ACELab Technical Report TR-2021-03

Mapping Addresses to L3/CHA Slices in Intel Processors

Document Revision 1.00
July 26, 2021, revised to 2021-09-10
Status: Initial Release

John D. McCalpin
mccalpin@tacc.utexas.edu
Advanced Computing Evaluation Laboratory
Texas Advanced Computing Center
The University of Texas at Austin
www.tacc.utexas.edu

Mapping Addresses to L3/CHA Slices in Intel Processors

John D. McCalpin mccalpin@tacc.utexas.edu
Texas Advanced Computing Center
The University of Texas at Austin

Abstract: The distributed, shared L3 caches in Intel multicore processors are composed of "slices" (typically one "slice" per core), each assigned responsibility for a fraction of the address space. A high degree of interleaving of consecutive cache lines across the slices provides the appearance of a single cache resource shared by all cores. A family of undocumented hash functions is used to distribute addresses to slices, with different hash functions required for different numbers of slices. In all systems studied to date, the hash consists of a relatively short (16 to 16384 elements) "base sequence" of slice numbers, which is repeated with binary permutations for consecutive blocks of memory. The specific binary permutation used is selected by XOR-reductions of different subsets of the higher-order address bits. This report provides the base sequences and permutation select masks for Intel Xeon Scalable Processors (1st and 2nd generation) with 14, 16, 18, 20, 22, 24, 26, 28 slices, for 3rd Generation Intel Xeon Scalable Processors with 28 slices, and for Xeon Phi x200 processors with 38 slices.

1. Introduction

Since the release of the quad-core "Nehalem EP" (Xeon 5500 series) processors in 2009, all of Intel's "mainstream" Xeon processors have supported an L3 cache shared by all cores on the chip. In the Nehalem EP and Westmere EP (6 core), the hardware performance counter interface presents the image of a monolithic L3 resource. By the time Intel reached 8 cores with "Sandy Bridge EP" (Xeon E5) in 2012, the bandwidth and access rate required for an effective L3 cache could not be satisfied by a monolithic resource, and the L3 was explicitly split into eight "slices" (one per core), with each "slice" responsible for caching 1/8th of the cache line addresses. To spread the accesses across the slices as uniformly as possible, each aligned block of 8 consecutive cache lines is assigned to a permutation of the 8 L3 slices using an undocumented hash function. The L1 and L2 caches remain private to the cores, and only on an L2 cache miss does the processor have to compute the hash function to send the request to the proper L3 slice.

For power-of-two L3 slice counts it is straightforward to generate families of hash functions that can be computed quickly, and which can be "tuned" to satisfy second-order performance requirements. The former is important because the computation of the hash function is in the critical path for L3 accesses. Second-order performance requirements might include a desire to minimize conflicts for some set of important strides, or to ensure that the first cache line of any 4KiB page has an equal probability of being mapped to any of the slices.

For non-power-of-two L3 slice counts it is only modestly more difficult to design a low-cost hash function with desired characteristics – if you only need to support one slice count. Foreseeing the challenges of supporting many non-power-of-two slice counts, Intel developed a general set of hashing functions that have been applied to all processors from the Sandy Bridge EP to the recent 3rd generation Xeon Scalable Processors.

This report[1] presents a methodology for inverting the address-to-slice hash used in many recent Intel microprocessors. Results are presented for most "large" configurations (14 or more cores) of Intel Xeon Scalable Processors (1st and 2nd generations), for the Xeon Phi 7250, and for one 3rd-generation ("Ice Lake") Xeon Scalable Processor configuration (the 28-core Xeon Gold 6330).

[1] Portions of this material were presented at the IXPUG Fall Conference on 2018-09-25, available at http://dx.doi.org/10.26153/tsw/13161.

