Abstract

In this paper we present the virtual cache-based architecture of the Apollo DN4000 workstation. We discuss the architecture in detail, including the main factors that influenced its final design. Of particular interest is how our use of a virtually tagged, write-through cache made it possible to use a low-cost, off-the-shelf memory management unit and resulted in inclusion of a virtual write buffer. This architecture enables the CPU to execute reads and writes in zero memory wait states.

We discuss the influences of the operating system on the cache architecture, and the subsequent influences of the cache on the operating system to maintain cache coherency. In particular, we show how the way that objects are accessed in Apollo's system led to a write-allocate cache update policy to solve the virtual address synonym problem. We also show how maintaining cache coherency requires special consideration by the operating system during virtual-to-physical address mapping changes, context switches, modification of data without using virtual addresses (e.g., DMA), and processor modification of page table used bits.

Our primary design goal of the DN4000 was to double the CPU performance of the existing Apollo DN3000. We provide performance measurements indicating that our use of a virtually tagged cache and a virtual write buffer allowed us to attain this goal.

1. Introduction

The Apollo DN4000 is a high-performance workstation that fits in the mid-range of Apollo's product family. It is based on Motorola's 25 MHz MC68020 microprocessor, MC68881 floating-point coprocessor, and the MC68851 paged memory management unit (MMU) [Moto85a, Moto85b, Moto86]. The system has a 1 Gigabyte per-process virtual address space, supports between 4 and 32 Megabytes of physical memory, and utilizes an AT-compatible bus to support a wide range of system and user peripheral devices including monochrome and 8-plane color graphics, Winchester disks, and Apollo token-ring and Ethernet network interfaces [Apo87a].

The software system running on the DN4000 is the Apollo Domain® [Leac83, Levi85] which allows each workstation to function autonomously as a single-user machine, in addition to being part of an integrated network of other workstations, file servers, compute servers, printers, etc. to permit a high degree of cooperation and sharing. The operating system itself consists of a common low-level kernel that supports Berkeley 4.2 UNIX®, AT&T System V UNIX, and Apollo Aegis®.

The DN4000 achieves its high performance with the aid of a virtual cache. Of particular interest is how our use of a virtually tagged, write-through cache made it possible to use a low-cost, off-the-shelf memory management unit (MMU) and resulted in inclusion of a virtual write buffer. Virtual caches do present some problems, however. Virtual address synonyms [Smit82], or aliases, can exist when two or more virtual addresses point to the same physical data. Care must be taken to ensure that accesses to synonyms index to the same cache entry, and that data modifications are reflected in accesses to all synonyms of the data.

Also, many memory accesses will not result in MMU activity in a virtually tagged system. Because the MMU performs various housekeeping functions, such as ensuring the validity of virtual-to-physical mappings, the operating system plays a significant role in preventing stale data from residing in the cache; i.e., it helps maintain cache coherency.

The next section discusses the evolution of the DN4000's cache architecture, and the main influences that led to its final design. Section 3 describes the cache architecture in detail and discusses the solution to the virtual address synonym problem. Section 4 describes the functioning of the virtual write buffer and how it allows the processor to...
execute writes in zero memory wait states. Modifications to the operating system to support the architecture are discussed in Section 5, and we conclude with some performance measurements.

2. Evolution of the Architecture
The architecture of the DN4000 was influenced by several factors. Our primary design goal was to create a machine with twice the CPU performance of Apollo's entry-level personal workstation, the DN3000 [Apol87a]. Secondly, we wanted to maintain as much of the DN3000's architecture as possible, including its use of the PMMU. This way, much of the operating system work done for the DN3000 could be reused, and the PMMU would provide a low-cost, low-power, high-density memory management unit.

Both the DN3000 and the DN4000 are based on the Motorola MC68020 microprocessor, MC68881 floating-point coprocessor, and MC68851 paged memory management unit. The DN3000 runs at 12 MHz with one memory wait state. Increasing the clock rate of the MC68020 and using the same memory subsystem would yield some performance increase, but not enough to meet the design goal of doubling the DN3000's performance. Therefore, we developed a virtual cache-based processor/memory architecture. This allowed us to implement a high-performing 25 MHz CPU by circumventing the need to access the lower speed PMMU on every cache access. Consequently, reads that hit in the cache execute in zero wait-states.

