Intel 64 Multicore Functionality

By Michael Almandeel

Introduction

I would like to share what I have learned over the past couple of months. This is a combination of various structures and paradigms that relate to multicore cpu architecture and its applications. The main goal at the time of our discussion was utilizing the **m**ultiple **a**pic **d**escription **t**able provided by ACPI(a universal standard that intel implements). Although since it is interesting I will briefly talk about it, it turns out that ACPI and the madt are not precisely whats needed, although they do provide vital data necessary to accomplish my goal. I saw the idea on a seemingly reliable site, but apparently there is no structure in acpi that does what the commenter said the madt does. It just goes to show, if the data doesn't come from an official source, it may not be accurate. I had to read the acpi standard and read the official intel 64 system programming manual(vol. 3) in order to find the truth.

Since I wasn't going to let this turn into a "black hole" as you put it, I found a way to do it which will actually work. I will also discuss intel multicore cache behavior, interprocessor interrupts, and advanced apic functionality. It is important to read my previous paper "Implementing a dispatcher in Intel 64" to understand how my dispatcher works in detail, as I don't repeat such details here, unless I'm changing the dispatcher's operation(I will later).

About ACPI

The description of the madt that I found on the internet was not explained correctly, and in fact disagrees with ACPI. Officially, the madt contains metadata unrelated to the apic's underlying implementation, and instead describes revision numbers and other basic data. The boot image of the OS uses the data provided by ACPI essentially as configuration data, which the OS uses to decide which software to use to interact with the hardware. The obvious example of such software would be device drivers, but there is also other software such as that used to program the apic's and page tables of the cpu(s) at boot time.

I think the coolest facet of ACPI is it's portability. I was initially under the impression that ACPI was implemented as binary Intel instructions on the boot rom on the motherboard, but this is not the case. Interestingly, ACPI is actually stored on the boot rom as **bytecode**. At boot time, the cpu loads a **virtual machine**(typically vendor implemented), which reads the bytecode from the rom and maps that bytecode to a native machine binary which is then executed. The cpu in turn interacts with the hardware as necessary(testing, programming, storing acpi data structures in memory etc)until finally the kernel image is loaded.

At some point, the kernel also uses the data structures provided by ACPI to program various hardware on its own behalf and load the appropriate device drivers. Much more is done during boot, but I'm not going to cover it(uefi and the tpm are awesome though). I will ovcoarse talk about how the kernel can program the apic's using the madt as a rough roadmap.

Multicore Dispatching

We program each local apic via it's control registers, and specify the interrupt gates that respond to said interrupts on the IDT (see my dispatcher paper for more). In the case of multiple cores, we use the %idtr register of each core to specify a unique IDT(previously we didn't use this register as there was only 1 IDT). We set it to the local apic id(more on this later). This way, each core can have its own unique set of interrupt service routines.

The most important thing to realize is that a local apic's control registers can only be read from and written to by the actual core to which it corresponds. We cannot access the registers from any other core. This can be confusing, because these registers are memory mapped. However, the memory addresses used are the same for each core. Reading from or writing to one of these addresses on a given core accesses the associated register on its own local apic.

Anyway, we will assume that the booted kernel image has done some work, and at some point each core jumps to the entry point of the code that we use to program the apic. We use the madt to determine which code we will use, because the revision number determines the apic's capabilities and therefore specific register addresses, and the number of cores determines how many sets of dispatcher software copies and thread scheduler queues we will need, and how much memory must be allocated for the sets.

When we are programming the apic, we execute the CPUID instruction with leaf 0Bh. Within the value returned in %rax, is the local apic id(this serves as the id of the associated core). This value is used as an index into an array in the kernel(we built the array using madt data) that stores the entry point of each core's task dispatcher, interrupt manager and burst time manager code, which is already loaded in memory.

Note that each dispatcher and its associated modules(i.e the burst time and interrupt managers) are implemented as relocatable code and are updated to reference the address(s) of its own task queue(s) , each instruction's offset relative to the module's actual entry point, and the entry points of associated modules where applicable when they are loaded.

