A deep dive into CMA

March 14, 2012
This article was contributed by Michal "mina86" Nazarewicz

The Contiguous Memory Allocator (or CMA), which LWN looked at back in June 2011, has been developed to allow allocation of big, physically-contiguous memory blocks. Simple in principle, it has grown quite complicated, requiring cooperation between many subsystems. Depending on one's perspective, there are different things to be done and watch out for with CMA. In this article, I will describe how to use CMA and how to integrate it with a given platform.

From a device driver author's point of view, nothing should change. CMA is integrated with the DMA subsystem, so the usual calls to the DMA API (such as `dma_alloc_coherent()`) should work as usual. In fact, device drivers should never need to call the CMA API directly, since instead of bus addresses and kernel mappings it operates on pages and page frame numbers (PFNs), and provides no mechanism for maintaining cache coherency.

For more information, looking at `Documentation/DMA-API.txt` and `Documentation/DMA-API-HOWTO.txt` will be useful. Those two documents describe the provided functions as well as giving usage examples.

Architecture integration

Of course, someone has to integrate CMA with the DMA subsystem of a given architecture. This is performed in a few, fairly easy steps.

CMA works by reserving memory early at boot time. This memory, called a *CMA area* or a *CMA context*, is later returned to the buddy allocator so that it can be used by regular applications. To do the reservation, one needs to call:

```
void dma_contiguous_reserve(phys_addr_t limit);
```

just after the low-level "memblock" allocator is initialized but prior to the buddy allocator setup. On ARM, for example, it is called in `arm_memblock_init()`, whereas on x86 it is just after memblock is set up in `setup_arch()`.

The `limit` argument specifies physical address above which no memory will be prepared for CMA. The intention is to limit CMA contexts to addresses that DMA can handle. In the case of ARM, the limit is the minimum of `arm_dma_limit` and `arm_lowmem_limit`. Passing zero will allow CMA to allocate its context as high as it wants. The only constraint is that the reserved memory must belong to the same zone.

The amount of reserved memory depends on a few Kconfig options and a `cma` kernel parameter. I will describe them further down in the article.

The `dma_contiguous_reserve()` function will reserve memory and prepare it to be used with CMA. On some architectures (eg. ARM) some architecture-specific work needs to be performed as well. To allow that, CMA will call the following function:

```
void dma_contiguous_early_fixup(phys_addr_t base, unsigned long size);
```

It is the architecture's responsibility to provide it along with its declaration in the `asm/dma-contiguous.h` header file. If a given architecture does not need any special handling, it's enough to provide an empty function definition.

It will be called quite early, thus some subsystems (e.g. `kmalloc()`) will not be available. Furthermore, it may be called several times (since, as described below, several CMA contexts may exist).

The second thing to do is to change the architecture's DMA implementation to use the whole machinery. To allocate CMA memory one uses:

```
struct page *dma_alloc_from_contiguous(struct device *dev, int count, unsigned int align);
```

Its first argument is a device that the allocation is performed on behalf of. The second specifies the *number of pages* (not bytes or order) to allocate. The third argument is the alignment expressed as a page order. It enables allocation of buffers whose physical addresses are aligned to 2^{align} pages. To avoid fragmentation, if at all possible pass zero here. It is worth noting that there is a Kconfig option (`CONFIG_CMA_ALIGNMENT`) which specifies maximum alignment accepted by the function. Its default value is 8 meaning 256-page alignment.

The return value is the first of a sequence of `count` allocated pages.

To free the allocated buffer, one needs to call:

```
bool dma_release_from_contiguous(struct device *dev, struct page *pages, int count);
```

The `dev` and `count` arguments are same as before, whereas `pages` is what `dma_alloc_from_contiguous()` returned. If the region passed to the function did not come from CMA, the function will return `false`. Otherwise, it will return `true`. This removes the need for higher-level functions to track which allocations were made with CMA and which were made using some other method.

Beware that `dma_alloc_from_contiguous()` may not be called from atomic context. It performs some “heavy” operations such as page migration, direct reclaim, etc., which may take a while. Because of that, to make `dma_alloc_coherent()` and friends work as advertised, the architecture needs to have a different method of allocating memory in atomic context.

The simplest solution is to put aside a bit of memory at boot time and perform atomic allocations from that. This is in fact what ARM is doing. Existing architectures most likely already have a special path for atomic allocations.

Special memory requirements

At this point, most of the drivers should “just work”. They use the DMA API, which calls CMA. Life is beautiful. Except some devices may have special memory requirements. For instance, Samsung's S5P Multi-format codec requires buffers to be located in different memory banks (which allows reading them through two memory channels, thus increasing memory bandwidth). Furthermore, one may want to separate some devices' allocations from others to limit fragmentation within CMA areas.

CMA operates on contexts. Devices use one global area by default, but private contexts can be used as well. There is a many-to-one mapping between `struct device`s and a `struct cma` (ie. CMA context). This means that a single device driver needs to have separate `struct device` objects to use more than one CMA context, while at the same time several `struct device` objects may point to the same CMA context.

To assign a CMA context to a device, all one needs to do is call:

```
int dma_declare_contiguous(struct device *dev, unsigned long size,
                           phys_addr_t base, phys_addr_t limit);
```

As with `dma_contiguous_reserve()`, this needs to be called after memblock initializes but before too much memory gets grabbed from it. For ARM platforms, a convenient place to put the call to this function is in the machine's `reserve()` callback. This won't work for automatically probed devices or those loaded as modules, so some other mechanism will be needed if those kinds of devices require CMA contexts.

