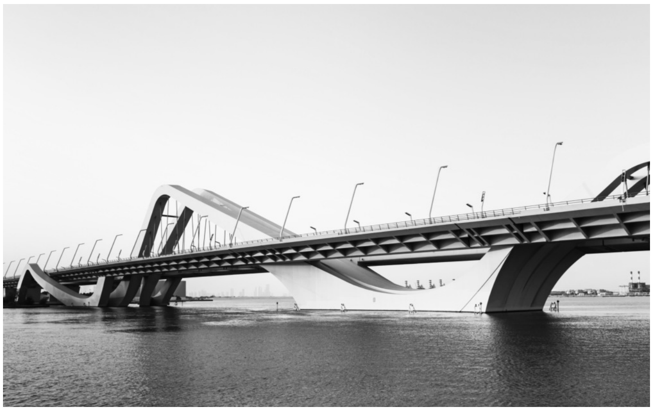
Marcel Weiher

iOS and macOS™ Performance Tuning

Cocoa®, Cocoa Touch®, Objective-C®, and Swift™



Opened in November 2010, the **Sheikh Zayed Bridge** offers a stunning entryway to the city of Abu Dhabi, United Arab Emirates. Designed by the visionary architect Zaha Hadid, the bridge appears to onlookers as a series of rising and falling concrete waves, reminiscent of the region’s nearby sand dunes. Hadid’s “waves,” reaching 64 meters at their peak, appear to propel themselves toward the city. Her design’s extraordinary energy is reinforced by dynamic nighttime lighting, making the bridge an unforgettable local landmark.

Apple][+ had access times of around 450 ns, which were state of the art for that time. With the processor running at 1 MHz and accessing at most 1 byte of memory per cycle, memory was actually twice as fast as required for the CPU, allowing Woz to use every other DRAM cycle to refresh the display without the CPU ever noticing. By the time of the first Mac, with the CPU running at 7 MHz and DRAM access times at 250 ns, the addition of video refresh was already having a slight impact on the CPU's access to memory, reducing the effective speed to 6 MHz, despite the puny 512×342 monochrome display. The contemporary Amiga could actually almost completely starve the CPU of memory access cycles due to its higher-resolution color graphics and additional co-processors, prompting the designers to partition its physical memory into *chip memory* that could be accessed by the video logic and *fast memory* that was exclusive the CPU.

Whereas modern CPUs have gotten over 1,000 times faster since the Apple][+, memory access times have improved only around a factor of 10, to somewhere between 25 and 45 ns. The complex arrangement of caches is designed to hide the difference in latency. At the same time main memory interfaces have been redesigned to provide more data with each access, dramatically increasing the bandwidth while further hurting latencies, all to try and keep the CPU supplied with data for typical programs. Fortunately, all of this complexity is pretty much transparent, mostly even inaccessible to the programmer, who only sees normal memory access instructions. So why bother looking at the cache hierarchy and memory interface?

The data in Figure 5.2 provides an answer to that question: Different memory access patterns have different performance characteristics, varying by a factor of up to 100! The data was obtained by running Example 5.2. This program is designed to create access patterns that are either sequential or random, and either stay within the bounds of a specific cache or require main memory access.

Example 5.2 Test memory access

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MBSIZE 16
#define SIZE (MBSIZE  * 1024 * 1024)

#define UNROLL 4
#define COUNT    ( 1000 * 1000)