Besides the above differences, the dispatcher is set up and functions as described in my previous paper. Its only now we have a different copy of the dispatcher and a unique IDT for each core, at the trade off of a more costly setup routine. This enables the scheduler to keep track of each cores queue and allows it to schedule threads for a specific core by writing to its associated queue(s).

I have chosen to use a shared scheduler for versatility, but it requires locking the queue of the core on which it is scheduling a process. Now the dispatcher must test and set the lock before it passes control to the next task(which it does immediately if it obtains the lock). In case the lock is not free, it simply IRETQ's back to the scheduler. This will certainly slow down this cores dispatcher a bit, but there is no such thing as a free lunch. If we decide to use a different scheduler for each core we are pretty much confined to running a task on one specific core(if using a lock free implementation),while still having to lock anyway anytime we want to switch cores.

An affinity mask(in windows) is a bit vector created by the user at process creation time, indicating on which cores(bits) this processes threads will be permitted to run. It’s also possible to set an affinity mask for each individual thread. The scheduler keeps track of which cores are in the affinity group(via the mask) and only schedules the thread on the ready queues of eligible cores. It is thru this mechanism that we can take advantage of **processor affinity.** It can also be used to orchestrate several threads in parallel with a high level of control.

Advanced Apic Functionality

In my previous paper, I spoke about the apic briefly and described how to manage the timer to facilitate burst time based dispatching. The apic(particularly its local variant), has advanced capabilities beyond that of an ordinary interrupt controller. The simplified view of interrupts I described last time is dated. Advanced programmable interrupt controllers have new powerful features, such as inter processor interrupts and priority based interrupt dispatching.

In the modern apic, maskable interrupts are not only interruptible by NMIs, but by any interrupt with a higher **priority**. This behavior enables priority based, preemptive scheduling as seen in a modern operating system.

Priority is indicated by the numerical value of the interrupt vector. The lower the vector, the higher the priority. Effectively, a maskable interrupt can only be interrupted by an interrupt with a lower vector. The local apic has the TPR and PPR registers to assist in the delivery ordering process.

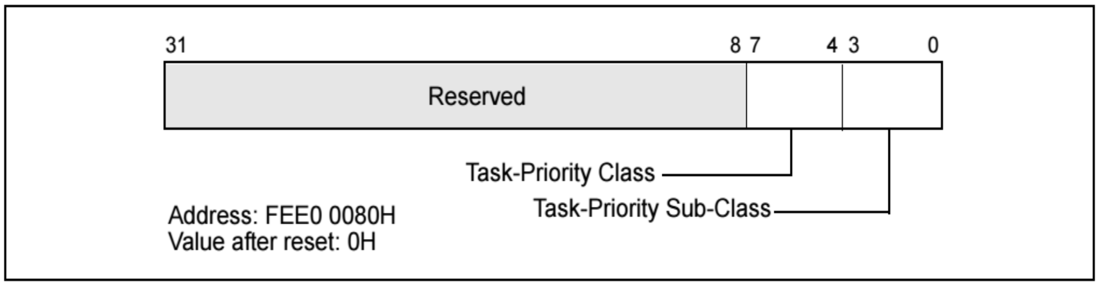

Figure 11-18. Task-Priority Register (TPR)

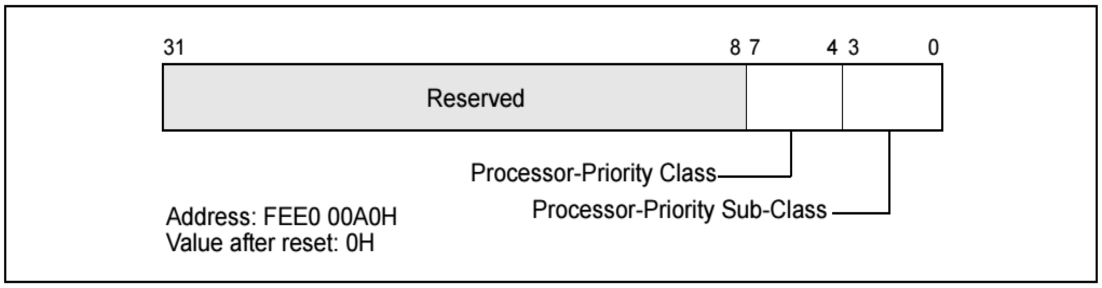

Figure 11-19. Processor-Priority Register (PPR)

