Memory models for embedded multicore SoCs, Part 2 - Cache and Overlays

Gitu Jain, Synopsys - August 27, 2013

Editor's Note: Multicore architectures find use across a diverse range of applications thanks to their performance and efficiency. By combining several general-purpose MCU cores -- or MCU cores and specialized cores such as DSPs -- IC manufacturers can deliver devices well tuned to specific application requirements. Real World Multicore Embedded Systems brings together specialists offering the latest thinking on each facet of a multicore architecture. This excerpt offers an in-depth review and discussion on the options available to designers for creating a memory architecture best suited to their particular embedded system application. Part 1 offered an overview of multicore SoC memory architectures. This installment describes strategies for cache, virtual memory and overlays.

Adapted from "Real World Multicore Embedded Systems, 1st Edition", B Moyer, Editor (Newnes)

Data cache
The data cache hierarchy in typical processors is composed of several levels. The L1, or primary cache, is a small, high-speed cache residing in the CPU core (on-chip cache). It typically ranges in size from 8 KB to 64 KB. The L1 cache is normally split into two separate caches, one for instructions and the other for data.

The L2, or secondary cache, is bigger, slower, and less expensive than the L1 cache and, unlike the L1 cache, it usually resides outside the CPU core (off-chip cache). It typically ranges in size from 64 KB to 4 MB. Many processors may also include a third larger, slower and less expensive cache at the L3 level just before the main memory. The L2 and L3 caches may be shared between the processor cores.

If the data requested by the cache client (processor or core) is already contained in the cache, a so-called cache hit, then this request can be served by simply reading the cache, which is comparatively faster. Otherwise, upon a cache miss, the data has to be recomputed or fetched from its original storage location, which is slower. The percentage of accesses that result in cache hits is known as the hit rate or hit ratio of the cache. The hit rate has a direct effect upon the overall system performance.

Multi-level caches generally operate by checking the L1 cache first; if it hits, the processor proceeds at high speed. If the smaller cache misses, the next larger L2 cache is checked, and so on, before external memory is accessed. The cache controller, discussed later, handles the details of retrieving
the required data from the closest level of cache (or main memory) that contains it.

There are two basic approaches to modifying data stored in a cache: write-through, where the data is updated in both the cache and the main memory simultaneously; and write-back, where only the copy of the data in the cache is updated. A modified cache block is written back to the main memory just before it is replaced by another. Write-back cache is more complex to implement, since it needs to keep track of and mark “dirty” any modified memory locations so that they can be written into main memory at some later time, when they are evicted from the cache.

A read miss on the dirty data by another cache of a multicore system may require two memory accesses to service: one for the cache holding the modified data to write it back to the main memory, and then one to retrieve the needed data into the current cache. This process of keeping the caches of different cores consistent is referred to as coherency, and will be discussed in section 5.

A cache controller is a hardware block that can dynamically move code and data from main memory to the cache memory and back. The incoming data or code replaces old code or data which is currently not being used in the cache memory. The replacement policy of a cache is the heuristic responsible for ejecting a cache line to make room for an incoming cache line, based upon the address of the cache line in main memory.

The replacement policy is called fully associative if the cache line can be placed anywhere in the cache. On the other end, if the cache line can only be placed in one place in the cache, the cache is direct mapped. Most replacement policies are somewhere in between, called N-way set associative, where an entry in main memory can go to any one of N places in the cache. The higher the associativity, the more chance there is of getting a cache hit as there are more cache locations in which a particular cache line can reside. This does, however, come with a cost: it takes more time and power to search for a cache line. Most caches implement a 2- or 4-way associativity as increasing the associativity beyond this is shown to have less effect on cache hit rate. See Figure 4.2.

![Figure 4.2. Cache associativity.](image)

Cache lines

The memory of a cache is divided into cache lines. This is the size of data that is read from or written into memory at one time. Depending upon the size of the cache, there are N number of lines that can be stored in the cache at any given time, where N = cache size/line size.