The first argument of the function is the device that the new context is to be assigned to. The second specifies the *size in bytes* (not in pages) to reserve for the areas. The third is the physical address of the area or zero. The last one has the same meaning as `dma_contiguous_reserve()`'s `limit` argument. The return value is either zero or a negative error code.

There is a limit to how many “private” areas can be declared, namely `CONFIG_CMA_AREAS`. Its default value is seven but it can be safely increased if the need arises.

Things get a little bit more complicated if the same non-default CMA context needs to be used by two or more devices. The current API does not provide a trivial way to do that. What can be done is to use `dev_get_cma_area()` to figure out the CMA area that one device is using, and `dev_set_cma_area()` to set the same context to another device. This sequence must be called no sooner than in `postcore_initcall()`. Here is how it might look:

```
static int __init foo_set_up_cma_areas(void)
{
    struct cma *cma;

    cma = dev_get_cma_area(device1);
    dev_set_cma_area(device2, cma);
    return 0;
}
postcore_initcall(foo_set_up_cma_areas);
```

As a matter of fact, there is nothing special about the default context that is created by `dma_contiguous_reserve()` function. It is in no way required and the system will work without it. If there is no default context, `dma_alloc_from_contiguous()` will return `NULL` for devices without assigned areas. `dev_get_cma_area()` can be used to distinguish between this situation and allocation failure.

`dma_contiguous_reserve()` does not take a size as an argument, so how does it know how much memory should be reserved? There are two sources of this information:

There is a set of Kconfig options, which specify the default size of the reservation. All of those options are located under “Device Drivers” » “Generic Driver Options” » “Contiguous Memory Allocator” in the Kconfig menu. They allow choosing from four possibilities: the size can be an absolute value in megabytes, a percentage of total memory, the smaller of the two, or the larger of the two. The default is to allocate 16 MiBs.

There is also a `cma=` kernel command line option. It lets one specify the size of the area at boot time without the need to recompile the kernel. This option specifies the size in bytes and accepts the usual suffixes.

So how does it work?

To understand how CMA works, one needs to know a little about migrate types and pageblocks.

When requesting memory from the buddy allocator, one provides a `gfp_mask`. Among other things, it specifies the "migrate type" of the requested page(s). One of the migrate types is `MIGRATE_MOVABLE`. The idea behind it is that data from a movable page can be migrated (or moved, hence the name), which works well for disk caches, process pages, etc.

To keep pages with the same migrate type together, the buddy allocator groups pages into "pageblocks," each having a migrate type assigned to it. The allocator then tries to allocate pages from pageblocks with a type

corresponding to the request. If that's not possible, however, it *will* take pages from different pageblocks and may even change a pageblock's migrate type. This means that a non-movable page can be allocated from a `MIGRATE_MOVABLE` pageblock which can also result in that pageblock changing its migrate type. This is undesirable for CMA, so it introduces a `MIGRATE_CMA` type which has one important property: only movable pages can be allocated from a `MIGRATE_CMA` pageblock.

So, at boot time, when the `dma_contiguous_reserve()` and/or `dma_declare_contiguous()` functions are called, CMA talks to memblock to reserve a portion of RAM, just to give it back to the buddy system later on with the underlying pageblock's migrate type set to `MIGRATE_CMA`. The end result is that all the reserved pages end up back in the buddy allocator, so they can be used to satisfy movable page allocations.

During CMA allocation, `dma_alloc_from_contiguous()` chooses a page range and calls:

```
int alloc_contig_range(unsigned long start, unsigned long end,
                       unsigned migratetype);
```

The `start` and `end` arguments specify the page frame numbers (or the *PFN* range) of the target memory. The last argument, `migratetype`, indicates the migration type of the underlying pageblocks; in the case of CMA, this is `MIGRATE_CMA`. The first thing this function does is to mark the pageblocks contained within the `[start, end)` range as `MIGRATE_ISOLATE`. The buddy allocator will never touch a pageblock with that migrate type. Changing the migrate type does not magically free pages, though; this is why `__alloc_conting_migrate_range()` is called next. It scans the PFN range and looks for pages that can be migrated away.

Migration is the process of copying a page to some other portion of system memory and updating any references to it. The former is straightforward and the latter is handled by the memory management subsystem. After its data has been migrated, the old page is freed by giving it back to the buddy allocator. This is why the containing pageblocks had to be marked as `MIGRATE_ISOLATE` beforehand. Had they been given a different migrate type, the buddy allocator would not think twice about using them to fulfill other allocation requests.

Now all of the pages that `alloc_contig_range()` cares about are (hopefully) free. The function takes them away from buddy system, then changes pageblock's migrate type back to `MIGRATE_CMA`. Those pages are then returned to the caller.

Freeing memory is much simpler process. `dma_release_from_contiguous()` delegates most of its work to:

```
void free_contig_range(unsigned long pfn, unsigned nr_pages);
```

which simply iterates over all the pages and puts them back to the buddy system.

Epilogue

The Contiguous Memory Allocator patch set has gone a long way from its first version (and even longer from its predecessor – Physical Memory Management posted almost three years ago). On the way, it lost some of its functionality but got better at what it does now. On complex platforms, it is likely that CMA won't be usable on its own, but will be used in combination with ION and dmabuf.

Even though it is at its 23rd version, CMA is still not perfect and, as always, there's still a lot that can be done to improve it. Hopefully though, getting it finally merged into the -mm tree will get more people working on it to create a solution that benefits everyone.