2. Background and Related Work

2.1. Motivation and Nomenclature

For Intel processors of the Xeon E5 v1/v2/v3/v4 lines ("Sandy Bridge EP", "Ivy Bridge EP", "Haswell EP", "Broadwell EP"), the L3 cache is shared, distributed, and *inclusive*[2]. In this context, "inclusive" means that all addresses that are cached in any of the private L1 and/or L2 caches of the cores must also have a valid entry in the L3 cache. This enables the hardware to check in a single location (an L3 slice) for an address and be guaranteed that its absence there ensures that it is not present in any L1 or L2 cache. Without this property, all L1 and L2 caches would have to be probed/snooped on each L3 miss, and this becomes impractical for more than about four L3 slices. To enforce inclusion, for any line chosen to be victimized from the L3 cache, the hardware invalidates the line from all private L1 and L2 caches on the chip before it is invalidated in the L3 cache. Since the L3 cache is shared by all cores, this means that it is possible for cores to cause cache lines to be invalidated from other core's *private* caches without accessing the same addresses – they simply need to access enough addresses that map to the same "set" in the L3 cache to overflow that set's associativity. This loophole in the definition of "private" can result in unexpected cache misses, lowering performance [1] and opening security holes [2].

Starting with the Xeon Scalable Processors, the L3 cache is shared, distributed, and (at least mostly) *exclusive*. Addresses that are cached in the private L1 and L2 caches are not cached in the L3. Instead, lines that are chosen as victims in the private L2 caches are sent to the L3 to be available for re-use. Limiting probes/snoops of the L1 and L2 caches is still required, but the L3 no longer provides the required information. Instead, these processors add a new facility called a "Snoop Filter" (SF) that tracks the addresses cached in the L1 and L2 caches. In effect, the Snoop Filter acts as the *cache tags* of an inclusive L3 cache but does not contain space to cache the actual data. Like the inclusive L3 cache in earlier processors, the Snoop Filter is *inclusive* of the private caches and cache lines must be evicted from all private caches before the corresponding Snoop Filter entry can be invalidated. In these processors, the Snoop Filter forms part of the "Caching and Home Agent" (CHA), which is shared, distributed, and co-located with the L3 cache slices. This combination of CHA/SF/L3 is responsible for coherence processing and L3 caching for the chip.

The Xeon Phi x200 processors ("Knights Landing") are slightly different. These processors do not support an L3 cache, but still require inclusive Snoop Filters to avoid excessive snooping of the private caches. As in the Xeon Scalable Processors, these Snoop Filters are part of the shared, distributed CHA.

For the remainder of this report, any of the terms related to L3, CHA, SF, or "slice" are effectively synonyms – referring to the distributed caching and coherence functionality of this cluster of units.

2.2. Inversion of the Address-to-Slice Hash

Perhaps the first paper to invert the address-to-L3-slice hash for an Intel processor was [3], which looked at L3 conflicts to derive sets of addresses that mapped to the same L3 slice in a 4-slice processor. The structure of the formulation (which also appears in all subsequent work) is based on a "binary permutation" operator, with the specific permutation number selected by XOR-reduction of different subsets of the upper address bits[3].

[2] The scope of sharing, distribution, and inclusivity is almost always one processor chip in these systems and will be assumed to be so throughout.

[3] The earliest papers assigned addresses to groups based on their L3 cache conflict behavior, and so were unable determine the actual L3 slice numbers – they just grouped addresses into a set of bins that map 1:1 onto the L3 slices. Later studies have been able to map lines to the hardware provided L3 slice numbers.

Definition: The "**binary permutation**" operator is a simple permutation operator that can be implemented very cheaply in hardware or software.
For an input sequence a with elements $a_i : 0 \le i < 2^n$, applying binary permutation p $(0 \le p < 2^n)$ generates the permuted output sequence b with elements $b_i = a_{i \oplus p} : 0 \le i < 2^n$.
Here (and throughout), $\oplus$ is the binary exclusive-OR operator.
What do these permutations do to the sequence?

- Binary permutation 0 is the identity operator.
- Binary permutation 1 swaps elements in every even/odd pair.
- Binary permutation 2^{n-1} swaps the first and last halves of the sequence.
- Binary permutation $2^n - 1$ reverses the sequence.