Additionally, to achieve zero wait-state writes, a virtual write buffer* was created to reside between the virtual cache and MMU. Figure 1 shows a block diagram of the DN4000's processor/memory subsystem including the virtual cache and the write buffer.

![Figure 1. Processor/Memory Subsystem](image)

Another major influence on the architecture was cost. Specifically, we wanted to add only minimally to the DN3000's cost, while attaining our goal of doubling its performance. This served to further enforce our use of the low-cost PMMU, as well as leading to reusing the DN3000 form factor, and extensive use of surface-mount technology. Cost, board real-estate, and availability of cache tag comparators also placed restrictions on our implementation of the cache, effectively limiting its size.

3. DN4000 Cache Architecture
Cache memories and their benefits have been discussed in great detail in the literature (see [Smit86] for a partial bibliography). In general, a cache may be addressed with either physical or virtual addresses. In a physically addressed cache each processor memory reference is translated by the MMU to yield a physical address from which a cache index and a physical address tag are derived. A virtually addressed cache has its cache index derived directly from the virtual address, and most often will have physical address tags. Caches of the latter type usually will overlap the access to the MMU with the cache fetch to minimize the effects of accessing the MMU on every cache access [Knap85]. In both the above cases the speed of the MMU is critical to the overall performance of the cache and hence the machine itself.

The DN4000 has a cache that is virtually addressed, but additionally has cache tags derived from the virtual address. While this type of design adds complexity to the operating system software to help maintain cache coherency (see Section 5), it circumvents the need to access the MMU on every cache access, and hence allows the use of a lower-cost, lower-speed MMU.

The cache is direct mapped and contains both instructions and data. It is 8 Kbytes in size, organized as 2048 lines of 4 bytes each. In addition to the virtual address tag bits, each line in the cache contains a task alias identifier. This 3-bit field is provided by the PMMU on context switch and reflects one of eight most-recently-used processes in the system. It effectively acts as an approximation to full address space identifiers, and serves to reduce cache flushing due to context switches. The flushing of cache data requires flushing the entire cache; this occurs in a single processor cycle.

The write policy used by the cache is write-through, with write-allocate. On each processor write, both main memory and the cache are updated. If a cache miss occurs then the indexed line is allocated to contain the new data. Section 3.2 discusses why the "write-allocate" cache update policy is necessary to maintain cache coherency in the face of virtual address synonyms.

Each cache entry contains a data field and a tag field. The tag field is composed of two parts:

1. An address tag that holds the upper 19 virtual address bits, plus the task alias field generated by the PMMU on context switch.

*Certain aspects of this technology are subject to patents which are pending with the United States Patent and Trademark Office.
2. A condition code tag that holds:
   a. Information about the validity of each byte in the entry
   b. User/supervisor status
   c. A "modified" bit stating whether the entry has been successfully written to during a previous write operation; this is used to indicate whether subsequent writes to this virtual address will succeed with no wait states.

Figure 2 shows the organization of the virtual cache.

The cache supports the write-allocate policy by containing information about the validity of each byte in cache entries. For example, a single byte write operation by the processor allocates the indexed entry, but only that byte is valid for future references. A reference to an invalid byte results in a cache miss and the entire line is refilled from physical memory. The cache also accommodates all types of write operations including aligned, and misaligned byte, word, and longword references.

The user/supervisor bit in each cache entry provides protection to encached data. Thus, for instance, a user access to data inserted in the cache while the processor was in supervisor mode will result in a cache miss. This then allows the MMU to validate the access.

3.1 Virtual Cache Operation

The validation of each cache reference is done at two levels. The first occurs at the beginning of each cache reference. The processor's virtual address bits <12:2> provide an index to one specific cache entry. Cache hardware compares virtual address bits <31:13> and bits <2:0> (the task alias bits) with the stored address tag in that cache entry.

