Physical Address Decoding in Intel Xeon v3/v4 CPUs: A Supplemental Datasheet

Marius Hillenbrand
Karlsruhe Institute of Technology
os@itec.kit.edu

September 5, 2017

The mapping of the physical address space to actual physical locations in DRAM is a complex multistage process on today's systems. Research in domains such as operating systems and system security would benefit from proper documentation of that address translation, yet publicly available datasheets are often incomplete. To spare others the effort of reverse-engineering, we present our insights about the address decoding stages of the Intel Xeon E5 v3 and v4 processors in this report, including the layout and the addresses of all involved configuration registers, as far as we have become aware of them in our experiments. In addition, we present a novel technique for reverse-engineering of interleaving functions by mapping physically present DRAM multiple times into the physical address space.

Contents

1 Introduction

The naive view of physical memory, as a linear address space that represents main memory and memory-mapped devices, is overly simple on today's non-uniform memory access (NUMA) multiprocessor architectures and within complex system-on-chips (SoCs) [2, 1]. In reality, the physical address space is mapped to actual physical locations in several steps. There might not even be a *single* physical address space that is universally used by all CPUs [2].

As main memory is a shared resource, this mapping from physical addresses to the actual physical structure of DRAM has implications for performance and security [3, 13, 14, 15]. Knowledge about that translation process is required to make use of side channels and covert channels based on the operation of DRAM [14]. Further, it can even be used to subvert protection based on virtual memory and to hide rootkits [15]. Research in operating systems and systems security would benefit from a thorough understanding of that address decoding process.

However, for current generation CPUs that decoding process is only sparsely documented, which presents a massive hurdle to researchers by forcing them to rely on reverse-

engineering [14]. For example, the datasheets for the outdated Intel Xeon 5500/7500 generation CPUs offered a detailed description of how a request traveled from the last level caches to DRAM and provided the layouts and addresses of all involved configuration registers (at least for the 5500) [4, 5]. Yet the datasheets for recent Xeon E5 v3 and v4 generations *(Haswell and Broadwell-EP)* lack in these regards [6, 7].

In ongoing research, we have made a considerable effort towards understanding the physical address translation in Intel Xeon E5 v3 and v4 CPUs – which started out as a high-risk and time-intensive endeavor with unclear likelihood of success. We spent countless hours alternatively staring at hex dumps or rearranging memory modules in our test systems. In addition, we employed DRAM mapping aliases [3] as a novel technique for reverse-engineering interleaving functions – regularly locking up our test system because of incomplete knowledge. To spare other interested parties that effort, we decided to share our insights in this document. We hope to hereby encourage and enable future systems and security research based on a near-complete documentation of all involved translation steps and configuration registers.

In this report, we make the following contributions:

- We describe our approach in Section 3. In particular, we discuss how to use DRAM mapping aliases for reverse-engineering DRAM interleaving functions in Section 3.4. We apply this approach to socket interleaving, channel interleaving, and rank interleaving in the Xeon E5 v3 and v4 processors.

- We provide extensive documentation of the address decoding process in recent Intel Xeon E5 processors of the Haswell and Broadwell generations. We give an overview of the decoding process in Section 4. Further, we provide detailed descriptions of the operation and configuration register layout of the source address decoder in Section 5, the target address decoder in Section 6, and the memory mapper in Section 7. We also report the interleaving functions we discovered in those sections.

We discuss limitations of our report and the pitfalls we encountered in Section 8.

2 Background

The Xeon processors of the Haswell and Broadwell generations support non-uniform memory access (NUMA) configurations [10]. In multiprocessor systems, each processor may have memory attached to its local memory controller. The memory controllers built into these processors use DDR3 or DDR4 DRAM as primary main memory.

In Intel *lingua*, all the components of a processor chip outside actual CPU core are referred to as *uncore*. In this report, we are mainly concerned with that *uncore* and discuss addresses and layout of the *uncore configuration registers*.

DDR DRAM systems typically form a hierarchy: Each memory controller interfaces with one or multiple *memory channels*, which are independent buses. Each bus may be populated with memory modules *(dual inline memory modules, DIMMs)*. The DRAM chips on a memory module form groups that can be accessed independently. These groups

are called *rank*. In this report, we only cover address decoding down to the granularity of ranks. For a more thorough discussion of DRAM, please refer to the excellent book by Jacob et al. [9].

3 Approach

We derived all the information presented in this report from publicly available documentation (mainly datasheets from Intel's website) and from what can be observed on a running system (e.g., by retrieving hex dumps of configuration registers from within a running OS). We did not reverse-engineer software (i.e., the UEFI firmware) to avoid legal complications. Further, we did not have access to any documentation about Intel Xeon processors under a non-disclosure agreement.

We followed these steps for discovering configuration registers and their layout:

- First, read all available datasheets for previous CPU and current generations to recognize and exploit similarities. In our case, those included the datasheets for Xeon 5500, 5600, and 7500 generation CPUs, as well as those for Xeon E5 up to v4 and for Xeon E7 up to v3. Notably, the datasheets for the 7500 [5] and the E7 v2 generations [8] briefly describe the address decoding architecture. The datasheets for the 5500 and 5600 series [4] provide a complete description of all relevant configuration registers, including register layouts and addresses, for that generation of processors.

- Second, extract patterns common to all generations. Newer datasheets contain only incomplete lists of configuration registers. Thus, we looked for similar names for the pseudo-PCI devices that map these configuration registers, which then hinted at which registers to look for in each device.

- Finally, discover the configuration registers and their layout in the new generation, guided by what we learned about the previous generations.

For discovering interleaving functions, we used DRAM mapping aliases to map the DRAM both with interleaving and with regions dedicated to each socket/channel/rank. We searched for corresponding address pairs in both mappings and derived the interleaving functions from these pairs.

We introduce our test platforms below and then discuss how we specifically discovered the address decoding architecture, addresses and layouts of configuration registers, and interleaving functions.

3.1 Experimental Platforms

We ran all our experiments on a dual-socket Intel Xeon E5-2630 v3 (Haswell-EP), a single-socket Xeon E5-2618L v3 (Haswell-EP), and a single-socket Intel Xeon E5-2630 v4 (Broadwell-EP) system. We varied the amount and organization of memory during our exploration, as discussed below.

