privilege level (CPL) is numerically less (more privileged) than or equal to the RFLAGS.IOPL field, otherwise a general-protection exception (#GP) occurs.

Only software running at CPL = 0 can change the RFLAGS.IOPL field. Two instructions, POPF and IRET, can be used to change the field. If application software (or any software running at CPL>0) attempts to change RFLAGS.IOPL, the attempt is ignored.

System software uses RFLAGS.IOPL to control the privilege level required to access I/O-address space devices. Access can be granted on a program-by-program basis using different copies of RFLAGS for every program, each with a different IOPL. RFLAGS.IOPL acts as a global control over a program’s access to I/O-address space devices. System software can grant less-privileged programs access to individual I/O devices (overriding RFLAGS.IOPL) by using the I/O-permission bitmap stored in a program’s TSS. For details about the I/O-permission bitmap, see “I/O-Permission Bitmap” in Volume 2.

### 3.9 Memory Optimization

Generally, application software is unaware of the memory hierarchy implemented within a particular system design. The application simply sees a homogenous address space within a single level of memory. In reality, both system and processor implementations can use any number of techniques to speed up accesses into memory, doing so in a manner that is transparent to applications. Application software can be written to maximize this speed-up even though the methods used by the hardware are not visible to the application. This section gives an overview of the memory hierarchy and access techniques that can be implemented within a system design, and how applications can optimize their use.

#### 3.9.1 Accessing Memory

Implementations of the AMD64 architecture *commit* the results of each instruction—i.e., store the result of the executed instruction in software-visible resources, such as a register (including flags), the data cache, an internal write buffer, or memory—in program order, which is the order specified by the instruction sequence in a program. Transparent to the application, implementations can execute instructions in any order and temporarily hold out-of-order results until the
instructions are committed. Implementations can also *speculatively* execute instructions—executing instructions before knowing their results will be used (for example, executing both sides of a branch). By executing instructions out-of-order and speculatively, a processor can boost application performance by executing instructions that are ready, rather than delaying them behind instructions that are waiting for data. However, the processor commits results in program order (the order expected by software).

When executing instructions out-of-order and speculatively, processor implementations often find it useful to also allow out-of-order and speculative memory accesses. However, such memory accesses are potentially visible to software and system devices. The following sections describe the architectural rules for memory accesses. See “Memory System” in Volume 2 for information on how system software can further specify the flexibility of memory accesses.

**Read Ordering.** The ordering of memory reads does not usually affect program execution because the ordering does not usually affect the state of software-visible resources. The rules governing read ordering are:

- *Out-of-order reads are allowed.* Out-of-order reads can occur as a result of out-of-order instruction execution. The processor can read memory out-of-order to prevent stalling instructions that are executed out-of-order.

- *Speculative reads are allowed.* A speculative read occurs when the processor begins executing a memory-read instruction before it knows whether the instruction’s result will actually be needed. For example, the processor can predict a branch to occur and begin executing instructions following the predicted branch, before it knows whether the prediction is valid. When one of the speculative instructions reads data from memory, the read itself is speculative.

- *Reads can usually be reordered ahead of writes.* Reads are generally given a higher priority by the processor than writes because instruction execution stalls if the read data required by an instruction is not immediately available. Allowing reads ahead of writes usually maximizes software performance.

Reads can be reordered ahead of writes, except that a read *cannot* be reordered ahead of a prior write if the read is from
the same location as the prior write. In this case, the read instruction stalls until the write instruction is committed. This is because the result of the write instruction is required by the read instruction for software to operate correctly.

Some system devices might be sensitive to reads. Normally, applications do not have direct access to system devices, but instead call an operating-system service routine to perform the access on the application’s behalf. In this case, it is system software’s responsibility to enforce strong read-ordering.

**Write Ordering.** Writes affect program order because they affect the state of software-visible resources. The rules governing write ordering are restrictive:

- **Generally, out-of-order writes are not allowed.** Write instructions executed out-of-order cannot commit (write) their result to memory until all previous instructions have completed in program order. The processor can, however, hold the result of an out-of-order write instruction in a private buffer (not visible to software) until that result can be committed to memory.

  System software can create non-cacheable *write-combining* regions in memory when the order of writes is known to not affect system devices. When writes are performed to write-combining memory, they can appear to complete out of order relative to other writes. See “Memory System” in Volume 2 for additional information.

- **Speculative writes are not allowed.** As with out-of-order writes, speculative write instructions cannot commit their result to memory until all previous instructions have completed in program order. Processors can hold the result in a private buffer (not visible to software) until the result can be committed.