If the above address tag match occurs, then the second level of cache validation begins. During a read operation, access information from the MC68020 (size information and a function code that indicates user or supervisor access [Moto85]), and the two least-significant virtual address bits are checked against the contents of the stored condition code for that entry. If a condition code match occurs, the microprocessor reads the contents of the cache entry's data field and the processor cycle ends. If a condition code or address tag match does not occur, a cache miss is generated and the microprocessor reads physical memory instead. The MMU then checks to determine whether the reference can be encashed* and if it can, the entire 4-byte cache entry is updated.

If an address tag match occurs on a write operation, the cache entry's byte, word, or longword data field is updated. Hardware also updates the size and access protection information that resides within the condition code tag, and the hardware examines the "modified" bit there. If the "modified" bit is set, the microprocessor completes a zero wait-state write cycle, and the write buffer continues the write operation to physical memory. If the "modified" bit is not set, the microprocessor waits (for MMU access validation) until the write to physical memory completes. If the MMU signals a write-protect or user/supervisor access violation, hardware invalidates the cache entry.

If the address tag match does not occur during a write operation, the microprocessor does the write to physical memory. If the memory update succeeds, and the data is encacheable, the data is placed in the cache. Also, the "modified" bit in the condition code tag is set so that future write operations to this cache line will not have to wait for MMU validation and will be captured in the write buffer.

3.2 Virtual Address Synonyms

Virtual address synonyms, or aliases, exist when two or more virtual addresses point to the same physical data. There are some inherent problems associated with synonyms. The first is the case where multiple synonyms do not index to the same cache line.

Consider the following situation in which two processes have access to shared data. Each process uses its own virtual address. Process 1 reads from address VA1 and the data is inserted in the cache at line L1. Process 2 then writes to address VA2 (which is a synonym to VA1), but line L1 is not affected because VA1 and VA2 do not index to the same cache line. When Process 1 reads VA1,
it accesses stale data from line L1 instead of the new data that's been written by Process 2.

There are several ways to solve this problem. One is to incorporate a hardware reverse translation buffer which, for every line in the cache, indicates all virtual addresses associated with a particular physical address. To avoid the use of this additional hardware, some systems don't allow the existence of synonyms at all [Cleg86, Wood86], or require that data be made non-encacheable whenever different user-level programs map to the same physical data [Chen87].

The solution for the DN4000 was an outgrowth of the way that objects are accessed in Apollo's system [Leac85]. In this system, a program accesses an object by "mapping" 32 Kbyte segments into the program's virtual address space (and subsequently accessing the mapped addresses to result in demand-paging). Because all the mappings are chosen by the operating system on a modulo 32 Kbyte boundary — and because the cache size is less than the segment size — all virtual address synonyms are guaranteed to index to the same cache line. This is exemplified in Figure 3. Segment A and Segment B are mappings of the same 32 Kbyte portion of a file object. Thus, virtual addresses in each mapping at the same offset from the beginning of the segment are synonyms (depicted in Figure 3 by VA1 and VA2). This implies that the low 15 bits of VA1 and VA2 are identical. Because only 13 bits are used to index into the 8 Kbyte virtual cache, VA1 and VA2 are guaranteed to index to the same cache line.

Another problem with virtual address synonyms exists if only a write-through without write-allocate policy is used. In this situation, Process 1 reads from address VA1, and the data is inserted in the cache at line L1. Process 2 then writes to address VA2 (which is a synonym to VA1 and indexes to line L1), but L1 is not affected because a write miss occurs as the result of a cache tag mismatch. When Process 1 reads VA1, it accesses stale data from line L1 instead of the new data that's been written by Process 2.

To solve this problem, a write miss must do one of two things: invalidate the indexed cache entry, or update the entry to contain the new data. So that subsequent accesses to the same data would result in cache hits, we used the latter approach in the DN4000.

Note that our examples used synonyms accessible through mappings in different processes. They equally as well could have used mappings within the same process. In fact, if the cache tags did not contain task alias identifiers (which would require a cache flush on every context switch), only single-process examples would be applicable.

