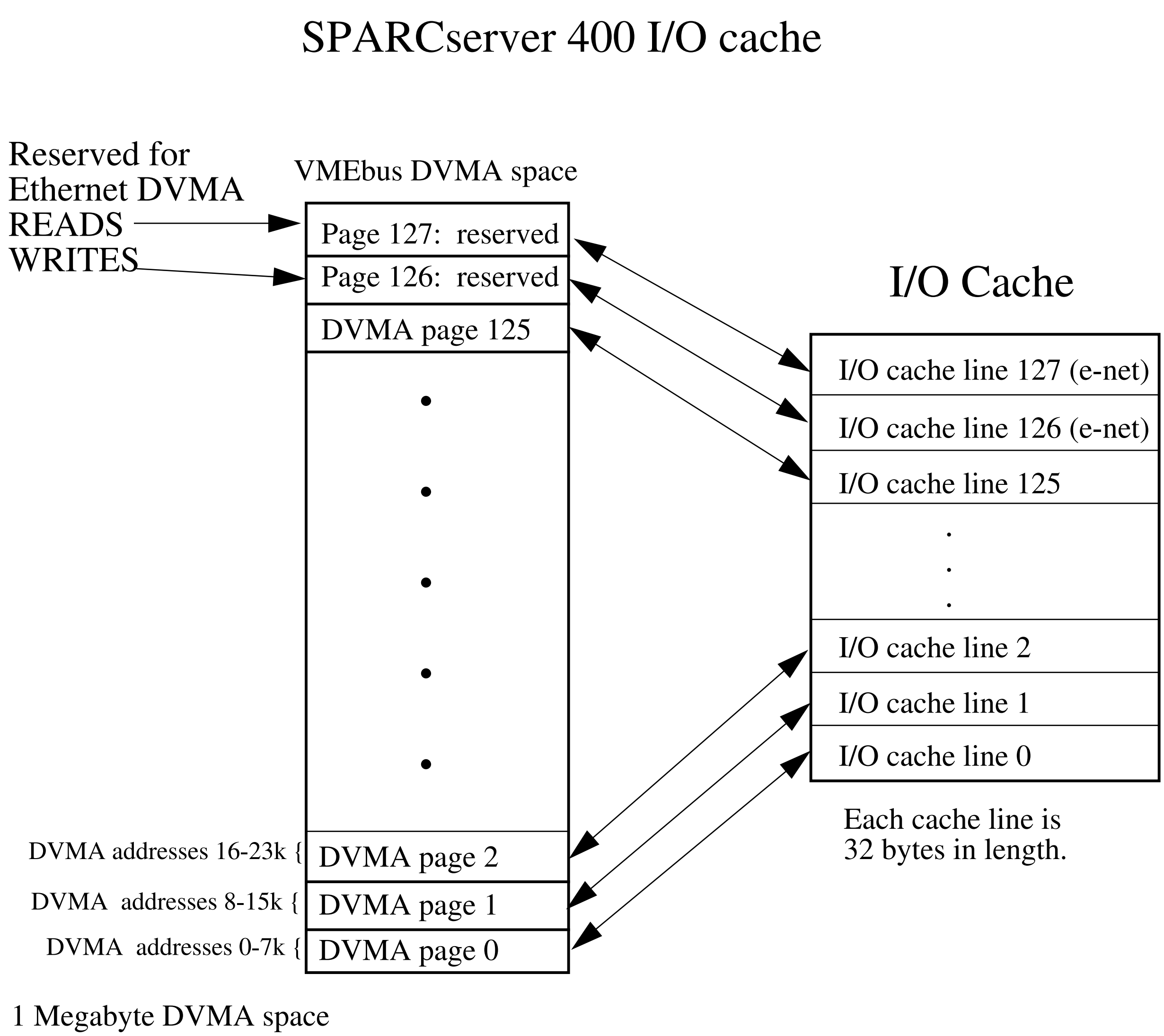


Figure 8: I/O cache allocation.

11.6 I/O Cache Summary

The result of this cache design is a quite a dramatic improvement in both CPU and I/O performance. At the same time that the higher I/O performance is being achieved, the CPU is interrupted much less often, resulting in much higher CPU performance.

12.0 Moving Data in Cache Systems: Performance Considerations

There are some rare circumstances that you may want to avoid when writing programs for systems with caches. The hit rate on the cache is typically very high, but it is possible to write programs with low hit-rates.

