DIOMIDIS SPINELLIS

some types of memory are more equal than others

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business, a FreeBSD committer, and a four-times winner of the International Obfuscated C Code Contest.

dds@aueb.gr

Parts of this article are excerpted from Diomidis Spinellis's *Code Quality: The Open Source Perspective*, Addison Wesley, 2006. The last section was inspired by the book's Exercise 5–8.

IF WE WANT TO MAKE INTELLIGENT decisions regarding the performance of our systems, we must understand how the various types of memory we find in them work together to provide us with the illusion of a huge and blindingly fast memory store. For example, a program's space requirements often affect its execution speed. This happens because a computer system's memory is a complex amalgam of various memory technologies, each with different cost, size, and performance characteristics. Making our program's working set small enough to get a front seat on a processor's level 1 cache may provide us with a very noticeable boost in its execution speed. Along the same lines, in today's networked computing environments and distributed applications, lower size requirements translate into lower bandwidth requirements and, therefore, swifter program loading and operation. Finally, a program's service capacity is often constrained by the space requirements of its data set.

Due to a number of engineering decisions involving complicated tradeoffs, modern computers sport numerous different memory systems layered on top of each other. (As a general rule, whenever you see "complicated tradeoffs," read "cost.") At any time, our data will be stored in one (or more) of these many layers, and the way a program's code is organized may take advantage of the storage system's organization or be penalized by it. Some of the layers we will talk about are related to caching. In this article we describe them from the viewpoint of storage organization.

Let us summarize how data storage is organized on a modern computer. Figure 1, below, illustrates the hierarchy formed by different storage technologies. Elements near the top represent scarce resources: fast but expensive. As we move toward the bottom the elements represent abundant resources: cheap but slow. The fastest way to have a processor process a data element is for the element to be in a register (or an instruction). The register is encoded as part of the CPU instruction and is immediately available to it. However, this advantage means that processors offer only a

↓ Increasing size	CPU registers
	Level 1 cache (on chip)
	Level 2 cache
	Level 3 cache (off chip)
	Main memory
	Disk cache and banked memory
	Paged out memory
	File-based disk storage
↑ Increasing speed and cost	Off line storage

FIGURE 1. A MODERN COMPUTER'S STORAGE HIERARCHY

small fixed number of registers (eight, for example, on the ia-32; 128 on Sun's SPARC architecture.) See how a data processing instruction (such as add) is encoded on the arm architecture:

31 28	27 26	25	24 21	20	19 16	15 12	11 0
Cond	00	I	Opcode	S	Rn	Rd	Operand 2

Rn is the source register and Rd the destination. Each register is encoded using four bits, limiting the number of registers that can be represented on this architecture to 16. Registers are used for storing local variables, temporary values, function arguments, and return values. Nowadays, they are allocated to their various uses by the compiler, which uses extremely sophisticated algorithms for optimizing performance at a local and global level. In older programs you may find this allocation specified by the programmers, based on their intuition of which values should be placed in a register; here is a typical example from the Korn shell source code:

```
struct tbl *
global(n)
        register const char *n;
{
        register struct block *l = e->loc;
         register struct tbl *vp;
        register int c;
        unsigned h;
        bool_t array;
        int val;
```

This strategy might have been beneficial when compilers had to fit in 64KB of memory and could not afford to do anything clever with register allocation; modern compilers simply ignore the register keyword.

Main Memory and Its Caches

The next four layers of our hierarchy (from the level 1 cache up to the main memory) involve the specification of data through a memory address. This (typically 16, 32, or 64-bit) address is often encoded on a word separate from the instruction (it can also be specified through a register) and thus may involve an additional instruction fetch. Worse, it involves interfacing with dynamic RAMs, the storage technology used for a computer's main memory, which is simply not keeping pace with the speed increases of modern processors. Fetching an instruction or data element from main memory can have the processor wait for a time equivalent to that of the execution of hundreds of instructions. To minimize this penalty, modern processors include facilities for storing temporary copies of frequently used data on faster, more versatile, more easily accessible, and, of course, more expensive memory: a *cache*. For a number of reasons a memory cache is

typically organized as a set of blocks (typically 8–128 bytes long) containing the contents of consecutive memory addresses. Keep this fact in mind; we'll come back to it later on.

The *level 1 cache* is typically part of the processor's die. It is often split into an area used for storing instructions and one used for storing data, because the two have different access patterns. To minimize the cache's impact on the die size (and therefore on the processor's production yield[1] and its cost), this cache is kept relatively small. For example, the Sun Micro SPARC I featured a 4KB instruction and a 2KB data cache; moving upward, the Intel 3.2GHz Pentium 4 processor features a 1MB cache.

1. A larger processor die means there is a higher chance for an impurity to result in a malfunctioning chip, thus lowering the production's yield.