A memory address is split into a tag, index, and a block offset, as shown in Figure 4.3. The index is
used to determine in which cache line the data has been placed. The block offset specifies the position of the data in that particular cache line. The tag contains the most significant bits of the address and is stored in the cache line, as shown in Figure 4.3, and enables the cache to translate the cache address to a unique memory address. The data section contains the actual data. The flag bits are used by the cache coherence protocols and will be discussed in section 5.

For example, consider a system with 256 MB of memory and a direct-mapped L1 data cache of size 8 KB with 64-byte cache lines. There are 8 K/64 = 128 cache lines. The memory address has to be \( \log_2(256 \text{ MB}) = 28 \) bits long to access any location in memory. Since the cache line is 64 bytes the offset is \( \log_2(64) = 6 \) bits, and will point to the exact location of the memory being accessed on the cache line. The index into the cache in this case is \( \log_2(128) = 7 \) bits long. So the tag will have \((28 - 7 - 6) = 15\) bits. The tag is stored along with the cache line data. When a memory location is accessed, the cache controller checks to see if the cache already contains the data by using the memory index to access the appropriate cache line and comparing the tag value(s).

For 4-way associative L1 data cache, there will be \( 128/4 = 32 \) sets for the lines. This gives \( \log_2(32) = 5 \) different indices. Each memory access now needs to be checked against 4 different cache lines for a match. The memory address for a 4-way associative cache will be:

\[
\begin{array}{ccc}
15 & 7 & 6
\end{array}
\]

**Cache customization**

While all the cache blocks in a particular cache are all the same size and have the same associativity, typically “lower-level” caches, such as the L1 cache, are smaller in size and have smaller cache lines, while “higher-level” caches, such as the L2 cache, are larger in size and have larger cache lines.

When deciding upon a cache’s total size, consider the fact that a small cache is more energy efficient and has a good hit rate for a majority of applications, but a larger cache increases the range of applications displaying a good hit rate, at the expense of wasted energy for many applications.

An embedded system typically executes just a small set of applications for the system’s lifetime, in contrast to general-purpose desktops, so, ideally, we would like to tune the architecture to those applications. Two of the most important aspects for customizing a cache in an embedded system are the cache line size and the cache size.

**Rule of thumb**

As a rule of thumb, if the memory access pattern exhibits high spatial locality, i.e. is very regular and
consecutive, a longer cache line should be used since it minimizes the number of off-chip accesses and exploits the locality by pre-fetching elements that will be needed in the immediate future. On the other hand, if the memory access pattern is irregular, a shorter cache line is desirable as this reduces off-chip memory traffic by not bringing unnecessary data into the cache. The maximum size of a cache line is the main memory page size. Estimation techniques using data re-use analysis can be used to predict the number of cache hits and misses for different cache and cache line size. Based upon the results, the best size should be selected for the cache [1].

Designers of real-time embedded systems sometimes disable the use of cache as it can lead to unpredictable access times depending upon whether the data/instruction access resulted in a cache hit or not.

An alternate solution, for hardware systems that support this, is to lock down lines in the cache. System software can load critical data and instructions into the cache and instruct the cache to disable their replacement. This gives programmers the ability to keep what they need in cache and to let the caching mechanism manage less-critical instructions. The chief disadvantage of this approach is that, once data and instructions have been pinned down, it is not possible to reorganize the cache contents without significant overhead. A good overview of customization techniques for the memory architecture of embedded systems is presented in [2].

A cache improves program performance for the average case, which can lead to unpredictable behavior for real-time systems with critical and consistent performance requirements. An alternate solution for real-time systems is to use software-managed caches, a flexible, low-overhead mechanism that allows software to steer the replacement decision in the cache [3].

For example, an instruction that accesses memory can be annotated with an attribute, called a cache hint, to specify whether it should be retained in the cache or not. The way this works is that the application running on the embedded system is first instrumented to determine memory access patterns. Once locality of data access patterns is determined, cache hints are annotated to the memory instructions of the original program. This can result in performance improvements without the unpredictability attached to traditional hardware-managed cache.