The processor priority register indicates the interrupt level at which the core is currently running. Effectively, the PPR is equal to the vector currently being serviced by the core. As noted, the apic will not interrupt execution unless the interrupt has a higher priority(lower vector). Interrupts which are not serviced immediately are stored for later delivery. The PPR helps the apic make the decision of whether or not it should interrupt the core immediately or wait until later.

The task priority register provides a mechanism for a dispatcher to dispatch a task at an arbitrary priority level. When you set the TPR, you are instructing the local apic to mask interrupts of lower priority(higher vector) until the TPR is set to a value to which they are less than or equal to. If an interrupt is a higher priority than the TPR, it will interrupt the currently executing program as usual.

Using the TPR, the kernal can execute important systems threads at a high priority, allowing them to complete their work uninterrupted(mostly). An important thing to keep in mind when designing a system like this is that the vector of the burst timer *must* be lower than the value you set in the TPR. If not, the dispatched program can hijack the core and prevent other programs from being dispatched.

Another interesting function provided by apics is the **i**nter **p**rocessor **i**nterrupt, or IPI. This allows apics to send interrupts to each other. The target of an IPI can be a specific apic, a group of them, or all of them. An apic can even send IPI's to itself. When an IPI is sent, it is transmitted on the system bus. The apic 'snoops'(more on this later) the bus to determine if the IPI is intended for it, and fires or masks the interrupt based on the TPR and PPR. An IPI message contains an interrupt vector number which the apic issues to the core just as if it were a local interrupt(timer etc…).

A multiprocessor system can use IPI's for intercore communication as well as maximizing thruput of high priority interrupt service. This is particularly useful for real time systems, when you have high priority code that **must run now,** but the program running currently is also of high(but slightly lower priority). Ipis allow you to defer the interrupt to another core(even the core with the lowest PPR value, if you desire).

Another interesting use of the IPI is sending an interrupt to one's self. Some programmers use software interrupts naively expecting the interrupt to fire and run its isr at its native priority . Unfortunately, software interrupts merely call the isr associated with the specified vector, while still executing at the current interrupt priority level, as the apic is not actually involved in its functionality. Typically we want to execute the isr at its native interrupt level, using the apic to order the interrupt; an IPI for the specified vector addressed to self does just that.

One reason we would want to do this would be for a paradigm called a 'deferred procedure call' or DPC(windows). DPCs are useful to schedule less important work for later, after more critical work has been done. We issue a DPC by firing an ipi at self with a lower number than the PPR. This way we free the core to respond to more important interrupts, and service the DPC when there isn't any more important work to be done.