3.2 Address Decoding Architecture

We assumed that newer generations will follow the overarching architecture for physical address decoding as introduced with the Xeon 7500 generation. The concept of source and target address decoders stems from Intel's QuickPath Interconnect (QPI) [11] and likely remains in use because recent generation Xeon CPUs still use QPI as their NUMA interconnect. In addition, the E7 v2 datasheet describes the same addressing architecture and the more recent datasheets suggest no fundamental changes.

3.3 Configuration Registers

We followed a two-step process of first discovering configuration registers and then understanding their layout. We iterated that process for all the stages of the address decoding process, beginning at the target address decoder as an anchor (because the current datasheets document some registers in that stage), then working from there towards the source address decoder, and finally towards the DRAM address decoder.

In both steps, we employed an approach broadly related to differential cryptanalysis against the BIOS firmware's memory configuration code: We varied the number, size, and organization of memory modules (DIMMs) in our test system, booted the system into Linux, and then dumped the contents of regions of memory-mapped configuration registers, searching for correlating changes. The overall address decoding architecture guided what changes we were looking for. As an example, when doubling the amount of memory in the system, we expected the *limit* registers in the source and target address decoders to double. Further, when populating two memory channels instead of one, we expected global interleaving settings to change, as well as the second channel's configuration to switch from *disabled* to an active setting.

As we performed that analysis manually (i.e., by looking at the differences in hex dumps), we aimed to minimize the differences between each pair of configurations to compare. For example, replacing a single dual-rank DIMM with two single-rank DIMMs of half the size each restricted the changes to the memory mapper and allowed us to identify the configuration registers specific to that channel.

We used the dual-socket system for exploring the configuration of NUMA node interleaving. For that purpose, we switched *ACPI NUMA support* on and off in the BIOS setup. With that setting off, the firmware assumes that the OS is not aware of NUMA and interleaves the physical address space between all nodes to avoid unbalanced memory allocation [10].

Using the dual-socket system proved valuable also with the configuration registers local to each NUMA node by both speeding up the exploration process and by making it easier to discover *offset* registers. By populating the memory slots attached to each socket in a different fashion, we were able to explore two configurations for each time-consuming cycle of shutting down, rearranging DIMMs, and rebooting the system. Further, the address decoding scheme uses several address *offsets*, which are hard to discover because they are typically configured to zero in a single-socket system – we cannot tell apart these configuration registers set to *0* from nonexistent registers that also show up as *0* in a

hex dump. Using a dual-socket system trivially avoided that by forcing nonzero offsets in the second socket's configuration registers.

3.4 Interleaving Functions

Interleaving spreads adjacent cache lines across different entities in the DRAM hierarchy, such as channels or ranks. Simple interleaving schemes place subsequent cache lines on subsequent channels/ranks by using address bits for indexing channels/ranks. Other schemes combine several address bits in more complex binary functions, for example using *XOR* to achieve a pseudo-random distribution. Often, these interleaving functions are undocumented, so that interested parties have to rely on reverse-engineering [14, 12].

We propose to utilize DRAM mapping aliases [3] for reverse-engineering interference functions. For that purpose, we map the same physical DRAM twice into the physical address space, once with interleaving and once as a linear mapping, so that we can identify the interleaving functions by finding correspondences between both address mappings.

Linear mapping places the memory of each rank, channel, and NUMA socket as contiguous and distinct region in physical memory. Thus, within that linear mapping, we can unambiguously and trivially identify each physical address with its location in DRAM (i.e., socket, channel, and rank indices). In the interleaved mapping, though, we do not know that relation as we are unaware of the interleaving functions that map a given physical address to a specific socket / channel / rank.

In our approach, we use corresponding pairs of addresses from the linear and the interleaved mapping that refer to the same underlying DRAM. From the address in the linear mapping, we can extract on which socket, channel, or rank that address resides, as that mapping is known. Thereby, we derive the *interleaving index* of the address in the interleaved mapping. Given a sufficiently large set of these address pairs, we can reconstruct the interleaving function.

For searching corresponding address pairs, we rely on the fact that cache lines map to a single access to DRAM (a *burst transfer* on the memory bus). Thus, a data structure that fits within a cache line will be contiguous in both the linear and the interleaved mapping. We use this observation by first placing *marker data structures* in one of the mappings and then searching for these markers in the other mapping. By storing the address where we placed each marker within that marker, we gain a pair of corresponding addresses whenever we find a marker in the second mapping.

In each marker, we store a prefix of 16 random bytes (common to all markers, but chosen anew for each experiment), the physical address where we placed the marker, and a CRC checksum over the random prefix and the address. We align each marker at the start of a cache line. When searching for markers, we can quickly identify potential candidates using the random prefix and the alignmtne to cache lines. Then, we validate the checksums in each candidate to avoid mistaking random data for a marker[1]. Once we found a marker, we add the physical address where we found the marker together with the *placement* address stored within the marker to the set of found pairs.

[1]The remaining chance is negligible.

With this approach, it is irrelevant whether we place the markers in the interleaved mapping and then search them in the linear mapping or vice versa. In any case, we derive the *interleaving index* (e.g., the number of the channel that an address maps to) from the address in the linear mapping and the corresponding physical address from the interleaved mapping.

Finally, we derive the minimum XOR-based function from the *(address, index)-pairs* we gathered, using the same approach that Maurice et al. proposed when reverse-engineering the interleaving functions for Intel's last level cache slices [12]: For each bit of the interleaving function, for each address bit, we check whether we can eliminate that address bit from the function, by

1. set `trials` to 0
2. randomly choose an address from the pairs we gathered,
3. derive a new address by inverting the address bit currently under consideration,
4. searching for the new address in the *(address, index)-pairs*,
5. if not found, increment `trials` and go back to step 2 if `trials` is below a limit,
6. if found, check whether both addresses differ in the index bit currently under consideration
7. if they do not differ, then the current address bit cannot be an input to the interleaving function currently under consideration, so eliminate that address bit from the function,
8. if they do differ, increment `trials` and go back to step 2 if `trials` is below a limit (we aborted each loop after one million trials).

Note that we left Linux unaware of the additional mappings in the physical address space to avoid conflicting allocations. Thus, we had to patch the driver code of Linux's `/dev/mem/` to allow access to any physical address (in function `valid_phys_addr_range` in `drivers/char/mem.c`). As a result, we could simply `mmap` the alternative mappings.

