NAME

I/OAT - Intel I/O Acceleration Technology

SYNOPSIS

To compile this driver into your kernel, place the following line in your kernel configuration file:

device ioat

Or, to load the driver as a module at boot, place the following line in loader.conf(5):

ioat_load="YES"

In loader.conf(5):

hw.ioat.force_legacy_interrupts=0

In loader.conf(5) or sysctl.conf(5):

hw.ioat.enable_ioat_test=0
hw.ioat.debug_level=0 (only critical errors; maximum of 3)

typedef void
(*bus_dmaengine_callback_t)(*void *arg, int error*);

bus_dmaengine_t
ioat_get_dmaengine(*uint32_t channel_index*);

void
ioat_put_dmaengine(*bus_dmaengine_t dmaengine*);

int
ioat_get_hwversion(*bus_dmaengine_t dmaengine*);

size_t
ioat_get_max_io_size(*bus_dmaengine_t dmaengine*);

int
ioat_set_interrupt_coalesce(*bus_dmaengine_t dmaengine, uint16_t delay*);

uint16_t

ioat_get_max_coalesce_period(*bus_dmaengine_t dmaengine*);

void
ioat_acquire(*bus_dmaengine_t dmaengine*);

int
ioat_acquire_reserve(*bus_dmaengine_t dmaengine, uint32_t n, int mflags*);

void
ioat_release(*bus_dmaengine_t dmaengine*);

*struct bus_dmadesc **
ioat_copy(*bus_dmaengine_t dmaengine, bus_addr_t dst, bus_addr_t src, bus_size_t len, bus_dmaengine_callback_t callback_fn, void *callback_arg, uint32_t flags*);

*struct bus_dmadesc **
ioat_copy_8k_aligned(*bus_dmaengine_t dmaengine, bus_addr_t dst1, bus_addr_t dst2, bus_addr_t src1, bus_addr_t src2, bus_dmaengine_callback_t callback_fn, void *callback_arg, uint32_t flags*);

*struct bus_dmadesc **
ioat_copy_crc(*bus_dmaengine_t dmaengine, bus_addr_t dst, bus_addr_t src, bus_size_t len, uint32_t *initialseed, bus_addr_t crcptr, bus_dmaengine_callback_t callback_fn, void *callback_arg, uint32_t flags*);

*struct bus_dmadesc **
ioat_crc(*bus_dmaengine_t dmaengine, bus_addr_t src, bus_size_t len, uint32_t *initialseed, bus_addr_t crcptr, bus_dmaengine_callback_t callback_fn, void *callback_arg, uint32_t flags*);

*struct bus_dmadesc **
ioat_blockfill(*bus_dmaengine_t dmaengine, bus_addr_t dst, uint64_t fillpattern, bus_size_t len, bus_dmaengine_callback_t callback_fn, void *callback_arg, uint32_t flags*);

*struct bus_dmadesc **
ioat_null(*bus_dmaengine_t dmaengine, bus_dmaengine_callback_t callback_fn, void *callback_arg, uint32_t flags*);

DESCRIPTION

The **I/OAT** driver provides a kernel API to a variety of DMA engines on some Intel server platforms.

There is a number of DMA channels per CPU package. (Typically 4 or 8.) Each may be used

independently. Operations on a single channel proceed sequentially.

Blockfill operations can be used to write a 64-bit pattern to memory.

Copy operations can be used to offload memory copies to the DMA engines.

Null operations do nothing, but may be used to test the interrupt and callback mechanism.

All operations can optionally trigger an interrupt at completion with the *DMA_INT_EN* flag. For example, a user might submit multiple operations to the same channel and only enable an interrupt and callback for the last operation.

The hardware can delay and coalesce interrupts on a given channel for a configurable period of time, in microseconds. This may be desired to reduce the processing and interrupt overhead per descriptor, especially for workflows consisting of many small operations. Software can control this on a per-channel basis with the **ioat_set_interrupt_coalesce**() API. The **ioat_get_max_coalesce_period**() API can be used to determine the maximum coalescing period supported by the hardware, in microseconds. Current platforms support up to a 16.383 millisecond coalescing period. Optimal configuration will vary by workflow and desired operation latency.