3.9.2 **Forcing Memory Order**

**Special instructions are provided for application software to force memory ordering in situations where such ordering is important. These instructions are:**

- **Load Fence**—The LFENCE instruction forces ordering of memory loads (reads). All memory loads preceding the LFENCE (in program order) are completed prior to completing memory loads following the LFENCE. Memory loads cannot be reordered around an LFENCE instruction,
but other non-serializing instructions (such as memory writes) can be reordered around the LFENCE.

- **Store Fence**—The SFENCE instruction forces ordering of memory stores (writes). All memory stores preceding the SFENCE (in program order) are completed prior to completing memory stores following the SFENCE. Memory stores cannot be reordered around an SFENCE instruction, but other non-serializing instructions (such as memory loads) can be reordered around the SFENCE.

- **Memory Fence**—The MFENCE instruction forces ordering of all memory accesses (reads and writes). All memory accesses preceding the MFENCE (in program order) are completed prior to completing any memory access following the MFENCE. Memory accesses cannot be reordered around an MFENCE instruction, but other non-serializing instructions that do not access memory can be reordered around the MFENCE.

Although they serve different purposes, other instructions can be used as read/write barriers when the order of memory accesses must be strictly enforced. These read/write barrier instructions force all prior reads and writes to complete before subsequent reads or writes are executed. Unlike the fence instructions listed above, these other instructions alter the software-visible state. This makes these instructions less general and more difficult to use as read/write barriers than the fence instructions, although their use may reduce the total number of instructions executed. The following instructions are usable as read/write barriers:

- **Serializing instructions**—Serializing instructions force the processor to commit the serializing instruction and all previous instructions before the next instruction is fetched from memory. The serializing instructions available to applications are CPUID and IRET. A serializing instruction is committed when the following operations are complete:
  - The instruction has executed.
  - All registers modified by the instruction are updated.
  - All memory updates performed by the instruction are complete.
  - All data held in the write buffers have been written to memory. (Write buffers are described in “Write Buffering” on page 121).
- **I/O instructions**—Reads from and writes to I/O-address space use the IN and OUT instructions, respectively. When the processor executes an I/O instruction, it orders it with respect to other loads and stores, depending on the instruction:
  - IN instructions (IN, INS, and REP INS) are not executed until all previous stores to memory and I/O-address space are complete.
  - Instructions following an OUT instruction (OUT, OUTS, or REP OUTS) are not executed until all previous stores to memory and I/O-address space are complete, including the store performed by the OUT.

- **Locked instructions**—A locked instruction is one that contains the LOCK instruction prefix. A locked instruction is used to perform an atomic read-modify-write operation on a memory operand, so it needs exclusive access to the memory location for the duration of the operation. Locked instructions order memory accesses in the following way:
  - All previous loads and stores (in program order) are completed prior to executing the locked instruction.
  - The locked instruction is completed before allowing loads and stores for subsequent instructions (in program order) to occur.

Only certain instructions can be locked. See “Lock Prefix” in Volume 3 for a list of instructions that can use the LOCK prefix.

### 3.9.3 Caches

Depending on the instruction, operands can be encoded in the instruction opcode or located in registers, I/O ports, or memory locations. An operand that is located in memory can actually be physically present in one or more locations within a system’s **memory hierarchy**.

**Memory Hierarchy.** A system’s memory hierarchy may have some or all of the following levels:

- **Main Memory**—Main memory is external to the processor chip and is the memory-hierarchy level farthest from the processor's execution units. All physical-memory addresses are present in main memory, which is implemented using relatively slow, but high-density memory devices.

- **External Caches**—External caches are external to the processor chip, but are implemented using lower-capacity, higher-performance memory devices than system memory.
The system uses external caches to hold copies of frequently-used instructions and data found in main memory. A subset of the physical-memory addresses can be present in the external caches at any time. A system can contain any number of external caches, or none at all.

- **Internal Caches**—Internal caches are present on the processor chip itself, and are the closest memory-hierarchy level to the processor's execution units. Because of their presence on the processor chip, access to internal caches is very fast. Internal caches contain copies of the most frequently-used instructions and data found in main memory and external caches, and their capacities are relatively small in comparison to external caches. A processor implementation can contain any number of internal caches, or none at all. Implementations often contain a first-level instruction cache and first-level data (operand) cache, and they may also contain a higher-capacity (and slower) second-level internal cache for storing both instructions and data.