4 Address Decoding Overview

All in all, the address decoding from physical addresses to physical locations in DRAM in Intel Xeon processors in the Haswell and Broadwell generations comprises four steps:

1. Upon a miss in one of the last level caches (LLC), or when an uncacheable memory request is handled by the LLC [8, § 3.1.1], the LLC has to decide where to route a request for that cache line. For that purpose, the *source address decoder (SAD)*, a part of the LLC, compares the request's physical address to a set of configured regions. When the address matches a region, that region defines which NUMA node to route the request to. If desired, the region can be set to *interleave* between multiple NUMA nodes. While the process is similar for DRAM-backed and MMIO regions, we focus on DRAM-backed regions in this report.

2. Once the request arrives at its destination NUMA node, the *target address decoder (TAD)* of that node processes the request's physical address. The TAD matches

that address to its own set of regions. Each region in the TAD defines the target memory channels for matching physical addresses, or a set of target channels for interleaving. The TAD can subtract offsets from the address or perform bit shifts for interleaving. Thereby, the TAD translates from the global physical address space to a *per-channel* address space that is local to the specific memory channel within a specific NUMA node. Each *per-channel* address space ranges from 0 to the capacity of all the memory attached to that channel.

3. For each memory channel, the *memory mapper* matches the request's address to another set of regions to determine which rank to map the request to. Each region can target a single rank or interleave between up to eight ranks. Similar to the TAD, the memory mapper can subtract offsets and perform bit shifts to map the per-channel address space to a local *per-rank* address space.

4. Finally, the memory mapper translates from the *per-rank* address space to the address bits used by the physical organization of the respective rank: It slices the request's address into components that address a single bank, row, and column within the rank.

4.1 Configuration Registers

All the *uncore* configuration registers that we discuss in this report are memory-mapped. Specifically, they are mapped in the *PCI configuration spacE* of pseudo-PCI devices on a dedicated PCI bus for each processor uncore [7, §1.1.2, 6, §1.1.2]. The uncore component called *Processor Configuration Agent (Ubox)* is responsible for implementing these PCI devices and mapping transactions to their configuration space to reads/writes to the uncore configuration registers [8, §9].

Each processor socket has two such dedicated PCI buses for configuration accesses: The first bus provides access to the processor's integrated I/O devices such as the IOMMU or the I/O xAPIC and the second bus offers access to the configuration registers for address decoding (among others), which we focus on in this report. While being freely configurable, the uncore configuration showed up as bus `0xff` in all our single-socket systems and as buses `0x7f` and `0xff` in all our dual-socket systems.

In the following sections, we will refer to PCI devices relative to these buses and only provide PCI device and function numbers.

4.2 Common Patterns

Many decoding stages share common idioms in the layout and the semantics of their configuration registers, which we introduce in this section.

4.2.1 Compact Encoding of Consecutive Regions

All the address decoders only support a set of consecutive regions and encode them in a compact way: The *limit address* of each region implicitly defines the *base address* of

the successive region. The base address of the first region is fixed at 0. In most cases, *limits* are defined at the granularity of 64 MiB. A physical address matches a region if its high-order address bits 26 and above are less than or equal to the *limit* of that region and greater than the *limit* of the preceding region.

Each address decoder offers a fixed number of region registers. As systems typically use fewer regions, superfluous region registers can be marked as unused (e.g., our dual-socket system only uses two out of twenty regions in the SAD). In most cases, regions are marked as unused implicitly by setting their *limit* to the same as that of their predecessor. Some region registers contain an explicit *enable* flag.

4.2.2 Interleave Target List

Interleaving at all levels (sockets, channels, ranks) uses a level of indirection for added flexibility in selecting the target for each physical address. First, the decoding stage calculates an *interleave index* as a function of the address bits. Then, it looks up the destination socket, channel, or rank in a *list of interleave targets* using that index.

5 Source Address Decoder

The source address decoder (SAD) defines the layout of the physical address space for each set of processors that share a last level cache (i.e., a NUMA node). It is responsible for directing memory requests to the NUMA node where the addressed memory cell is locally attached. The SAD can map whole regions to a single NUMA node or interleave regions between several NUMA nodes, which is also called *socket interleaving*[2].

The SAD's configuration registers map to device 15, function 4 `(0f.4)`[3]. We provide an overview of the configuration space of that device in Figure 1.

5.1 SAD Regions

The SAD matches physical addresses against a list of consecutive regions. Each region is encoded in one of the *SAD DRAM region* registers, according to the layout in Register 5.1, following the encoding pattern described in Section 4.2.1. The SAD supports up to 20 regions (as with the 7500 generation [5, §4.4]).

When a request's physical address matches a region, that region defines to which NUMA node the SAD will direct the request. For that purpose, the SAD always treats regions as if they were 8-way *socket interleaved*, which we describe next.

[2]Strictly speaking, the term *socket interleaving* is misleading, because a processor package may occupy one socket but comprise multiple NUMA nodes. We use the term anyway to be consistent with Intel's datasheets.

[3]Linux uses hexadecimal numbers for denominating PCI buses. Thus, we provide device/function specifiers in hex in parenthesis, the same as they appear in, for example, `/proc/bus/pci`.

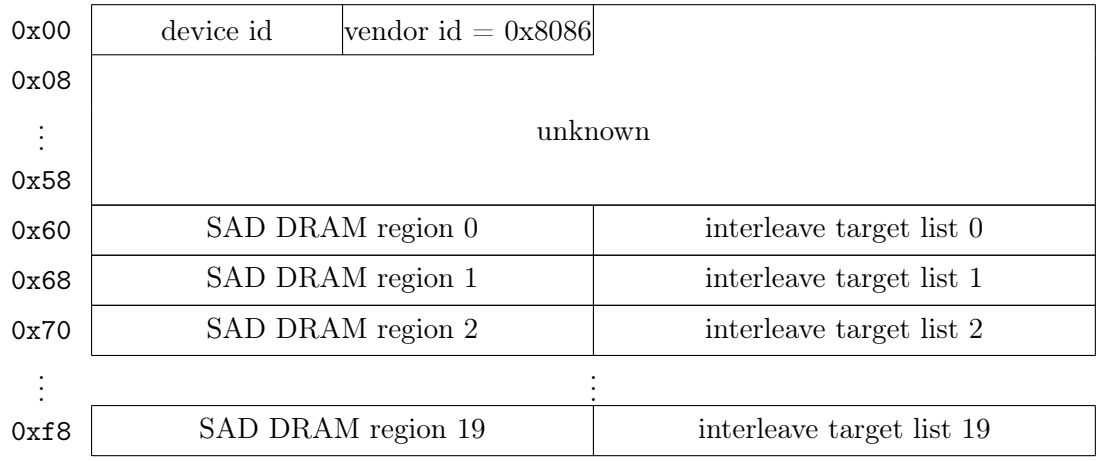
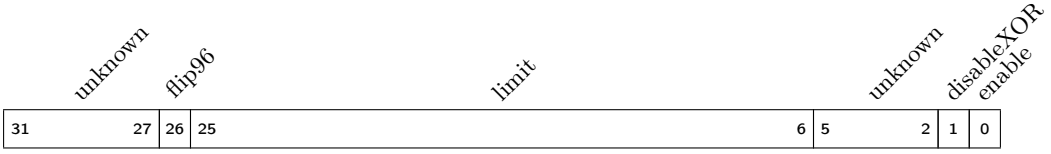