All operations are safe to use in a non-blocking context with the *DMA_NO_WAIT* flag. (Of course, allocations may fail and operations requested with *DMA_NO_WAIT* may return NULL.)

Operations that depend on the result of prior operations should use *DMA_FENCE*. For example, such a scenario can happen when two related DMA operations are queued. First, a DMA copy to one location (A), followed directly by a DMA copy from A to B. In this scenario, some classes of I/OAT hardware may prefetch A for the second operation before it is written by the first operation. To avoid reading a stale value in sequences of dependent operations, use *DMA_FENCE*.

All operations, as well as **ioat_get_dmaengine**(), can return NULL in special circumstances. For example, if the **I/OAT** driver is being unloaded, or the administrator has induced a hardware reset, or a usage error has resulted in a hardware error state that needs to be recovered from.

It is invalid to attempt to submit new DMA operations in a *bus_dmaengine_callback_t* context.

The CRC operations have three distinct modes. The default mode is to accumulate. By accumulating over multiple descriptors, a user may gather a CRC over several chunks of memory and only write out the result once.

The *DMA_CRC_STORE* flag causes the operation to emit the CRC32C result. If *DMA_CRC_INLINE*

is set, the result is written inline with the destination data (or source in **ioat_crc**() mode). If *DMA_CRC_INLINE* is not set, the result is written to the provided *crcptr*.

Similarly, the *DMA_CRC_TEST* flag causes the operation to compare the CRC32C result to an existing checksum. If *DMA_CRC_INLINE* is set, the result is compared against the inline four bytes trailing the source data. If it is not set, the result is compared against the value pointed to by *crcptr*.

ioat_copy_crc() calculates a CRC32C while copying data. **ioat_crc**() only computes a CRC32C of some data. If the *initialseed* argument to either routine is non-NULL, the CRC32C engine is initialized with the value it points to.

USAGE

A typical user will lookup the DMA engine object for a given channel with **ioat_get_dmaengine**(). When the user wants to offload a copy, they will first **ioat_acquire**() the *bus_dmaengine_t* object for exclusive access to enqueue operations on that channel. Optionally, the user can reserve space by using **ioat_acquire_reserve**() instead. If **ioat_acquire_reserve**() succeeds, there is guaranteed to be room for *N* new operations in the internal ring buffer.

Then, they will submit one or more operations using **ioat_blockfill**(), **ioat_copy**(), **ioat_copy_8k_aligned**(), **ioat_copy_crc**(), **ioat_crc**(), or **ioat_null**(). After queuing one or more individual DMA operations, they will **ioat_release**() the *bus_dmaengine_t* to drop their exclusive access to the channel. The routine they provided for the *callback_fn* argument will be invoked with the provided *callback_arg* when the operation is complete. When they are finished with the *bus_dmaengine_t*, the user should **ioat_put_dmaengine**().

Users MUST NOT block between **ioat_acquire**() and **ioat_release**(). Users SHOULD NOT hold *bus_dmaengine_t* references for a very long time to enable fault recovery and kernel module unload.

For an example of usage, see *src/sys/dev/ioat/ioat_test.c*.

FILES

/dev/ioat_test
: test device for ioatcontrol(8)

SEE ALSO

ioatcontrol(8)

HISTORY

The **I/OAT** driver first appeared in FreeBSD 11.0.

AUTHORS

The **I/OAT** driver was developed by Jim Harris <*jimharris@FreeBSD.org*>,
Carl Delsey <*carl.r.delsey@intel.com*>, and
Conrad Meyer <*cem@FreeBSD.org*>. This manual page was written by
Conrad Meyer <*cem@FreeBSD.org*>.

CAVEATS

Copy operation takes bus addresses as parameters, not virtual addresses.

Buffers for individual copy operations must be physically contiguous.

Copies larger than max transfer size (1MB, but may vary by hardware) are not supported. Future versions will likely support this by breaking up the transfer into smaller sizes.

BUGS

The **I/OAT** driver only supports blockfill, copy, and null operations at this time. The driver does not yet support advanced DMA modes, such as XOR, that some I/OAT devices support.