Because of the inherent size limitations of the on-chip cache, a *level 2 cache* is sometimes implemented through a separate memory chip and control logic, either packaged with the processor or located near the processor. This can be a lot larger: it used to be 64KB on early 486 PC motherboards; an Intel 3.2GHz Xeon processor comes with 2MB. Finally, computer manufacturers are increasingly introducing in their designs a *level 3 cache*, which either involves different speed versus cost tradeoffs or is used for keeping a coherent copy of data in multiprocessor designs.

How do these levels of the memory hierarchy relate to our code and its properties? By reducing a program's memory consumption and increasing its locality of reference, we can often speed up its performance. All of us have witnessed the pathological case where increased memory consumption coupled with a lack of locality of reference leads to a dramatic performance drop due to thrashing. In the following paragraphs we will examine the winning side of the coin, where appropriate design and implementation decisions can lead to time performance increases.

Memory savings can translate into speed increases when the corresponding data set is made to fit into a more efficient part of a memory hierarchy. In an ideal world, all of our computer's memory would consist of the high-speed memory chips used in its cache. (This ideal world actually exists, and it is called a government-funded supercomputer.) We can, however, also pretend to live in the ideal world, by being frugal in the amount of memory our application requires. If that amount is small enough to fit into the level 2 (or, even better, the level 1) cache, then we will notice an (often dramatic) speed increase. Here is an actual code comment detailing this fact:

```
// Be aware that time will be affected by the buffer fitting/not
// fitting in the cache (ie, if default_total*sizeof(T) bytes
// fit in the cache).
```

Cases where the effort of fitting an application into a cache can be a worthwhile exercise typically involve tight, performance-critical, code. For example, a JVM implementation that could fit in its entirety into a processor's level 1 instruction cache would enjoy substantial performance benefits over one that couldn't.

There are, however, many cases where our program's data or instructions could never fit the processor's cache. In such cases, improving a program's locality of reference can result in speed increases, as data elements are more likely to be found in a cache. Improved locality of reference can occur both at the microscopic level (e.g., two structure elements being only 8 bytes apart) and at the macroscopic level (e.g., the entire working set for a calculation fitting in a 256KB level 1 cache). Both can increase a program's speed, but for different reasons.

Related data elements that are very close together in memory have an increased chance of appearing together in a cache block, one of them causing the other to be *prefetched*. Earlier on, we mentioned that caches organize their elements in blocks associated with consecutive memory addresses. This organization can result in increased memory access efficiency, as the second related element is fetched from the slow main memory as a side effect of filling the corresponding cache block. For this reason some style guides (such as the following excerpt from the FreeBSD documentation) recommend placing structure members together ordered by use.

```
* When declaring variables in structures, declare them sorted
* by use, then by size, and then by alphabetical order. The
* first category normally doesn't apply, but there are
* exceptions. Each one gets its own line.
```

(The exceptions referred to above are probably performance-critical sections of code, sensitive to the phenomenon we described.)

In other cases, a calculation may use a small percentage of a program's data. When that working set is concentrated in a way that allows it all to fit into a cache at the same time, the calculations will all run at the speed of the cache and not at that of the much slower main memory. Here is a comment from the NetBSD TCP processing code describing the rationale behind a design to improve the data's locality of reference:

```
* (2) Allocate syn_cache structures in pages (or some other
* large chunk).  This would probably be desirable for
* maintaining locality of reference anyway.
```

Locality of reference can also be important for code; here is another related comment from the X Window System VGA server code:

```
* Reordered code for register starved CPU's (Intel x86) plus
* it achieves better locality of code for other processors.
```

Disk Cache and Banked Memory

Moving down our memory hierarchy, before reaching the disk-based file storage we encounter two strange beasts: the disk cache and banked memory. The disk cache is a classic case of space over time optimization, and the banked memory is . . . embarrassing. Accessing data stored in either of the two involves approximately the same processing overhead, and for this reason they appear together in our table. Nevertheless, their purpose and operation are completely different, so we'll examine each one in turn.

The *disk cache* is an area of the main memory reserved for storing temporary copies of disk contents. Accessing data on disk-based storage is at least an order of magnitude slower than accessing main memory. Note that this figure represents a best (and relatively rare) case: sustained serial I/O to or from a disk device. Any random-access operation involving a head seek and a disk rotation is a lot slower; a six-orders-of-magnitude difference between disk and memory access time (12ms over 2ns) should not surprise you. To overcome this burden, an operating system aggressively keeps copies of the disk contents in an area of the main memory it reserves for this purpose. Any subsequent read or write operations involving the same contents (remember the locality-of-reference principle) can then be satisfied by reading or writing the corresponding memory blocks. Of course, the main memory differs from the disk in that its contents get lost when power is lost; therefore, periodically (e.g., every 30 seconds on some UNIX systems) the cache contents are written to disk.