Figure 1: Layout of the Source Address Decoder's PCI configuration space, device 15, function 4 (0f.4). The SAD supports up to 20 different regions. The PCI device id is `0x2ffc` on Haswell and `0x6ffc` on Broadwell-EP.

Register 5.1: Source Address Decoder Region



enable Use this region definition if set.

limit Limit of this region in multiples of 64 MiB.

disableXOR When set, then use bits 8:6 as interleave index. When reset, then XOR 18:16 onto 8:6 to get the interleave index.

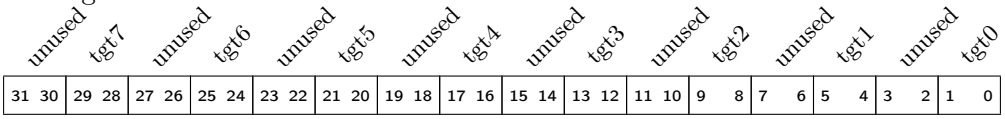
flip96 Use bit 9 instead of bit 6 to calculate the interleave index.

5.2 Socket Interleaving

The SAD can interleave adjacent cache lines between different NUMA nodes. Intel's datasheets use the term *socket interleaving* for that mechanism. When an OS is not aware of the NUMA topology of a system, socket interleaving helps to balance memory requests across all NUMA nodes, because each allocated page frame unavoidably covers memory from all NUMA nodes. When the OS is NUMA-aware, however, the NUMA nodes' memory is typically mapped as dedicated regions to let the OS control on which nodes to allocate each page [10].

Socket interleaving uses the pattern of *interleave target lists*, which we described above in Section 4.2.2: The SAD first maps the physical address of a request to one of eight *interleave indices* and then uses that index to look up the destination NUMA node in a list of interleave targets for that region. We present the layout of the interleave target list that accompanies each SAD region in Register 5.2 (see Figure 1 for the addresses of these registers). This scheme allows all variants from 2-way up to 8-way interleaving by setting several entries of the target list to the same NUMA node. In particular, we are not aware of a flag to *disable* socket interleaving; instead, you set all entries in the target list to the same value to map a region to a single NUMA node.

Register 5.2: Source Address Decoder Interleave Target List



tgtN target NUMA node (or socket) for interleave index *N*.

We are aware of four variants how the SAD calculates the interleave index from the physical address. Two flags in the SAD's region registers (see Register 5.1), which we call `flip96` and `disableXOR`, select the variant:

- With both flags set to 0, the SAD will XOR the address bits 18:16 to the address bits 8:6 to form the interleave index. Specifically, the index bits i_n will be calculated from the physical address bits a_n as follows: $i_2 = a_{18} \oplus a_8$, $i_1 = a_{17} \oplus a_7$, and $i_0 = a_{16} \oplus a_6$.
- When enabling the `disable_XOR` flag, the SAD will only use the lower address bits 8:6 as index.
- When enabling the `flip96` bit, the SAD will use address bit 9 instead of bit 6. That is, $i_0 = a_{16} \oplus a_9$ (when using XOR).

Our results concur with those reported by Pessl et al. on a two-socket system [14] but extend them to the full interleaving function. Pessl et al. found that $a_{17} \oplus a_7$ selected the socket with 2-way socket interleaving, which we identified as i_1. Thus, their system's *interleave target list* was likely set as `(0, 0, 1, 1, 0, 0, 1, 1)`, which matches what we observed when enabling socket interleaving in the BIOS setup on our test system.

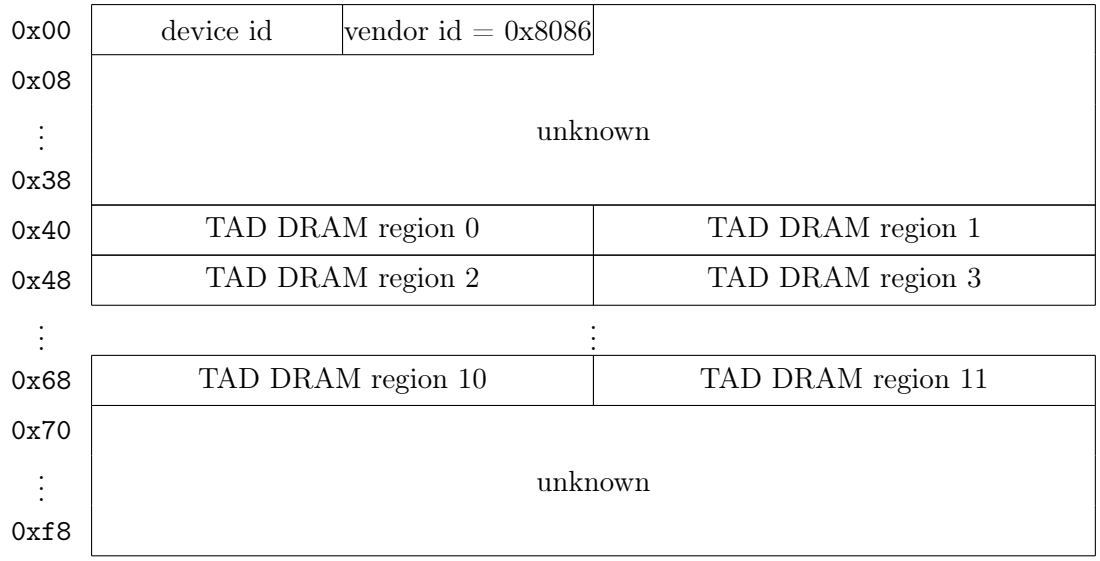


Figure 2: Layout of the home agent's target address decoder PCI configuration space, device 18, function 0 (12.0). The TAD supports up to 12 different regions. The PCI device id is `0x2fa0` on Haswell and `0x6fa0` on Broadwell-EP.