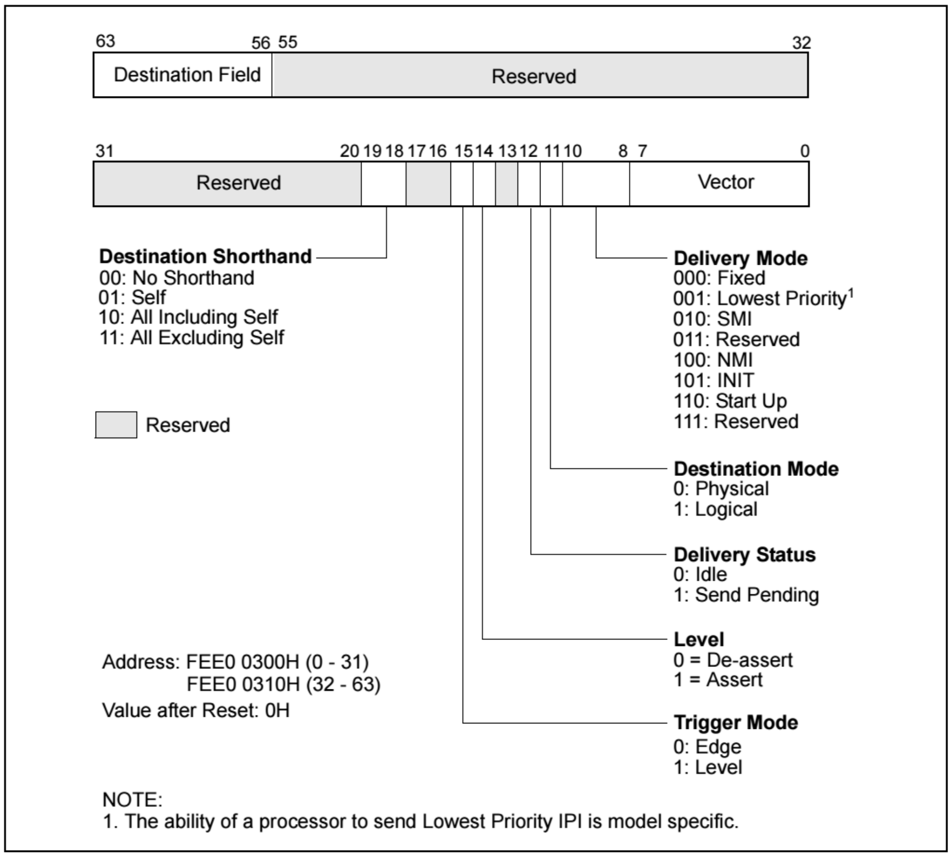

Figure 11-12. Interrupt Command Register (ICR)

We can send IPIs using the **i**nterrupt **c**ontrol **r**egister, or ICR. In the vector field, we indicate the interrupt vector we would like the target apic(s) to fire. For basic addressing, there is a destination shorthand field, which can address either self, all cores, or all cores minus self. If we don't specify a shorthand, the destination field is used; in it we indicate the local apic id of the target core, or the group id of the target core(s). Once we have finished modifying the ICR, we set the delivery status bit to send pending(this will indicate to the apic its ready to be sent). The apic sends the IPI, and sets the delivery status bit to idol. This indicates a new IPI may be issued using the ICR.