Furthermore, for some types of data (such as elements of a database transaction log, or a file system's directory contents—the so-called directory metadata) the 30-second flush interval can be unacceptably high; such data is often scheduled to be written to disk in a *synchronous* manner or through a time-ordered *journal*. Keep in mind here that some file systems, either by default (the Linux *ext2fs*) or through an option (the FreeBSD FFS with soft updates enabled), will write directory metadata to disk in an asynchronous manner. This affects what will happen when the system powers down in an anomalous fashion, due to a power failure or a crash. In some implementations, after a reboot the file system's state may not be consistent with the order of the operations that were performed on it before the crash.

Nevertheless, the performance impact of the disk cache is big enough to make a difference between a usable system and one that almost grinds to a halt. For this reason, many modern operating systems will use all their free memory as a disk cache.

As we mentioned, banked memory is an embarrassment; we would not be discussing it at all but for the fact that the same embarrassment keeps recurring (in different forms) every couple of years. Recall that with a variable N bits wide we can address 2^N different elements. Consider the task of estimating the number of elements we might need to address (the size of our address space) over the lifetime of our processor's architecture. If we allocate more bits to a variable (say, a machine's address register) than those we would need to address our data, we end up wasting valuable resources. On the other hand, if we underestimate the number of elements we might need to address, we will find ourselves in a tight corner.

Intel architecture	*Address bits*	*Addressing limit*	*Stopgap measure*
8080	16	64KB	IA-16 segment registers
IA-16	20	1MB	XMS (Extended Memory Specification); LIM EMS (Lotus/Intel/Microsoft Expanded Memory Specification)
IA-32	32	4GB	PAE (Physical Address Extensions); AWE (Address Windowing Extensions)

TABLE 1. SUCCESSIVE ADDRESS SPACE LIMITATIONS AND THEIR INTERIM SOLUTIONS

In Table 1 you can see three generations of address space limitations encountered within the domain of Intel architectures, and a description of the corresponding solutions. Note that the table refers only to an architecture's address space; we could draw similar tables for other variables, such as those used for addressing physical bytes, bytes in a file, bytes on a disk, and machines on the Internet. The technologies associated with the table's first two rows are fortunately no longer relevant. One would think that we would have known by now to avoid repeating those mistakes, but this is, sadly, untrue.

As of this writing, some programs and applications are facing the 4GB limit of the 32-bit address space. There are systems, such as database servers and busy Web application servers, that can benefit from having at their disposal more than 4GB of physical memory. New members of the IA-32 architecture have hardware that can address more than 4GB of physical

memory. This feature comes under the name Physical Address Extensions (PAE). Nowadays we don't need segment registers or BIOS calls to extend the accessible memory range, because the processor's paging hardware already contains a physical-to-virtual address translation feature. All that is needed is for the address translation tables to be extended to address more than 4GB. Nevertheless, this processor feature still does not mean that an application can transparently access more than 4GB of memory. At best, the operating system can allocate *different* applications in a *physical* memory area larger than 4GB by appropriately manipulating their corresponding virtual memory translation tables. Also, the operating system can provide an API so that an application can request different parts of the physical memory to be mapped into its virtual memory space—again, a stopgap measure, which involves the overhead of operating system calls. An example of such an API is the Address Windowing Extensions (AWE) available on the Microsoft Windows system.

Swap Area and File-Based Disk Storage

The next level down in our memory storage hierarchy moves us away from the relatively fast main memory into the domain governed by the (in comparison) abysmally slow and clunky mechanical elements of electromagnetic storage devices (hard disks). The first element we encounter here is the operating system's *swap area* containing the memory pages it has temporarily stored on the disk, in order to free the main memory for more pressing needs. Also here might be pages of code that have not yet been executed and will be paged in on demand. At the same level in terms of performance, but more complicated to access in terms of the API, is the file-based disk storage. Both areas have typically orders-of-magnitude larger capacity than the system's main memory. Keep in mind, however, that on many operating systems the amount of available swap space or the amount of heap space a process can allocate is fixed by the system administrator and cannot grow above the specified limit without manual administrative intervention. On many UNIX systems the available swap space is determined by the size of the device or file specified in the swapon call and the corresponding command; on Windows systems, the administrator can place a hard limit on the maximum size of the paging file. It is therefore unwise not to check the return value of a malloc memory allocation call against the possibility of memory exhaustion. The code in the following code excerpt could well crash when run on a system low on memory:

```
TMPOUTNAME = (char *) malloc (tmpname_len);
strcpy (TMPOUTNAME, tmpdir);
```

The importance of the file-based disk storage in relationship to a program's space performance is that disk space tends to be a lot larger than a system's main memory. Therefore, *uncaching* (Bentley's term) is a strategy that can save main memory by storing data into secondary storage. If the data is persistent and rarely used, or does not exhibit a significant locality of reference in the program's operation, then the program's speed may not be affected; in some cases by removing the caching overhead it may even be improved. In other cases, when main memory gets tight, this approach may be the only affordable one. As an example, the UNIX sort implementations will only sort a certain amount of data in-core. When the file to be sorted exceeds that amount, the program will work by splitting its work into parts sized according to the maximum amount it can sort. It will sort each part in memory and write the result to a temporary disk file. Finally, it will *merge sort* the temporary files, producing the end result. As another

example, the *nvi* editor will use a backing file to store the data corresponding to the edited file. This makes it possible to edit arbitrarily large files, limited only by the size of the available temporary disk space.