int main(int argc, char *argv[] )
{
  if (argc > 2) {
    long stride=atol(argv[2]);
    char *ptr=malloc( SIZE + 20 * stride  );
    memset( ptr, 55, SIZE + 10 * stride );
    char *cur=ptr;
    long curCount=atol(argv[1]) * COUNT/UNROLL;
    long result=0;
    long headroom=UNROLL * stride;
    while ( curCount-- > 0 ) {
      result+=*cur; cur+=stride;
      result+=*cur; cur+=stride;
      result+=*cur; cur+=stride;
      result+=*cur; cur+=stride;
      if ( ((cur-ptr)+headroom) > SIZE ) {
        cur-=(SIZE-headroom);
      }
    }
    printf("result: %ld\n",result);
  } else {
    printf("usage: %s <access-count-in-millions> <stride>\n",argv[0]);
  }
}
```

The inset for Figure 5.2 clearly shows the speed differences between L1, L2, and L3 cache memory. It also shows that this speed difference does not matter for sequential accesses with strides significantly smaller than the cache line size of 64 bytes. Once the stride exceeds the cache line size, though, the access pattern no longer matters. The penalty for random access is virtually nonexistent for accesses within L1, rises to a factor 2 for L2, and to slightly over factor 4 for L3.

The main graph in Figure 5.2 adds the data for main memory access, using the 1-GB buffer size to force the accesses out of the caches. As you can see from the rightmost bar, random access to main memory takes about 38 ns, roughly 100 times the amount to access data in the L1 cache. In fact, the data for the cached accesses, which is repeated in Figure 5.2, has to be scaled down to be almost unreadable in order to accommodate the bar for random main memory access on the page.

However, sequential access (strides 1–2) suffers virtually no slowdown, even when having to plow through a gigabyte of main memory, showing the tremendous bandwidth available with DDR3-1333 memory, but also highlighting the ever increasing gap between latency and bandwidth.

Whereas caches reduce memory access times by a factor of 100 for a small subset of main memory, virtual memory expands addressable memory beyond what is physically installed, but at a potentially huge performance cost. Just how large a difference is shown by the somewhat ridiculous graph in Figure 5.3, which tries to compare the random access time of main memory (36 ns) with the random access time of the fastest available solid-state disks (SSDs) of around 10 μs. What was by far the longest bar in Figure 5.2 now becomes a barely visible sliver, whereas the SSD access time takes the entire length of the page. The fastest spinning disks, at more than 100 times slower still, won't fit and the times for the CPU caches don't rise visibly above the x-axis.

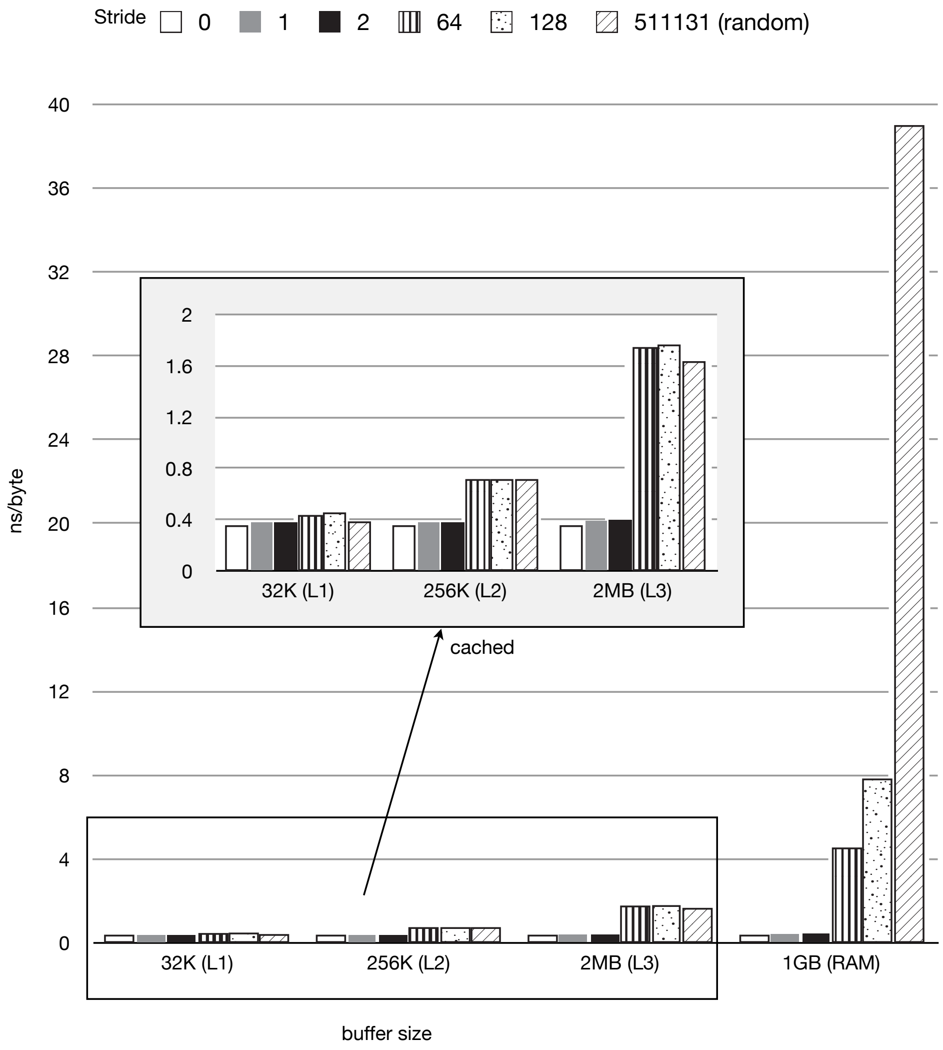


Figure 5.2 Memory access

Showing the different parts of the memory hierarchy in a single graph requires using either a log scale (Figure 5.4) or a different book format. The consequences of these numbers should be clear: It really, really pays to stay on the "good" side of the memory hierarchy, and conversely, the performance penalties for getting on the bad side are severe. So if you have essentially random access patterns (as you do with individually allocated objects), you should try very hard to keep your *working set*, the

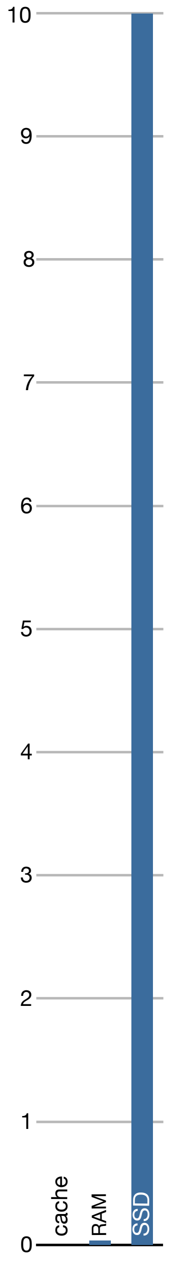


Figure 5.3 SSD vs. RAM access times

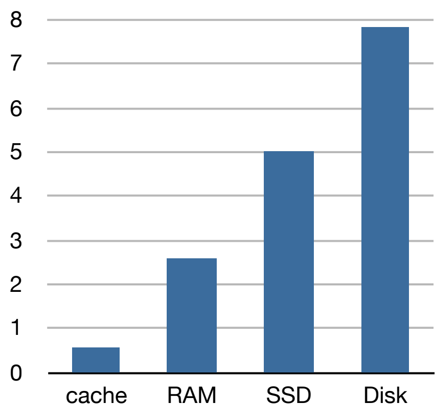


Figure 5.4 Memory hierarchy access times, log scale

set of information your program is working with/on in a given time interval, within the caches, preferably L1 or L2. If you have data sets that do not fit within the caches, make sure to access them sequentially—that way you get to take advantage of memory's high bandwidth rather than being stuck with its high latency.

Mach Virtual Memory

Although the memory hierarchy contains hardware of vastly different types and characteristics, the operating system makes it all appear uniform to the userland developer. The Mach microkernel that OS X is based on does this by separating the concepts of *address space* and *memory*, with the user interacting primarily with address space and the operating system and hardware cooperating to back that address space with different kinds of resources.

Irrespective of the physical memory organization discussed in the previous section, the address space of processes on most modern operating systems is organized into fixed-size *pages*, currently 4,096 bytes on iOS and OS X. You can determine the size yourself using the program in Example 5.3. When your program tries to access memory, it uses a virtual address. This address is translated by the CPU to a physical address by shifting it right by 12 bits to get a page number. That page number is translated to a physical page using page tables maintained by the operating system (with a small cache inside the CPU, the *translation lookaside buffer*, one of the busiest pieces of hardware on a CPU) and the low 12 bits are then used to access memory within the page.

Example 5.3 Determine VM page size