4. Virtual Write Buffer

In a write-through cache architecture, physical memory is updated on all memory write operations. To improve performance, write buffers are often incorporated to capture memory writes and allow the processor to continue executing before the write has actually completed. Achieving this concurrency usually is accomplished by placing the write buffer on the physical or memory bus. This effectively "hides" the memory access time from the processor.

The DN4000's write buffer is a single longword deep and resides on the virtual side of the MMU (hence the name "Virtual Write Buffer"). This provides a greater degree of overlap by additionally hiding the MMU virtual-to-physical address translation time from the processor. Thus, a write that has been captured in the write buffer allows the processor to complete its cycle in zero wait states and continue execution out of the cache. Concurrently, the write buffer writes the data through the MMU to physical memory.

All writes that have an address tag match and "modified" bit set in the indexed cache entry are buffered. If the tag mismatches or the "modified" bit is not set, the write cannot be buffered because the processor must wait while the write access is validated. Once the write has successfully completed, the "modified" bit is set to indicate that subsequent writes may be buffered. (Note that the "modified" bit is similar to the PageDirty bit in the Berkeley SPUR architecture [Wood86] in that it indicates when cache data has previously been written.)

5. Operating System Considerations

In any cache architecture that uses virtual tags, a significant burden falls on the operating system to help maintain cache coherency. We have already described how the
problems associated with virtual address synonyms are solved by the write-allocate cache update policy and by the way that objects are accessed in Apollo's system. Other problems exist though. The operating system must also provide support for virtual-to-physical address mapping changes, context switching, modifying data without using virtual addresses (e.g., DMA), and processor modification of page table used bits.

5.1. Virtual-to-Physical Address Mapping Changes
Usually, when address mapping changes occur, the required action is to flush the cache. Consider a process that unmaps a portion of its virtual address space; those addresses are made invalid in the MMU page tables. However, because read accesses that hit in the cache do not involve MMU activity, a cache flush ensures that subsequent references to the unmapped addresses will cause the MMU to see the invalid mapping during address translation*. Or, if a new mapping of the same virtual address range has been established, the MMU will validate accesses to those addresses and allow the new data to be encached.

Address mapping changes can also occur when a physical page frame is reclaimed from a process, thus breaking the virtual-to-physical mapping. To determine if such reclaims should result in a cache flush, the following question must be asked: "Can valid data be allowed to exist in the cache when no page exists to back up that data in physical memory?"

For read accesses, the answer is yes; read accesses could use the information that has been encached even when no page exists to back up that data in physical memory. However, recall that when write accesses hit in the cache, it is the write buffer that must carry out the write to physical memory. If the page were not memory-resident, the write buffer would incur a fault. Because the system cannot recover from this situation, we disallow writable data from being in the cache but not in physical memory at the same time.

However, since the operating system doesn't distinguish between reclaims of writable and non-writable pages, all page reclaims result in a cache flush. But, because the operating system reclaims pages in groups (and only one cache flush is required per group), we consider the performance impact to be negligible.

5.2. Context Switches
Virtually tagged caches that do not have address space identifiers as part of their tags must be flushed on every context switch. This ensures that a new process won't hit on data inserted into the cache by the previous process.

* Address mapping changes also require flushing the MMU's translation lookaside buffer so that page table traversal will occur during MMU address translations.

The DN4000's cache includes a task alias identifier within each cache tag. This means that cache flushes are required only on those context switches where a task alias is being reused.

5.3. Modifying Data Without Using Virtual Addresses
Some modifications to physical memory (such as memory accesses performed by DMA controllers and those that occur during MMU page table updates) do not update the cache. The operating system helps to ensure that they won't result in cache hits on stale data.

For example, after a disk controller modifies data in physical memory via DMA, a process cannot be allowed to hit in the cache on the data that existed prior to the DMA. Our operating system ensures this in one of two ways (depending on the DMA device).