6 Target Address Decoder

At each NUMA node, two components are responsible for handling requests to that node's local memory: The *home agent* handles the cache coherency protocol on the NUMA interconnect and the *integrated memory controller* interfaces with the local DRAM channels. The target address decoder (TAD) appears to be split between these two components. However, from the perspective of address decoding, it does not matter which component performs which part of the address translation. Thus, we will discuss the TAD as if it were a single entity.

The TAD's region configuration is duplicated between home agent and integrated memory controller and appears in two uncore configuration devices: The home agent's TAD configuration resides in device 18, function 0 (see Figure 2), whereas the integrated memory controller's TAD configuration is mapped to device 19, function 0 (see Figure 3). Both sets of registers are independent, yet BIOS firmware appears to set both to the same values.

6.1 TAD Regions

The TAD matches a request's physical address against a list of up to 12 consecutive regions. These regions are configured in the *TAD DRAM region registers* in both the home agent's and the integrated memory controller's TAD configuration spaces (Figures 2 and 3) and share a common layout which we depict in Register 6.1. Like with the SAD regions, each TAD region's base address is implicitly defined by the limit address of the preceding region, as we describe in Section 4.2.1.

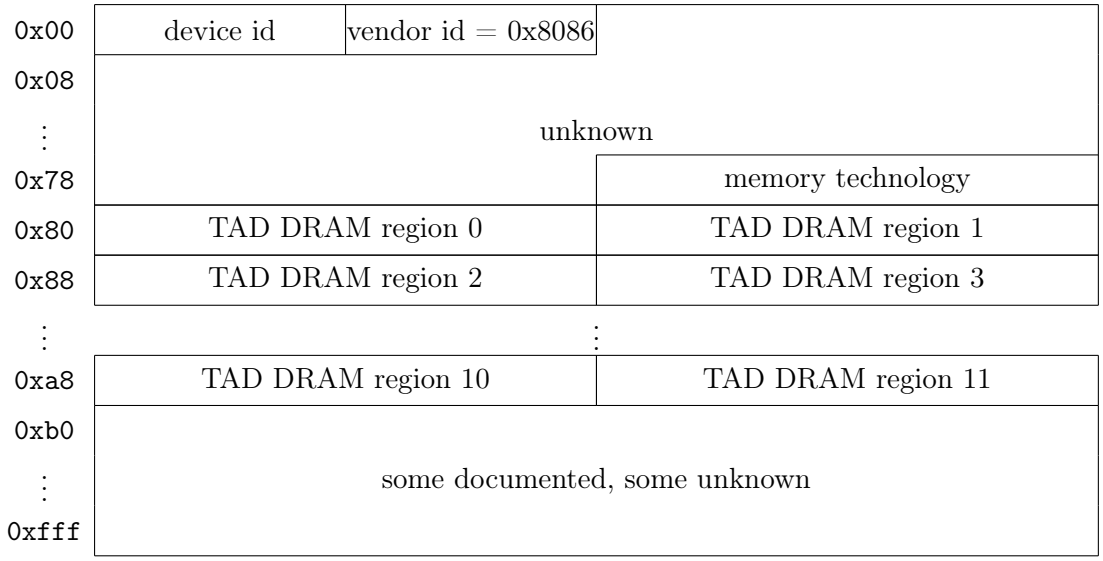


Figure 3: Layout of the integrated memory controller's target address decoder PCI configuration space, device 19, function 0 (13.0). The TAD supports up to 12 different regions. The PCI device id is `0x2fa8` on Haswell and `0x6fa8` on Broadwell-EP. Please refer to the datasheets for documentation of the *memory technology* register [7, §2.1.2, 6, §2.1.2].

The field *socket interleave wayness* instructs the TAD to remove the address bits used in socket interleaving by shifting the physical address to the right.

6.2 Channel Interleaving

When a request's address matches a TAD region, the TAD consults that region's configuration register to determine the memory channel that should handle the request. For that purpose, the field *channel interleave wayness* (see Register 6.1) defines how many channels to use for interleaving in that region – from mapping to a single channel up to 4-way channel interleaving. The TAD first calculates a 2-bit interleave index (i_1, i_0) by XOR-ing several address bits a_i according to the following function:

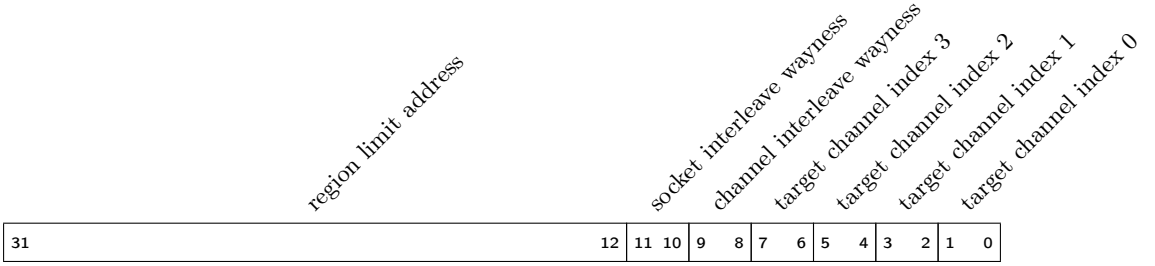
$i_0 = a_7 \oplus a_{12} \oplus a_{14} \oplus a_{16} \oplus a_{18} \oplus a_{20} \oplus a_{22} \oplus a_{24} \oplus a_{26}$

$i_1 = a_8 \oplus a_{13} \oplus a_{15} \oplus a_{17} \oplus a_{19} \oplus a_{21} \oplus a_{23} \oplus a_{25} \oplus a_{27}$

Then, the TAD looks up the destination channel in the *interleave target* fields in the region's configuration register, using up to 2, 3, or 4 of the target fields for 2-way, 3-way, or 4-way interleaving, respectively. When mapping the whole region to a single channel, the field *target channel index 0* denotes that channel.