```
#include <stdio.h>
#include <mach/vm_page_size.h>

int main() {
 printf("page-size: %ld mask: %lx shift: %d\n",vm_kernel_page_size,
          vm_kernel_page_mask,vm_kernel_page_shift);
 return 0;
}
```

The basic process outlined assumes that real, physical memory is used to back the virtual address being translated. This is not necessarily the case, and in fact address space provided to a process tends to start out not being backed by memory, but rather either *zero filled* or backed by the contents of a disk file.

In either case, the operating system will provide real memory when and if the memory is actually accessed, either zeroed or filled with the data from the disk file.

Whether mapped from disk or allocated from the OS, memory starts out in a *clean* state, meaning it hasn't been written to. Once your process writes to a memory location, that page of virtual memory gets marked as *dirty* by the virtual memory subsystem, meaning it differs from whatever backing store it has.

The reason this distinction is important is that dirty pages are significantly more expensive than clean pages, at least once memory becomes tight, and since the OS is tuned to utilize memory as much as possible, memory essentially always becomes tight, even if you have plenty available.

The OS tries to keep a minimum number of free pages available for allocations, so once free memory drops below a certain threshold, it will start looking for pages in memory that it can evict. Clean pages are easy to evict because all that needs to be done is to change their mapping to point back to the file on disk and the page added to the pool of free memory maintained by the kernel. I/O is only incurred when and if the page is needed again, and then only a read is required. Dirty pages on the other hand must first have their contents written to disk, and they cannot be reused until that I/O has finished.

iOS does not swap to disk, so dirty pages are even more expensive on iDevices. No matter how rarely used, a dirty page can never be written to disk, so the iOS has to even more aggressively swap clean pages (executable code, mapped files) or terminate the process.

Heap and Stack

"Heap" and "stack" traditionally refer to two distinct regions of dynamically allocated memory. Figure 5.5 shows this traditional arrangement: the static regions of a program, actual program code, initialized global/static data, and uninitialized global/static data are at the bottom of the address space. The rest of the address space