1. If the DMA device is used by the operating system for paging, then the pages involved must have been reclaimed previously (i.e., the virtual-to-physical address mappings to these pages were broken) and at that time, the cache was flushed. Since virtual addresses are not associated with pages until after DMA operations are complete, there is no danger that the cache contains stale data.

2. If the DMA device is accessed by a user-level device driver [Apol87b] then the operating system, to ensure cache coherence, has two choices.
   a. Flush the cache prior to every I/O operation. This guarantees that all data affected by the DMA is not cache-resident, and therefore, that it can't result in a hit on stale data.
   b. Make the buffers involved in the DMA non-encacheable at the time they are wired down (i.e., made non-pageable) by the operating system.

To avoid flushing the cache on each I/O operation we make user-level DMA buffers non-encacheable. When the buffers are unwired, indicating they no longer will be used in I/O operations, they are again made encacheable.

The MMU also uses physical addresses to modify data when it updates the state of page table used and modified bits during address translation. So that the operating system will always see the most up-to-date state of these bits, page tables are made non-encacheable.

5.4. Processor Updates to Page Table Used Bits
Physical memory accesses that go through the MMU set the used bit in the page table representing the page of data being accessed. The operating system examines the state of page table used bits in the implementation of its page replacement algorithm. If a used bit is clear then the corresponding page is a good candidate for replacement. If a used bit is set then the operating system clears
that whetstone benefits mainly from the increase in clock speed (25 MHz vs. 12 MHz) of the MC68881.

Table I. shows the results of our study.

All benchmarks are integer except for the floating point composite which has both integer and floating point components, and whetstone which is all floating point.

6. Performance

Our primary design goal of the DN4000 was to double the CPU performance of the existing DN3000*. To verify the DN4000's performance we ran several CPU-bound benchmarks on both the DN3000 and DN4000. In addition, to quantify the performance effects of the higher clock speed, virtual cache, and virtual write buffer, we ran the benchmarks on a DN4000 with the virtual cache and/or virtual write buffer disabled. Lastly, we studied the performance effect of cache flushing, and the impact of including task alias identifiers as part of the cache tags.

The benchmarks we ran are shown in Table 1. They provide a range of results from some commonly used small benchmarks as well as from some real applications. Results from puzzle, bubble, intmm, integer composite, and floating point composite were obtained from a benchmark developed at Stanford [Lint85], although many of the individual tests were developed at Berkeley and elsewhere. Dhrystone [Weic84] and whetstone [Curn76] are small synthetic benchmarks designed to simulate the behavior of real applications. The large compile and the espresso ECAD program from Berkeley represent some real applications, as do the programs run by Apollo's customers. All benchmarks are integer except for the floating point composite which has both integer and floating point components, and whetstone which is all floating point.

Table 1. shows the results of our study. All numbers are relative to the DN3000. The column entitled "DN4000 (cache only)" demonstrates the effectiveness of the 8 Kbyte virtual cache. Both the smaller programs and the large compile and espresso benefit significantly. Note that whetstone benefits mainly from the increase in clock speed (25 MHz vs. 12 MHz) of the MC68881.

The last column shows the effect of the virtual write buffer. It yields a performance increase of 5–15 percent, although 5 percent is the most likely improvement seen by real applications. Given that the write buffer constitutes less than 1 percent of the total CPU cost, we feel the performance improvement justifies its cost. A deeper write buffer would mitigate the negative effect of write stalls that occurs on back-to-back writes, but further investigation is needed to quantify the overall performance effect relative to its additional cost.

Feedback from several of Apollo's customers demonstrates that their applications run about 1.7 to 2.2 times faster on a DN4000. Nearly all are at least two times as fast, although we've encountered a couple that are less, reflecting cases where the application could benefit from a larger cache. Thus, overall we feel we've met our design goal of doubling the DN3000's CPU performance.