The Lineup

Component	*Nominal size*	*Worst case latency*	*Sustained throughput (MB/s)*	*$1 buys*	*Productivity (Bytes read / s / $)*	
					Worst case	*Best case*
L1 D cache	64KB	1.4ns	19022	10.7KB	$7.91 \cdot 10^{12}$	$2.19 \cdot 10^{14}$
L2 cache	512KB	9.7ns	5519	12.8KB	$1.35 \cdot 10^{12}$	$7.61 \cdot 10^{13}$
DDR RAM	256MB	28.5ns	2541	9.48MB	$3.48 \cdot 10^{14}$	$2.65 \cdot 10^{16}$
Hard drive	250GB	25.6ms	67	2.91GB	$1.22 \cdot 10^{11}$	$2.17 \cdot 10^{17}$

TABLE 2. PERFORMANCE AND COST OF VARIOUS MEMORY TYPES

(Author pauses to don his flame retardant suit.) To give you a feeling of how different memory types compare in practice, I've calculated some numbers for a fairly typical configuration, based on some currently best-selling middle-range components: an AMD Athlon XP 3000+ processor, a 256MB PC2700 DDR memory module, and a 250GB 7200 RPM Maxtor hard drive. The results appear in Table 2. I obtained the component prices from TigerDirect.com on January 19, 2006. I calculated the cost of the cache memory by multiplying the processor's price by the die area occupied by the corresponding cache divided by the total size of the processor die (I measured the sizes on a die photograph). The worst-case latency column lists the time it would take to fetch a byte under the worst possible scenario: for example, a single byte from the same bank and following a write for the DDR RAM, with a maximum seek, rotational latency, and controller overhead for the hard drive. On the other hand, the sustained throughout column lists numbers where the devices operate close to ideal conditions for pumping out bytes as fast as possible: eight bytes delivered at double the bus speed for the DDR RAM; the maximum sustained outer diameter data rate for the hard drive. In all cases, the ratio between bandwidth implied by the worst-case latency and the sustained bandwidth is at least one order of magnitude, and it is this difference that allows our machines to deliver the performance we expect. In particular, the ratio is 27 for the level 1 cache, 56 for the level 2 cache, 76 for the DDR RAM, and 1.8 million for the hard drive. Note that as we move away from the processor there are more tricks we can play to increase the bandwidth, and we can get away with more factors that increase the latency.

The byte cost for each different kind of memory varies by three orders of magnitude: with one dollar we can buy KBs of cache memory, MBs of DDR RAM, and GBs of disk space. However, as one would expect, cheaper memory has a higher latency and a lower throughput. Things get more interesting when we examine the productivity of various memory types. Productivity is typically measured as output per unit of input; in our case, I calculated it as read operations per second and $ cost for one byte. As you can see, if we look at the best-case scenarios (the device operating at its maximum bandwidth), the hard drive's bytes are the most productive. In the worst case (latency-based) scenarios the productivity performance of the disk is abysmal, and this is why disks are nowadays furnished with abundant amounts of cache memory (8MB in our case). The most productive device in the worst-case latency-based measurements is the DDR RAM. These results are what we would expect from an engineering point of view: the hard disk, which is a workhorse used for storing large amounts of data with the minimum cost, should offer the best overall productivity under

ideal (best-case) conditions, while the DDR RAM, which is used for satisfying a system's general-purpose storage requirements, should offer the best overall productivity even under worst-case conditions. Also note the low productivity of the level 1 and level 2 caches. This factor easily explains why processor caches are relatively small: they work admirably, but they are expensive for the work they do.

What can we, as programmers and system administrators, learn from these numbers? Modeling the memory performance of modern systems is anything but trivial. As a programmer, try to keep the amount of memory you use low and increase the locality of reference so as to take advantage of the available caches and bandwidth-enhancing mechanisms. As a system administrator, try to understand your users' memory requirements in terms of the hierarchy we saw before making purchasing decisions; depending on workload, you may want to trade processor speed for memory capacity or bandwidth, or the opposite. Finally, always measure carefully before you think about optimizing. And next time you send a program whizzing through your computer's memory devices, spare a second to marvel at the sophisticated technical and economic ecosystem these devices form.