

1 Virtual Memory Performance

Paging introduces indirection into memory accesses. This is bad, because a 2 level page table turns one memory request into 3. As we've seen the TLB caches translations and therefore eliminates many of those accesses to memory.

1.1 TLB performance

TLBs, like any kind of cache, increase performance by caching the most important data in fast memory. Crucial to good performance is the **hit rate**, the hit rate measures the percentage of page translation queries that can be satisfied by the TLB (and thereby avoid accesses to main memory). In practice, TLBs should have hit rates on the order of 95-99%. To measure the performance of a TLB, one is concerned with the overall average memory access time rather than the access time for any one request. This is basically the expected time for a memory access and is calculated like so: $time_{avg} = time_{miss} * missrate + time_{hit} * hitrate$. Therefore, if access to main memory is 10ns, and a normal 3-level page lookup requires 4 accesses, $time_{miss} = 40ns$, and assuming that the TLB lookup requires 2ns, $time_{hit} = 12ns$. Therefore a TLB with a 95% hit rate will have an expected access time of $40ns * .05 + 12ns * .95 = 13.4ns$, which is less than the expected access time of a single-level page table with no TLB! Now consider a TLB with 99% hit rate (which is more realistic), then $time_{avg} = 40ns * .01 + 12 * .99 = 12.28ns$ which is roughly a 10% improvement!

1.2 Process involvement

A TLB caches mappings from virtual pages to physical pages. These mappings, therefore are process specific. What this means is that the TLB itself will have to be flushed (the entries purged) during a process context switch. Additionally, the TLB may have to be updated if the page table is modified. For example, if a page's protections change, then the TLB may have to be flushed to accomodate that. Additionally, if a page is swapped in or out (residence information changes) then the TLB may have to be updated/notified. On some systems, the MMU allows the OS fine-grained control over TLB entries which would permit the OS to flush a specific entry. On other systems, this can be accomplished in hardware. On other systems, the OS may have no option other than to flush the entire TLB.

2 Locality of Reference

Locality of reference (often abbreviated simply as locality) is an extremely important concept in systems. The basic idea is that programs don't just access memory willy-nilly; a program tends to obsess about a small amount of data for a while and then move on to another small chunk of data. There are two main kinds of locality:

- **Temporal Locality:** Temporal locality is the tendency for accesses to a particular datum to be clustered in time. e.g. if you used it a little while ago, you're probably going to use it again soon.
- **Spatial Locality:** Spatial locality is the tendency for related bits of data to be clustered near each other in memory. This is more often a side-effect of how aggregations of data are structured. For example, in an array it is common to access element n , then element $n+1$, then $n+2$, etc. Therefore, these elements should be located near each other in memory.

Ok, what does this mean? What locality is essentially getting at is the so-called 90/10 rule. Essentially roughly 90% of the work is done by 10% of the code. That 10% is often composed of loops that iterate through program data. Locality is a side-effect of iteration. Although the whole set of data may be large, on any given iteration the loop is only concerned with a small chunk.

Consider the following code that simply calculates the average of an array of ints:

```
int vals[N] = ...
int sum = 0;
...
int i;
for(i = 0 ; i < N ; i++){
    sum += vals[i];
}
int average = sum / N;
```

So, how much data is here? Well, `vals` is $4N$ bytes large, `sum`, `i`, `average`, and `N` are 4 bytes each. So, if N is large, there's a lot of data here, but how much of it is actually needed at any given time? Consider the `for` loop. On each iteration, the loop:

1. Compares `i` with `N`
2. Adds `vals[i]` to `sum`
3. Increments `i`

Each loop iteration reads/modifies `i`, `sum` and `N`. This is an example of temporal locality. Each iteration is tiny (half a dozen instructions) so each access to `i` or `N` is separated from the others by only a few instructions. Note that `average` is only touched after the loop and is therefore not worth caching during loop execution.

Now consider the loops behavior over `vals`. Each element in `vals` is only read once, but they are read in-order. There is a regular, exploitable pattern here. This is an example of spatial locality, where things accessed nearby in time are located nearby in memory (`vals[i]` is 4 bytes away from `vals[i+1]`).