Figure 3-19 on page 121 shows an example of a four-level memory hierarchy that combines main memory, external third-level (L3) cache, and internal second-level (L2) and two first-level (L1) caches. As the figure shows, the first-level and second-level caches are implemented on the processor chip, and the third-level cache is external to the processor. The first-level cache is a split cache, with separate caches used for instructions and data. The second-level and third-level caches are unified (they contain both instructions and data). Memory at the highest levels of the hierarchy have greater capacity (larger size), but have slower access, than memory at the lowest levels.

Using caches to store frequently used instructions and data can result in significantly improved software performance by avoiding accesses to the slower main memory. Applications function identically on systems without caches and on systems with caches, although cacheless systems typically execute applications more slowly. Application software can, however, be optimized to make efficient use of caches when they are present, as described later in this section.
Figure 3-19. Memory Hierarchy Example

Write Buffering. Processor implementations can contain write buffers attached to the internal caches. Write buffers can also be present on the interface used to communicate with the external portions of the memory hierarchy. Write buffers temporarily hold data writes when main memory or the caches are busy responding to other memory-system accesses. The existence of write buffers is transparent to software. However, some of the instructions used to optimize memory-hierarchy performance can affect the write buffers, as described in “Forcing Memory Order” on page 117.

3.9.4 Cache Operation Although the existence of caches is transparent to application software, a simple understanding how caches are accessed can assist application developers in optimizing their code to run efficiently when caches are present.
Caches are divided into fixed-size blocks, called cache lines. Typically, implementations have either 32-byte or 64-byte cache lines. The processor allocates a cache line to correspond to an identically-sized region in main memory. After a cache line is allocated, the addresses in the corresponding region of main memory are used as addresses into the cache line. It is the processor’s responsibility to keep the contents of the allocated cache line coherent with main memory. Should another system device access a memory address that is cached, the processor maintains coherency by providing the correct data back to the device and main memory.

When a memory-read occurs as a result of an instruction fetch or operand access, the processor first checks the cache to see if the requested information is available. A read hit occurs if the information is available in the cache, and a read miss occurs if the information is not available. Likewise, a write hit occurs if a memory write can be stored in the cache, and a write miss occurs if it cannot be stored in the cache.

A read miss or write miss can result in the allocation of a cache line, followed by a cache-line fill. Even if only a single byte is needed, all bytes in a cache line are loaded from memory by a cache-line fill. Typically, a cache-line fill must write over an existing cache line in a process called a cache-line replacement. In this case, if the existing cache line is modified, the processor performs a cache-line writeback to main memory prior to performing the cache-line fill.

Cache-line writebacks help maintain coherency between the caches and main memory. Internally, the processor can also maintain cache coherency by internally probing (checking) the other caches and write buffers for a more recent version of the requested data. External devices can also check a processor’s caches and write buffers for more recent versions of data by externally probing the processor. All coherency operations are performed in hardware and are completely transparent to applications.

**Cache Coherency and MOESI.** Implementations of the AMD64 architecture maintain coherency between memory and caches using a five-state protocol known as MOESI. The five MOESI states are modified, owned, exclusive, shared, and invalid. See “Memory System” in Volume 2 for additional information on MOESI and cache coherency.
**Self-Modifying Code.** Software that writes into a code segment is classified as self-modifying code. To avoid cache-coherency problems due to self-modifying code, implementations of the AMD64 architecture invalidate an instruction cache line during a memory write if the instruction cache line corresponds to a code-segment memory location. By invalidating the instruction cache line, the processor is forced to write the modified instruction into main memory. A subsequent fetch of the modified instruction goes to main memory to get the coherent version of the instruction.

3.9.5 **Cache Pollution**

Because cache sizes are limited, caches should be filled only with data that is frequently used by an application. Data that is used infrequently, or not at all, is said to pollute the cache because it occupies otherwise useful cache lines. Ideally, the best data to cache is data that adheres to the principle of locality. This principle has two components: temporal locality and spatial locality.

- **Temporal locality** refers to data that is likely to be used more than once in a short period of time. It is useful to cache temporal data because subsequent accesses can retrieve the data quickly. Non-temporal data is assumed to be used once, and then not used again for a long period of time, or ever. Caching of non-temporal data pollutes the cache and should be avoided.

  Cache-control instructions ("Cache-Control Instructions" on page 124) are available to applications to minimize cache pollution caused by non-temporal data.

- **Spatial locality** refers to data that resides at addresses adjacent to or very close to the data being referenced. Typically, when data is accessed, it is likely the data at nearby addresses will be accessed in a short period of time. Caches perform cache-line fills in order to take advantage of spatial locality. During a cache-line fill, the referenced data and nearest neighbors are loaded into the cache. If the characteristics of spatial locality do not fit the data used by an application, then the cache becomes polluted with a large amount of unreferenced data.