The Cache

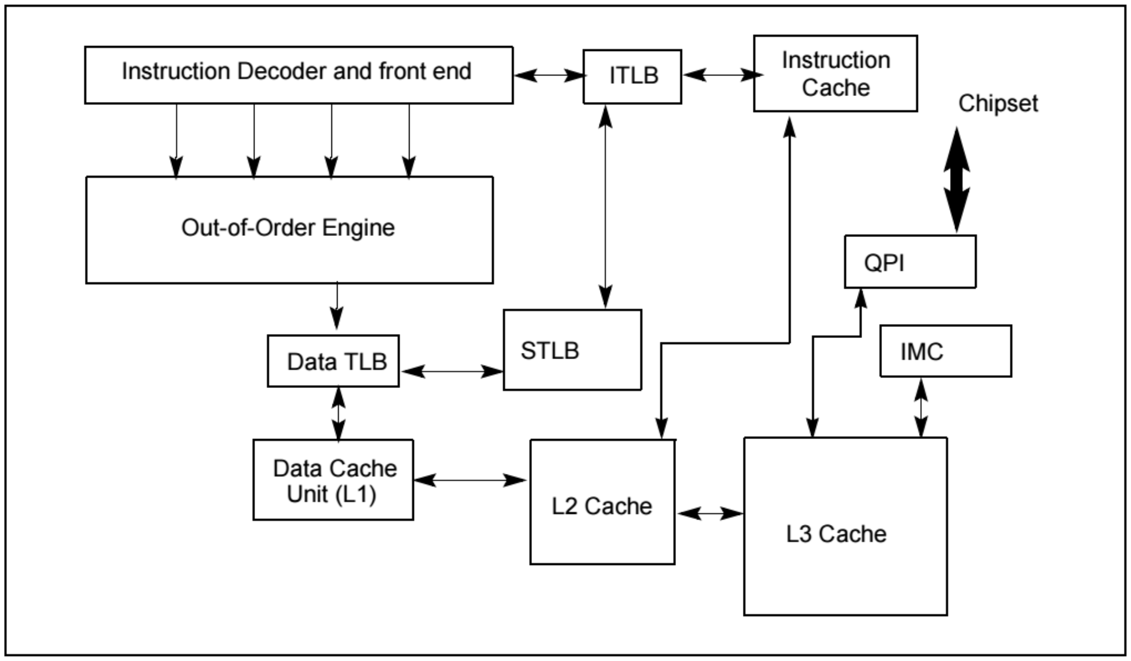

Figure 12-2. Cache Structure of the Intel Core i7 Processors

Since the audience of this paper knows a lot about caches already, I won't bother with the obligatory introduction to what a cache is or what it does. I'll discuss the different memory models, memory partitioning, cache coherence mechanisms, and strategies used to minimize the number of dram bus accesses implemented by Intel.

Modern Intel systems rely on many optimizations to increase bandwidth, and the cache architecture is no exception. Let's begin with the TLB(translation lookaside buffer). Instead of the traditional singleton TLB, there is now a separate TLB each for instructions and data, as well as a level 2 unified STLB, which like L2 cache provides significantly greater storage at the expense of bandwidth. In keeping with the theme, there is also a separate L1 cache each for instructions and data, which share an L2 cache much like the TLBs do. All cores share the same L3 cache.

The cache structure is connected to the system bus. The dram bus and the i/o busses(pci etc) are connected to the system bus as well. Each element that connects to the system bus has it's own controller(s) and buffer(s). These ensure that access to the system bus is properly arbitrated, and all transactions on the system bus are eventually executed.

Before going further, it's important to realize how memory is actually transferred between various components on the system bus. Dram reads and writes are only allowed in 1, 2, 4, 8, or 64 byte sequences, and can only read sequences atomically when the initial address of the operation begins at an **alignment index**. To be clear, we can only actually address memory at an integer multiple of its size, so for instance we can only address 32 bit transfers at the physical addresses of 0, 32, 64, 96… 32n(these are the alignment indexes). A virtual divider called the **alignment boundary** exist between every alignment index and it's preceding byte. When we request transfer of a 64 bit sequence out of alignment, what actually happens is two transfers occur to two different registers, which are shifted and or-d together into a single register to reconstruct the actual data that you requested. This process is transparent to the programmer.

The 64 byte transfer units are known as **cache lines**. As described earlier, a cache line is a 64 byte sequence of memory aligned on a 64 byte boundary. When a memory operation such as a load or store takes place between a core and dram , an entire cache line is transferred regardless of the actual amount of data specified by the instruction. For dram, the cache line which is transferred is the one in which the requested data resides. In the case that the data resides on both sides of a 64 byte alignment boundary, two cache lines will be transferred. As indicated above, this will result in the transfer of the data not being atomic. I/O transfers directly to memory can be of any size mentioned earlier, and utilize a paradigm called **d**irect **m**emory **a**ccess, or DMA.

Cache lines are useful because they reduce the number of dram bus operations. A useful performance optimization involves maximizing the number of spatially local operands(in physical memory) among temporally local instructions. This results in a significant reduction in the number of dram requests, which allows the core to operate with fewer memory stalls.

A very important thing to note regardless of the transfer size is that accessing data across alignment boundaries requires the system to perform additional work in order to service the request, and as a result there is a significant performance hit. Fast software takes into account the alignment boundaries and will even zero-pad data in order to avoid storing data across the boundaries. When you see padding and it is seemingly for no good reason, it's probably being done to respect alignment, as we all should.