Before forwarding a request's address to the memory mapper, the TAD will subtract an offset from that address and perform a bit shift to remove the bits used in interleaving, as necessary. The fields *socket interleave wayness* and *channel interleave wayness* in the region's configuration register determines how many bits to remove. For 3-way channel

Register 6.1: Target Address Decoder Region



region limit address — Limit address of this region in multiples of 64 MiB. Matched against the high-order address bits 26 and above.

socket interleave wayness — Degree of NUMA interleaving (0: no interleaving, 1: 2-way, 2: 4-way, 3: 8-way), controls how many bits to remove from a physical address.

channel interleave wayness — Degree of interleaving across memory channels (0: 1 channel, 1: 2 channels, 2: 3 channels, 3: 4 channels), defines how many bits to use as interleaving index and how many bits to remove from a physical address.

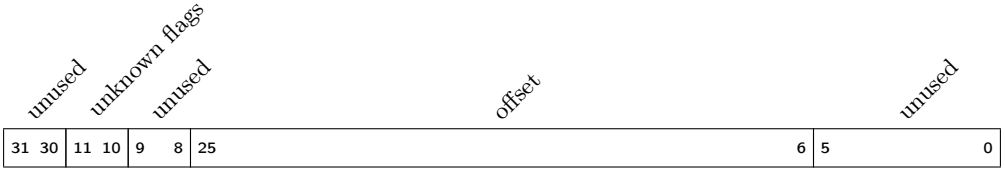
target channel index 3..0 — List of target channels. The interleaving index selects which entry to use.

interleaving, the TAD will divide the address by three instead of bit shifting it to the right.

The offset enables to use channel interleaving when channels are populated with unequal capacity. An offset can be set specific per channel and per region using registers in the per-channel configuration spaces of the integrated memory controller, see Figure 4. The registers follow the layout depicted in Register 6.2.

Keep in mind that the TAD first subtracts the offset from an address and then performs the bit shift to remove interleaving bits.

Register 6.2: Target Address Decoder Region Per-Channel Offset



unknown flags Appear to control address calculation with 3-way channel interleaving.

offset Offset to subtract from the physical address when matching a specific region and directing a request to a specific channel, in multiples of 64 MiB.

7 Memory Mapper

The *memory mapper* performs the final two translation steps from per-channel addresses to physical locations in DRAM. The first step maps the address space of the channel to the DRAM ranks attached to that channel, translating from per-channel addresses to per-rank addresses. The second step maps the linear per-rank address spaces to the actual DRAM addressing signals used on the memory bus. For that purpose, it splits the address into components that address a bank, row, and column, according to the physical organization of the rank.

In this report, we focus on the address decoding down to the granularity of ranks, that is, down to the first translation step performed by the memory mapper. Information about the mapping functions for individual banks have been reported by Pessl et al. [14].

The memory mapper uses a distinct set of configuration registers for each channel. These configuration registers share PCI devices with the channel-specific settings of the TAD, see Figure 4.

7.1 Rank (Interleaving) Regions

The memory mapper matches each per-channel address to a list of up to five contiguous *rank interleaving regions*. These regions follow the pattern of storing only the limit

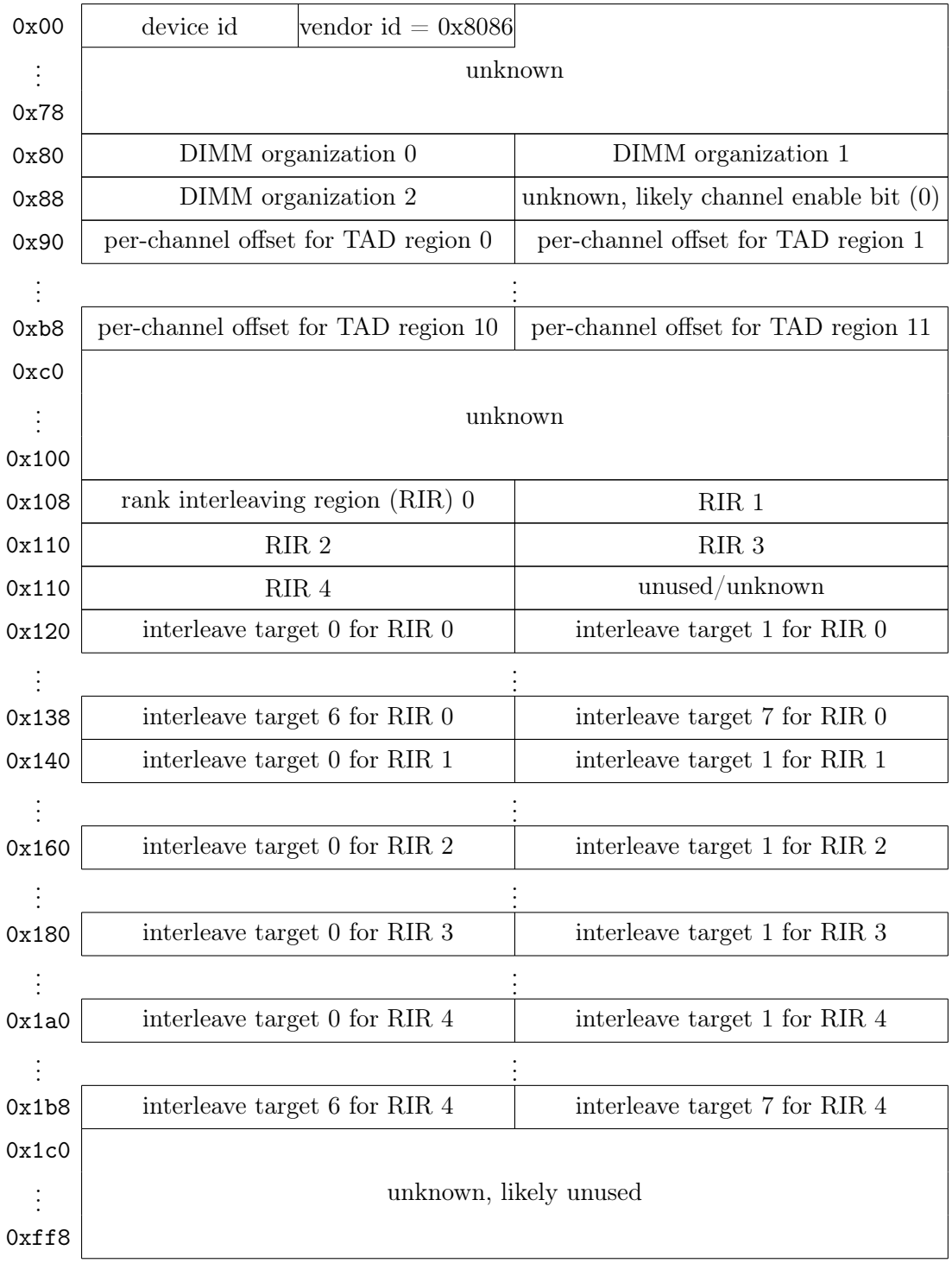
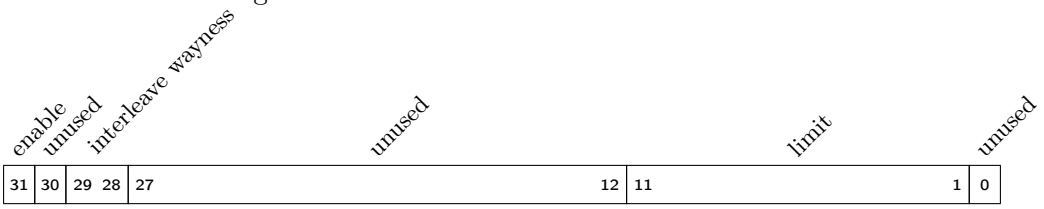