Virtual memory
Virtual memory allows users to store data on a hard disk, but still use it as if it were available in memory. The application makes accesses to the data in a virtual address space that is mapped to memory, whereas the actual data physically resides on the hard disk and is moved to memory for access. Virtual memory allows access to more RAM space than is physically available on the system. In a multi-tasking application, each task can have its own independent virtual address space called a discrete address space. The operating system on the computer is responsible for virtual memory management, with some support from a hardware memory management unit (MMU). Smaller embedded systems, however, do not have virtual memory or an MMU to handle the virtual memory.

Scratch pad
A scratch pad SRAM memory is an alternative to cache for on-chip storage. It is a small, high-speed internal memory used for temporary storage of calculations, data, and other work in progress. Scratch pads are better than traditional caches in terms of power, performance, area, and predictability, making them popular in real-time systems such as multimedia applications and graphic controllers. DSPs (digital signal processors) typically use scratch pads.

In embedded systems, the application code is known a priori, and the critical code and data, as identified by the embedded system designer, can be carefully placed in the scratch pad. This identification of which data to be placed in the scratch pad is performed while the application is
being designed, and can be done either manually using compiler directives, or automatically using a compiler. This is in contrast to cache memory systems, where the mapping of program elements is done during runtime.

**Efficiency of use**

Efficiency of use is dictated by the degree of locality of reference, similar to the cache. Scratch pad SRAM guarantees a single-cycle access time while access to the cache is subject to cache hits and misses. The address space of data mapped onto the scratch pad memory is disjoint from the address space of the main memory. An embedded system designer has to figure out how to partition the available on-chip memory space into data cache and scratch pad memory so that the total access time and power dissipation is minimized.

Figure 4.4 shows the architectural block diagram of an embedded core processor with both scratch pad and data cache [4]. The address and data busses from the CPU connect to the data cache, scratch pad memory, and the external memory interface (EMI) blocks. On a memory access request from the CPU, first the data cache is searched, and if there is a hit the cache transfers the data to the CPU. In the case of a cache miss the scratch pad is searched and if there is a hit, the scratch pad gains control of the data bus and transfers the data to the CPU. In the case of a miss, the EMI transfers a block of data equal to the cache line size from the external memory into the cache and CPU. Data transfer is managed using a DMA (direct memory access).

![Figure 4.4. Scratch pad memory.](image)

Scratch pad memory can also be used for software overlays as described in the next section.

**Software overlays**

A real-time embedded processor may choose not to have a built-in cache and cache controller. Instead, it may choose to utilize the on-chip SRAM memory as a scratch pad that can be used to store frequently used code by using software overlays.

Each code section mapped onto an overlay has a run space and live space. Live space is the space in the main memory where the code section resides when not running. Run space is the space in
internal memory where the code section resides during execution. Software called an overlay manager is responsible for moving the code from live space to run space. The linker and loader tools have to provide support for generating overlay symbols for the code sections to be mapped to overlays. The overlay symbols also contain information about the run space and the live space of the overlay. This information is used by the overlay manager to move the overlays dynamically. There can be multiple overlays in a system, each having a different live space but the same run space.

The embedded system programmer is responsible for identifying mutually exclusive code sections in the application in order to be able to use this feature. The time between swaps of these sections should be high so that they don’t degrade the performance by switching too frequently. The overlay section should not be larger than the run space available. Software overlays are normally used for code and rarely for data.

Adapted from "Real World Multicore Embedded Systems, 1st Edition," B Moyer, Editor (Newnes)

**Dr. Gitu Jain** is a Software Engineer at Synopsys and also teaches at the UC Santa Cruz Extension in Silicon Valley. She has 20 years of experience in software R&D at semiconductor companies, with expertise in parallel computing and EDA algorithm design. She has a Ph.D. in Electrical and Computer Engineering from the University of Iowa.

If you liked this and would like to see a weekly collection of related products and features delivered directly to your inbox, [click here to sign up for the EDN on Systems Design newsletter](#).