Table 1. Performance Comparison (relative to DN3000)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DN4000 (no cache or write buffer)</th>
<th>DN4000 (with cache only)</th>
<th>DN4000 (with both)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Puzzle</td>
<td>1.5x</td>
<td>2.0x</td>
<td>2.0x</td>
</tr>
<tr>
<td>Bubble</td>
<td>1.4x</td>
<td>1.8x</td>
<td>2.1x</td>
</tr>
<tr>
<td>Intmm</td>
<td>1.5x</td>
<td>2.1x</td>
<td>2.2x</td>
</tr>
<tr>
<td>Integer Composite</td>
<td>1.2x</td>
<td>1.8x</td>
<td>1.9x</td>
</tr>
<tr>
<td>Floating Point Composite</td>
<td>1.4x</td>
<td>2.0x</td>
<td>2.1x</td>
</tr>
<tr>
<td>Dhrystone</td>
<td>1.1x</td>
<td>1.9x</td>
<td>2.2x</td>
</tr>
<tr>
<td>Whetstone</td>
<td>1.9x</td>
<td>2.0x</td>
<td>2.1x</td>
</tr>
<tr>
<td>Large Compile</td>
<td>1.3x</td>
<td>1.9x</td>
<td>2.0x</td>
</tr>
<tr>
<td>Espresso</td>
<td>1.2x</td>
<td>2.0x</td>
<td>2.1x</td>
</tr>
<tr>
<td>Various Customer Applications</td>
<td>-</td>
<td>-</td>
<td>1.7–2.2x</td>
</tr>
</tbody>
</table>

Another useful performance measure in a virtually tagged architecture is the effect of cache flushing. Flushing serves to reduce the lifetime of cache data, thus potentially degrading the performance of applications by lowering their cache hit ratios. Unfortunately, acquiring quantitative cache hit ratio information over relatively long periods of operation is difficult to obtain.

We attempted to gain some insight by observing the impact of additional, unneeded cache flushes upon each entry into the operating system. Since most entries into the operating system do not result in a cache flush (implying that our additional cache flushes are indeed unneeded), a
performance degradation in this situation would indicate that cache flushing during normal operation could be degrading performance. However, our benchmarks results showed there to be no degradation due to the additional flushes.

One possible explanation is that each entry into the operating system effectively flushes the cache just by executing and accessing operating system code and data, meaning the actual cache flush has little effect. Another possibility is to conjecture that entries into the operating system most often happen when there is a transition in working set occurring. For example, a process unmapping a portion of its virtual address space, or page faulting (resulting in page reclamation), often precedes a change in the access behavior of the process. Thus, a cache flush at this time would have little effect on the cache hit ratio. We intend to investigate these possibilities further in future work.

Lastly, our experience with this architecture shows that inclusion of task alias identifiers in the cache tags has little or no performance benefit. We determined this by flushing the cache on every context switch during a multi-process benchmark and observing no performance degradation. This result is not surprising given the relatively small cache size, although future implementations with larger caches may experience a benefit.

7. Conclusion

In this paper we presented the virtual cache-based architecture of the Apollo DN4000 workstation. We described how our use of a virtually tagged, write-through cache made it possible to use a low-cost, off-the-shelf MMU and resulted in inclusion of a virtual write buffer. Virtual cache architectures, however, require solving the virtual address synonym problem. Our solution was an outgrowth of the way that objects are accessed in Apollo's system, and consisted of implementing a write-allocate cache update policy.

We also described the burden on the operating system to support the architecture. For example, because cache read hits do not involve MMU activity, and modifying data sometimes is done with physical addresses (e.g., DMA), the operating system must help maintain cache coherency.

Performance measurements were provided showing how our use of a virtually tagged cache and a virtual write buffer allowed us to attain our design goal of doubling the CPU performance of the existing Apollo DN3000.

Acknowledgements

The authors would like to thank Bryan Douros, Joe Gorfinkle, Daryl Kinney, Paul Leach, Rose O'Donnell, and Bernie Stumpf for their useful input and feedback relating to the virtual cache. We would also like to thank Hugh Lauer for his useful comments relating to the performance section, Glenn Crossman for his help with running benchmarks, and Marjorie Solomon for her contributions to the style and presentation of the paper.

References

[Apol87a] Apollo Computer, Inc. Domain Series 3000/Se-