Figure 4: Layout of the integrated memory controller's per-channel TAD and rank interleaving configuration spaces, device 19, functions 2-5 for channels 0-3 (13.2-13.5, with device ids from `0x2faa` to `0x2fad`). See datasheets for the DIMM organization registers [7, §2.3.2, 6, §2.3.2]. The PCI device ids range from `0x2faa` to `0x2fad` on Haswell and from `0x6faa` to `0x6fad` on Broadwell-EP.

address of each region, which implicitly defines the base address of the successive region (see Section 4.2.1). Each region is configured according to Register 7.1.

Register 7.1: Rank Interleaving Region



enable Use this region definition if set.

interleave wayness How many ranks to interleave in this region (0: single rank, 1: 2-way, 2: 4-way, 3: 8-way).

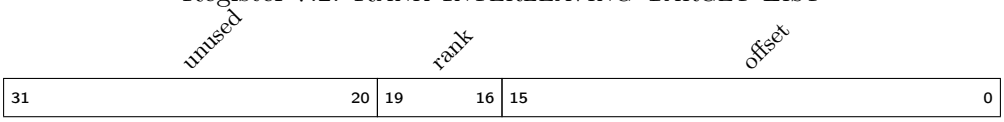
limit Limit of this region in multiples of 512 MiB.

7.2 Rank Interleaving

The memory mapper supports rank interleaving across two, four, and eight ranks, as well as mapping a region to a single rank. The number of ranks to interleave is defined in the registers that configure the regions (see the field *interleave wayness* in Register 7.1).

The memory mapper employs the pattern of *interleave target lists*, which we described above in Section 4.2.2, and directly uses address bits 13 to 15 as the 3-bit interleaving index. For each possible rank interleaving region, the interleaving target list comprises eight registers (i.e., one per target), see Figure 4, offsets `0x120` and following. Each of these registers uses the layout depicted in Register 7.2.

Register 7.2: Rank Interleaving Target List



rank Rank encoded as 2-bit DIMM number and 2-bit rank index within a DIMM.

offset Offset (in multiples of 16 MiB) to subtract after removing interleave bits.

After extracting the interleaving index from a request's address, the memory mapper removes these bits from the address by shifting the higher order address bits right. Then, it subtracts the offset specific to this interleaving target.

The offset is useful for mixed configurations in a channel. Imagine a memory channel populated with three ranks (e.g., with three single-rank DIMMs) that uses a region with

two-way rank interleaving for two of the ranks and another region that maps the third rank without interleaving. The offset field in the second region, first interleave target will be set to the length of the first region (*no* interleaving means just a single target).

In contrast to socket and channel interleaving, the interleaving function for ranks is relatively simple. We are not aware of any flags that change this function.

Note that we used address bits as seen by the memory mapper in the discussion above. When using socket and/or channel interleaving, a physical address is shifted to the right before being handed over to the memory mapper. Thus, the rank interleaving bits can appear in the physical address at higher bits than 13 to 15. Keeping that in mind, our results confirm the rank address mapping that Pessl et al. reported for a dual Haswell-EP system [14]. They found that bit 15 of the physical address chooses between two ranks on each of two channels, which aligns with bit 13 of the per-channel address after shifting the address right by two bits for 2-way channel and 2-way socket interleaving.

8 Limitations & Pitfalls

In this section, we list limitations of our exploration, such as configuration options we deliberately omitted, and pitfalls we encountered and circumvented.

8.1 Processors With Two Memory Controllers

Some Xeon E5 processors feature two integrated memory controllers and home agents. The datasheets suggest that the seconary devices mirror the configuration registers at different PCI config spaces [6, §1.1.2]. We cannot confirm that assumption because we do not have a processor model with two memory controllers available.

8.2 3-Way Channel Interleaving

We deliberately exclude 3-way channel interleaving from this report. While apparently supported by the TAD, we found no benefit in tackling the additional complexity. We typically populate all four channels anyway.

8.3 Interaction of Socket and Channel Interleaving

When reverse-engineering interleaving functions, we considered interleaving at each address decoding step in isolation – that is, we analyzed channel interleaving without socket interleaving and rank interleaving without socket or channel interleaving. Comparing our channel interleaving function with that reported by Pessl et al. [14], we have to assume that the combination of socket and channel interleaving results in a different channel interleaving function than what we observed.

8.4 Open Page / Closed Page Mapping

The memory mapper supports two address mapping schemes, referred to as *open page* and *closed page* address mapping scheme (see register *mcmtr* in [7, §2.1.2]). Jacob provides

an overview of *row-buffer-management policies* and the related address mapping schemes in [9, §13.2]. In this report, we only present results for the *open page* mapping scheme, which is chosen by the BIOS firmware in all our test systems. The datasheet from the 5500 generation processors suggest that the rank interleaving bits differ between the two addressing schemes [4, §2.17.2].