Now consider the code itself. Remember code is also located in memory. This code is basically instructions executed one after another until the loop (spatial locality). The loop itself keeps executing the same instructions over and over again (temporal locality). In fact, code tends to exhibit extremely good locality.

2.1 Exploiting Locality

So, we've got this locality property. What do we do with it? One word, caching! TLBs work because only a small subset of all the program's data is important at any given time. TLBs can only contain a small number of entries because a TLB large enough to accommodate an entire page table would be prohibitively expensive. However, it doesn't matter. As long as the program's "hot" data fit on a few dozen pages (modern TLBs have on the order of 128 entries) then the TLB should be fine. So in operation a TLB will first start empty. Then it will be filled by the pages currently in use by the application. As the application moves from one hot set to the next, old translations will be evicted to make room for the newer, more relevant ones (using some variant of LRU, usually).

3 Thrashing

Of course, no memory manager (no matter how perfect) can prevent a system from slowing down when too many processes are running. A condition, known as thrashing, can occur when memory pressure reaches a point where every new page access causes a swap. This happens when the size of running process' **working sets** exceeds that of main memory.

Working sets: A working set is the set of pages that will be used by the process in the near future. It is a fuzzy term and can be highly dependant on the data inputs to the program. Basically, a working set corresponds to the set of "hot" pages for a program. If, for all ready/running processes, the size of their combined working sets exceeds main memory, the OS will have to swap pages after a context switch. This results in thrashing. Consider the following pathological case:

1. A process runs and triggers a page fault.
2. The process is blocked while the OS reads in the required page.
3. Other processes are allowed to run, and they steal the blocked process' pages
4. The faulted page is read in and the process resumes, but faults on a page evicted by the other processes

This leads to a situation where processes are blocked waiting for the disk to swap in needed pages. This over-utilizes the disk and under-utilizes the CPU. This is bad.

Solutions: The OS needs to have some metric to determine if thrashing is occurring. A convenient one is *page fault frequency* (pff). A drop in pff means that the process' working set is able to reside comfortably in memory. A rise in pff means that the process' working set may not be completely resident.

4 I/O and Virtual Memory

Swapped pages are stored on disk, which means that the CPU has to perform I/O operations occasionally to satisfy memory requests. No problem, you say, the system has DMA which means

that the CPU just has to start the I/O and the DMA controller will take care of the rest. That seems like a decent answer, until you remember that the DMA controller has no knowledge of virtual addresses. All the DMA controller knows is physical addresses, and because virtual addresses are process specific, it seems impossible to make the DMA controller aware of virtual memory. Additionally, virtual pages may not be contiguous in memory and the DMA controller may clobber contiguous pages if the transfer size is larger than the page size. How can we make DMA I/O compatible with paged virtual memory?

Solution 1: I/O buffering: One solution would be for the OS to reserve a chunk of physical memory specifically for DMA transfers. When the DMA is done, the OS steps in and copies the data over to the appropriate pages corresponding to the virtual addresses requested by the process. This solves the virtual/physical address problem, but now the OS must copy every byte from a device into main memory. This is what DMA was supposed to avoid!

Solution 2: Translation: With translation, the device drivers break up I/O transfers to respect page boundaries. For example, consider a 4K transfer from disk starting at offset 512 on a virtual page (with 4K page sizes). Because the DMA just dumps bytes into main memory, the transfer will have to be staged to write into the appropriate (not necessarily contiguous) physical pages. In this case, this means one 3.5K transfer into the physical page corresponding to the first page of the file, and one 512 byte transfer into the physical page corresponding to the second page. This is not particularly efficient for I/O devices, so most modern hard drives are equipped with a local cache. The disk reads all of the data into the cache, and then dispatches it in fragments to the DMA controller.

Additionally, the virtual memory manager can't re-allocate DMA pages until the transfer is complete. Otherwise bytes may be written to the wrong part of a process' address space, or to another process' address space altogether. This is solved by allowing the I/O system to "pin" pages. The memory manager cannot remap a pinned page (pinning is just another flag in the page table entry). However, now device drivers must be able to communicate with the memory manager, in order to pin and unpin pages. This is messy.