12.1 Central Cache Performance Considerations

For example, assume your user program needs to copy 64k of data from one buffer location to another. The buffers might be defined in your program as:

```
1. char buf_from[64*1024];
2. char buf_to[64*1024];
```

Note that it is fastest to use *bcopy* when transferring a large amount of data from one memory location to another both in user programs and inside the kernel. (*Bcopy* is documented under *bstring(3)* in the *Sun UNIX Interface Reference Manual*.) However, to illustrate a point, assume the data copying is done in a user program, and *bcopy()* is not used. Inside your program, you copy bytes from **buf_from** to **buf_to**:

```
3. register char *p1, *p2;
4. register int i, j;
5.
6.    p1 = buf_from;
7.    p2 = buf_to;
8.    for (j=0; j < 64*1024; j++) {
9.           *p2++ = *p1++;
10.   }
```

On machines with 64k-byte virtual address caches, the above code causes a high number of cache misses, due to a number of coinciding events:

- The C-compiler allocated storage for **buf_from** and **buf_to** right after each other and the size of **buf_from** is a multiple of the cache size -- 64k.
- The Sun cache is organized as a *direct mapped virtual address cache*. This means that the cache location into which data will be placed is:

  (the virtual address of the data) **MOD** (the size of the cache)

  Actually, the 4 lowest order bits in the virtual address are not used in this calculation on a SPARCstation 1+. This is because each cache block contains 16-bytes: the least number of bytes that the cache will move on reads or writes is always 16 bytes - not just 1 byte at a time.

So in the program, above, here's what happens:

1. On line 9, one byte is read from **buf_from** and since it wasn't in the cache, that byte and the next 15 are read from main memory and stored in the cache.

2. The CPU requests to write one byte to the address at **buf_to.** Because **buf_to** is exactly 64k away from **buf_from**, the same cache block will be used -- the 16 bytes that had previously been read into the cache will be invalidated, and the 16 bytes from **buf_to** will be read into the same cache block. The first byte of this cache block will then be written with the first byte from **buf_from**, making that cache block ‘‘dirty’’.
3. The CPU requests to read the next address -- at **buf_from+1**. This will once again be in the same cache block, so that ‘‘dirty’’ cache block of 16 bytes will be written to main memory. Then the same 16 bytes from **buf_from** to **buf_from+15** will be read into the cache.

As you can see, a lot of time is spent loading and unloading the cache with data it never uses more than once. The point of a cache is to load data that it will use a number of times. Also, remember:

- Since each cache block is a block of 16 bytes, *each* cache read or write-back must read or write not just the amount of data the CPU explicitly requested to read or write, but a full 16 bytes.

It is easy to make the program on the previous page run much faster -- just make the buffers offset by at least a cache block of 16 bytes. This can easily be done by adding 16 bytes to the **buf_from** array:

```
char buf_from[64*1024+16];
char buf_to[64*1024];
```

When this code segment is run with the above declarations, it performs much faster (for example, 40% faster on a SPARCstation 1+).

12.2 I/O Cache Performance Considerations

When writing programs that perform I/O on the SPARCserver 400, make sure the beginning address of the transfer is on a 32-byte boundary: The SPARCserver 400 kernel will only cache data, in the I/O cache if it is aligned on a 32-byte boundary. *Non-aligned data I/O activity will not be served by the I/O cache, which causes a severe performance penalty.*

To guarantee that a buffer is aligned properly, use the *valloc()* library call instead of *malloc().*

13.0 Summary

An overview of various types of caches that can be implemented in a system, along with some of the trade-offs have been shown. Sun chose unique approaches for cache designs in their systems. The choice of implementing central and I/O caches in varying sizes, performances and cost, have proven to be very effective.

14.0 Where to go for more information

For more information on the cache, see:

The article *"Maximizing Performance by Choosing Best Memory",* by William Van Loo, in the August 1, 1987 issue of *System Design/Computers and Computer Systems.*

A Cached System Architecture Dedicated for the System IO Activity on a CPU Board, by Hsieh, Wei and Loo in the proceedings of the 1989 International Conference on Computer Design, October 2, 1989.

Cache Memories, by A. J. Smith, Computing Surveys, Volume 14, No. 3, September 1982, pp. 473-530.

A Case for Direct-Mapped Caches, by Mark D. Hill, Computer, IEEE, Dec. 1988, pp 25-40.

Coherency for Multiprocessor Virtual Address Caches, by James R. Goodman, in the proceedings from the Symposium on Architectural Support for Programming Languages and Operating Systems, IEEE Computer Society, 1987, pp. 72-81.

Table 1
Summary of Sun CPU Architectures

	Central Cache Size	Central Cache Type	Central Cache Type	Cache Line Size	I/O Cache Size	MMU Page Size	MMU Hardware Contexts
SPARCserver 400	128k	Write-back	Virtual	32 bytes	4k	8k	64
SPARCserver 300	128k	Write-through	Virtual	16 bytes	<none>	8k	16
SPARCstation 1+	64k	Write-through	Virtual	16 bytes	<none>	4k	8
SPARCstation SLC	64k	Write-through	Virtual	16 bytes	<none>	4k	8
Sun-4/200	128k	Write-back	Virtual	16 bytes	<none>	8k	8
Sun-3/200	64k	Write-back	Virtual	16 bytes	<none>	8k	8
Sun-3/400	64k	Write-back	Physical	16 bytes	2k	8k	1