We left out an exploration of the *closed page* mapping scheme because this setting is global for each memory controller and thus cannot be changed without disturbing a running system: Allocated data structures and page frames in physical memory would suddenly appear at different physical addresses and thereby crash any running system. In contrast, our approach establishes additional regions in the physical address space and leaves the existing mapping untouched so that a running operating system and its existing allocations remain undisturbed. As a consequence, our approach cannot be applied for global settings such as the DRAM address mapping scheme.

8.5 Configuration Space Access

On some of our systems, we found that not all the TAD registers were visible when accessing them via the `/proc/bus/pci` interface in Linux. The *uncore* exposes its configurations registers in the configuration spaces of pseudo-PCI devices. Linux relies on fields in the PCI configuration space to differentiate between conventional PCI and PCI-X or PCI express devices and, consequently, to determine the size of these configuration spaces (256 bytes for conventional devices versus 4K for some PCI-X and all PCI express devices). As the configuration spaces of the uncore PCI devices did not fully conform to the PCI specs on some of our systems, Linux would falsely guess the size of these configuration spaces as 256 bytes. Consequently, Linux will only expose the lower 256 bytes of the configuration space in `/proc/bus/pci/<device>` and thereby hide the registers that are mapped to the extended configuration area above that limit. To avoid that, we modified the function `pci_cfg_space_size` in the Linux kernel's source file `drivers/pci/probe.c` to use the extended configuration space for the uncore configuration devices we discuss in this report.

8.6 Live Reconfiguration

In our experiments, we successfully reconfigured the address decoding registers in a running system for regions of physical memory not currently in use. When accessing incorrectly configured regions, the SAD and TAD behaved gracefully and just aborted memory requests. However, we observed lockups of our test systems when we left the configuration of the memory mapper in an inconsistent or incorrect state.

8.7 Locking Configuration Registers

The real value of understanding the uncore configuration registers is being able to adjust them for a research prototype. To be practical, that means writing new settings to the configuration registers after an unmodified firmware established a commodity configuration. Unfortunately, firmware can lock the configuration registers of all address decoding

stages against modifications and there is no public documentation on that locking mechanism besides its existence [7, 6, §1.3 RW-LB]. YMMV. Our experience is mixed: On some of our test systems, the configuration registers remain writable, on others the firmware apparently locked them against modification at runtime.

9 Summary

The decoding of *physical addresses* to actually physical locations in DRAM is a complex multi-step process in current server-class processors, driven by the requirements to support NUMA architectures and the variable and asymmetric population of memory channels and NUMA nodes with DRAM. In this report, we describe that decoding process as implemented in the Intel Xeon E5 v3 and v4 processors. We provide the layouts and locations of configuration registers that have an essential influence on the decoding process but are left out from publicly available datasheets.

By utilizing *DRAM mapping aliases* for reverse-engineering, we derived the interleaving functions used by these processors for socket interleaving, channel interleaving, and rank interleaving. Our findings confirm previously published results. To the best of our knowledge, we extend on previous results because our approach permits us to discover the full interleaving functions (e.g., for eight-way socket or rank interleaving) while previous work is limited to the DRAM organization physically present in the systems under test (e.g., only two sockets or ranks).

References

[1] Reto Achermann et al. "Formalizing Memory Accesses and Interrupts." In: *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*. MARS 2017. Upsala, Sweden: Open Publishing Association, 2017, pp. 66–116. DOI: 10.4204/EPTCS.244.4[4]. URL: http://eptcs.web.cse.unsw.edu.au/paper.cgi?MARS2017.4.

[2] Simon Gerber et al. "Not Your Parents' Physical Address Space." In: *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*. HOTOS'15. Switzerland: USENIX Association, May 2015.

[3] Marius Hillenbrand et al. "Multiple Physical Mappings: Dynamic DRAM Channel Sharing and Partitioning." In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: ACM, 2017, 21:1–21:9. ISBN: 978-1-4503-5197-3. DOI: 10.1145/3124680.3124742[5]. URL: http://doi.acm.org/10.1145/3124680.3124742.

[4] *Intel Xeon Processor 5500 Series Datasheet, Volume 2*. 321322-002. Intel Corporation. Apr. 2009.

[4] https://doi.org/10.4204/EPTCS.244.4

[5] https://doi.org/10.1145/3124680.3124742

[5] *Intel Xeon Processor 7500 Series Datasheet, Volume 2*. 323341-001. Intel Corporation. Mar. 2010.

[6] *Intel Xeon Processor E5 v4 Product Family Datasheet*. 333810-002US. Intel Corporation. June 2016.

[7] *Intel Xeon Processor E5-1600/2400/2600/4600 v3 Product Families Datasheet*. 330784-003. Intel Corporation. June 2015.

[8] *Intel Xeon Processor E7 v2 2800/4800/8800 Product Family Datasheet*. 329595-002. Intel Corporation. Mar. 2014.

[9] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123797519, 9780123797513.

[10] Christoph Lameter. "NUMA (Non-Uniform Memory Access): An Overview." In: *Queue* 11.7 (July 2013), 40:40–40:51. ISSN: 1542-7730. DOI: 10.1145/2508834.2513149[6]. URL: http://doi.acm.org/10.1145/2508834.2513149.

[11] R.A. Maddox, G. Singh, and R.J. Safranek. *Weaving High Performance Multiprocessor Fabric: Architectural Insights Into the Intel QuickPath Interconnect*. Books by Engineers, for Engineers. Intel Press, 2009. ISBN: 9781934053188.

[12] Clémentine Maurice et al. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters." In: *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*. RAID 2015. Kyoto, Japan: Springer-Verlag New York, Inc., 2015, pp. 48–65. ISBN: 978-3-319-26361-8. DOI: 10.1007/978-3-319-26362-5_3[7]. URL: http://dx.doi.org/10.1007/978-3-319-26362-5_3.

[13] Onur Mutlu and Lavanya Subramanian. "Research Problems and Opportunities in Memory Systems." In: *Supercomputing Frontiers and Innovations: an International Journal* 1.3 (Oct. 2014), pp. 19–55. ISSN: 2409-6008. DOI: 10.14529/jsfi140302[8]. URL: http://dx.doi.org/10.14529/jsfi140302.

[14] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 565–581. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[15] Wonjun Song et al. "PIkit: A New Kernel-Independent Processor-Interconnect Rootkit." In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 37–51. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/song.

[6] https://doi.org/10.1145/2508834.2513149
[7] https://doi.org/10.1007/978-3-319-26362-5_3
[8] https://doi.org/10.14529/jsfi140302