  Applications can avoid problems with this type of cache pollution by using data structures with good spatial-locality characteristics.
Another form of cache pollution is *stale data*. Data that adheres to the principle of locality can become stale when it is no longer used by the program, or won’t be used again for a long time. Applications can use the CLFLUSH instruction to remove stale data from the cache.

### 3.9.6 Cache-Control Instructions

General control and management of the caches is performed by system software and not application software. System software uses special registers to assign *memory types* to physical-address ranges, and page-attribute tables are used to assign memory types to virtual address ranges. Memory types define the cacheability characteristics of memory regions and how coherency is maintained with main memory. See “Memory System” in Volume 2 for additional information on memory typing.

Instructions are available that allow application software to control the cacheability of data it uses on a more limited basis. These instructions can be used to boost an application’s performance by prefetching data into the cache, and by avoiding cache pollution. Run-time analysis tools and compilers may be able to suggest the use of cache-control instructions for critical sections of application code.

**Cache Prefetching.** Applications can prefetch entire cache lines into the caching hierarchy using one of the prefetch instructions. The prefetch should be performed in advance, so that the data is available in the cache when needed. Although load instructions can mimic the prefetch function, they do not offer the same performance advantage, because a load instruction may cause a subsequent instruction to stall until the load completes, but a prefetch instruction will never cause such a stall. Load instructions also unnecessarily require the use of a register, but prefetch instructions do not.

The instructions available in the AMD64 architecture for cache-line prefetching include one SSE instruction and two 3DNow! instructions:

- *PREFETCHlevel*—(an SSE instruction) Prefetches read/write data into a specific level of the cache hierarchy. If the requested data is already in the desired cache level or closer to the processor (lower cache-hierarchy level), the data is not prefetched. If the operand specifies an invalid memory address, no exception occurs, and the instruction has no effect. Attempts to prefetch data from non-cacheable
memory, such as video frame buffers, or data from write-combining memory, are also ignored. The exact actions performed by the PREFETCH\textit{level} instructions depend on the processor implementation. Current AMD processor families map all PREFETCH\textit{level} instructions to a PREFETCH. Refer to the \textit{Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors}, order\# 25112, for details relating to a particular processor family, brand or model.

- PREFETCHT0—Prefetches temporal data into the entire cache hierarchy.
- PREFETCHT1—Prefetches temporal data into the second-level (L2) and higher-level caches, but not into the L1 cache.
- PREFETCHT2—Prefetches temporal data into the third-level (L3) and higher-level caches, but not into the L1 or L2 cache.
- PREFETCHNTA—Prefetches non-temporal data into the processor, minimizing cache pollution. The specific technique for minimizing cache pollution is implementation-dependent and can include such techniques as allocating space in a software-invisible buffer, allocating a cache line in a single cache or a specific way of a cache, etc.

- \textit{PREFETCH}—(a 3DNow! instruction) Prefetches read data into the L1 data cache. Data can be written to such a cache line, but doing so can result in additional delay because the processor must signal externally to negotiate the right to change the cache line's cache-coherency state for the purpose of writing to it.

- \textit{PREFETCHW}—(a 3DNow! instruction) Prefetches write data into the L1 data cache. Data can be written to the cache line without additional delay, because the data is already prefetched in the \textit{modified} cache-coherency state. Data can also be read from the cache line without additional delay. However, prefetching write data takes longer than prefetching read data if the processor must wait for another caching master to first write-back its modified copy of the requested data to memory before the prefetch request is satisfied.

The \textit{PREFETCHW} instruction provides a hint to the processor that the cache line is to be modified, and is intended for use
when the cache line will be written to shortly after the prefetch is performed. The processor can place the cache line in the modified state when it is prefetched, but before it is actually written. Doing so can save time compared to a PREFETCH instruction, followed by a subsequent cache-state change due to a write.

To prevent a false-store dependency from stalling a prefetch instruction, prefetched data should be located at least one cache-line away from the address of any surrounding data write. For example, if the cache-line size is 32 bytes, avoid prefetching from data addresses within 32 bytes of the data address in a preceding write instruction.

**Non-Temporal Stores.** Non-temporal store instructions are provided to prevent memory writes from being stored in the cache, thereby reducing cache pollution. These non-temporal store instructions are specific to the type of register they write:

- **GPR Non-Temporal Stores**—MOVNTI.
- **XMM Non-Temporal Stores**—MASKMOVQDQU, MOVNTDQ, MOVNTPD, and MOVNTPS.
- **MMX Non-Temporal Stores**—MASKMOVQ and MOVNTQ.