Memory models are effectively sets of rules and strategies for caching, accessing and updating memory. Since memory is used for many different purposes, it follows that there are many different patterns of reads and writes when interacting with it. Certain patterns benefit from specific types of memory behavior, specifically as it relates to the cache. The memory models that the chip supports are dependent on its capabilities, especially as it relates to the complexity of the circuitry that supports the cache(ie snooping). Modern Intel processors support all of the memory models I will discuss, but older series of cpus may not.

In order to enforce the various memory models, cpus need additional circuitry to support functions that enable the enforcement. Examples are write-combined buffers, load/store buffers, and snooping circuitry. Snooping is the ability of a core to examine the state of other core's caches and the system bus. By snooping and taking certain actions based on the observed state, we can enforce memory consistency among all caches and dram. This is generally referred to as **cache coherency**.

We will look at some of the models very briefly, as they are not generally used. Modern Intel systems are capable of partitioning physical memory such that each partition is managed and interacted with in accordance with the model of the programmers choosing. You can use the memory type range registers for this purpose, but discussion of them isn't covered here.

The first models we will cover are **uncacheable** and **strong uncacheable** memory. As you may have guessed, there is no use of the cache other than being an intermediary between the dram and the core. The cache is not relied upon to store the state, all reads and writes propagate over the system bus to/from dram. This type of memory is used for i/o, which uses DMA to write to dram.

Write combining cache is similar to the uncacheable types in that it does not rely on the cache to store state. However, it transmits writes to dram in burst, using a write-combining buffer to queue the writes that occur between bursts. On a burst, the WC buffer is drained to memory or i/o via the system bus. WC cache is not often used outside of writing to a frame buffer(to display graphics).

Write protected and **write thru** types are very similar. They are the first of the models that actually rely on the cache to enforce memory consistency. When a cores reads memory, they will either get it from the cache(because its there, and valid), or dram(which causes a cache line fill). When a core writes to memory, it is written to its corresponding line in the cache(if present) on the local core, and thru the system bus to dram. While this is happening, the other cores are snooping the system bus. They are interested in writes that correspond to any valid cache lines they currently store. If they see any that do, they either invalidate(WP, some WT) or update(some WT) the stored cache line. In addition to this, each core snoops the instruction caches of the other cores, and looks for writes to cache lines it currently stores. If it finds any, it marks its corresponding line in the cache invalid. The instant write back to the system bus along with the two kinds of snooping mentioned above ensure memory consistency throughout the entire system.

Write back cache is the most complex, and most efficient of the bunch. Coincidentally, this is the type most likely to be used in a modern system. Instead of writing back writes to a cache line immediately(as above), it waits. This reduces traffic on the system bus, and reduces writes to dram. It only writes back when a read request on that line is made. It uses the same types of snooping mentioned above, along with a couple more(these are needed to ensure consistency in the new model). In addition to snooping the bus for writes(upon which it invalidates the corresponding line), it snoops the bus for reads too. If it holds a valid cache line to satisfy the read request, it writes it back to the system bus as if it were dram itself. Dram notices this(by snooping) and cancels its own servicing of the request, in addition to writing the transmitted value back to dram as well. This is transparent to the reading core. In addition to that, each core snoops the other cores caches looking for valid cache lines in those cores that it has changed and not yet written back. When it finds one, it sends a stall signal to the target core and causes the offending cache line to be invalidated. With the addition of these new paradigms, we can enforce cache coherence in write back systems.

References:

https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/05_ACPI_Software_Programming_Model/ACPI_Software_Programming_Model.html (all acpi tables)

https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/05_ACPI_Software_Programming_Model/ACPI_Software_Programming_Model.html#multiple-apic-description-table-madt (madt)

https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/20_AML_Specification/AML_Specification.html#aml-specification (acpi high level language)

https://cdrdv2.intel.com/v1/dl/getContent/671447 (Intel 64 manual, see “cache control”, “local apic” and “multi processor support”)