**Removing Stale Cache Lines.** When cache data becomes stale, it occupies space in the cache that could be used to store frequently-accessed data. Applications can use the CLFLUSH instruction to free a stale cache-line for use by other data. CLFLUSH writes the contents of a cache line to memory and then invalidates the line in the cache and in all other caches in the cache hierarchy that contain the line. Once invalidated, the line is available for use by the processor and can be filled with other data.

### 3.10 Performance Considerations

In addition to typical code optimization techniques, such as those affecting loops and the inlining of function calls, the following considerations may help improve the performance of application programs written with general-purpose instructions.

These are implementation-independent performance considerations. Other considerations depend on the hardware implementation. For information about such implementation-
dependent considerations and for more information about application performance in general, see the data sheets and the software-optimization guides relating to particular hardware implementations.

### 3.10.1 Use Large Operand Sizes

Loading, storing, and moving data with the largest relevant operand size maximizes the memory bandwidth of these instructions.

### 3.10.2 Use Short Instructions

Use the shortest possible form of an instruction (the form with fewest opcode bytes). This increases the number of instructions that can be decoded at any one time, and it reduces overall code size.

### 3.10.3 Align Data

Data alignment directly affects memory-access performance. Data alignment is particularly important when accessing streaming (also called non-temporal) data—data that will not be reused and therefore should not be cached. Data alignment is also important in cases where data that is written by one instruction is subsequently read by a subsequent instruction soon after the write.

### 3.10.4 Avoid Branches

Branching can be very time-consuming. If the body of a branch is small, the branch may be replaceable with conditional move (CMOVcc) instructions, or with 128-bit or 64-bit media instructions that simulate predicated parallel execution or parallel conditional moves.

### 3.10.5 Prefetch Data

Memory latency can be substantially reduced—especially for data that will be used multiple times—by prefetching such data into various levels of the cache hierarchy. Software can use the PREFETCHx instructions very effectively in such cases. One PREFETCHx per cache line should be used.

Some of the best places to use prefetch instructions are inside loops that process large amounts of data. If the loop goes through less than one cache line of data per iteration, partially unroll the loop. Try to use virtually all of the prefetched data. This usually requires unit-stride memory accesses—those in which all accesses are to contiguous memory locations.

For data that will be used only once in a procedure, consider using non-temporal accesses. Such accesses are not burdened by the overhead of cache protocols.
3.10.6 Keep Common Operands in Registers

Keep frequently used values in registers rather than in memory. This avoids the comparatively long latencies for accessing memory.

3.10.7 Avoid True Dependencies

Spread out true dependencies (write-read or flow dependencies) to increase the opportunities for parallel execution. This spreading out is not necessary for anti-dependencies and output dependencies.

3.10.8 Avoid Store-to-Load Dependencies

Store-to-load dependencies occur when data is stored to memory, only to be read back shortly thereafter. Hardware implementations of the architecture may contain means of accelerating such store-to-load dependencies, allowing the load to obtain the store data before it has been written to memory. However, this acceleration might be available only when the addresses and operand sizes of the store and the dependent load are matched, and when both memory accesses are aligned. Performance is typically optimized by avoiding such dependencies altogether and keeping the data, including temporary variables, in registers.

3.10.9 Optimize Stack Allocation

When allocating space on the stack for local variables and/or outgoing parameters within a procedure, adjust the stack pointer and use moves rather than pushes. This method of allocation allows random access to the outgoing parameters, so that they can be set up when they are calculated instead of being held in a register or memory until the procedure call. This method also reduces stack-pointer dependencies.

3.10.10 Consider Repeat-Prefix Setup Time

The repeat instruction prefixes have a setup overhead. If the repeated count is variable, the overhead can sometimes be avoided by substituting a simple loop to move or store the data. Repeated string instructions can be expanded into equivalent sequences of inline loads and stores. For details, see “Repeat Prefixes” in Volume 3.

3.10.11 Replace GPR with Media Instructions

Some integer-based programs can be made to run faster by using 128-bit media or 64-bit media instructions. These instructions have their own register sets. Because of this, they relieve register pressure on the GPR registers. For loads, stores, adds, shifts, etc., media instructions may be good substitutes for general-purpose integer instructions. GPR registers are freed up, and the media instructions increase opportunities for parallel operations.
3.10.12 **Organize Data in Memory Blocks**

Organize frequently accessed constants and coefficients into cache-line-size blocks and prefetch them. Procedures that access data arranged in memory-bus-sized blocks, or memory-burst-sized blocks, can make optimum use of the available memory bandwidth.