Definition: The "**XOR reduction**" operator is the generalization of the 2-input binary exclusive-OR ("XOR") to sequences. The XOR reduction operator returns the "parity" of a bit sequence – *i.e.*, 0 if the number of bits set in the input sequence is even and 1 if the number of bits set is odd.
The operator is typically applied to an explicitly indexed vector of 1-bit values. If the argument is a multi-bit number (or an expression evaluating to a multi-bit number), the operator is applied to the bit vector implied by the binary representation of that number.

Using the results of that initial paper as an example, three equivalent representations of the XOR reduction equations for this 4-slice processor are:

1. List the physical address bits involved in each XOR reduction:

$$p_1 = i_{31} \oplus i_{30} \oplus i_{29} \oplus i_{27} \oplus i_{25} \oplus i_{23} \oplus i_{21} \oplus i_{19} \oplus i_{18}$$
$$p_2 = i_{31} \oplus i_{29} \oplus i_{28} \oplus i_{26} \oplus i_{24} \oplus i_{23} \oplus i_{22} \oplus i_{21} \oplus i_{20} \oplus i_{19} \oplus i_{17}$$

2. Highlight active address bits in each XOR reduction (implying the equations above)

address bit -->	*i31*	*i30*	*i29*	*i28*	*i27*	*i26*	*i25*	*i24*	*i23*	*i22*	*i21*	*i20*	*i19*	*i18*	*i17*
M1	1	1	1	0	1	0	1	0	1	0	1	0	1	1	0
M2	1	0	1	1	0	1	0	1	1	1	1	1	1	0	1

3. Combine the active address bits into a set of "permutation selector masks" that select address bits that participate in each XOR reduction. The masks equivalent to the two examples above are:
   - M_1 = 0xEAAC0000
   - M_2 = 0xB5FA0000

Version (b) typically provides the greatest intuition, but version (c) allows for a much simpler software implementation (such as the C language function in Figure 1) and a more concise representation.

```
int compute_perm(long addr, long *SelectorMasks)
{
    long i,j,k;
    int computed_perm = 0;
    for (int bit=0; bit<sizeof(SelectorMasks); bit++) {
        k = SelectorMasks[bit] & addr; // bitwise AND with mask
        j = __builtin_popcountl(k);    // count number of bits set
        i = j%2;                       // compute parity
        computed_perm += (i<<bit);     // scale and accumulate
    }
    return (computed_perm);
}
```

Figure 1: A function for computing the permutation number associated with an address using a set of permutation selector masks.

Subsequent research [4] used similar methodologies to present XOR selector patterns for 2, 4, and 8 L3 slices, and showed that the selectors were the same across processor generations (Sandy Bridge, Ivy Bridge, Haswell) at these power-of-two slice counts.

Yarom, et al. [5] removed some of the ambiguity from the earlier works by labeling the L3 slices (in this case according to the co-located core as determined by latency measurements[4]), and greatly extended earlier work by deriving a compact representation of the nonlinear part of the hash function required for systems with non-power-of-2 slice counts. This was soon followed by publication of results using hardware performance counters to identify the slice numbers[8] in processors with 2, 4, 6, and 8 slices.

More recently, a complete formulation of the address hash function for the Intel Xeon Phi x200 processors has been reported [9][10]. That formulation is not posed in terms of the same structure as the previous work discussed in this section but has been transformed into a compatible representation for this report.

3. Methodology

3.1. Measurements

Identification of the L3/CHA slice responsible for each physical address is a straightforward exercise using the L3/CHA hardware performance counters in the "uncore" of the Intel processor chips [11].

- Allocate and zero a large block of memory on a 2MiB boundary with Transparent Huge Pages enabled.
- Program one L3/CHA performance counter in each CHA to measure LLC_LOOKUPS.READ
- For each 2MiB-aligned sub-block of the array:
    - Get the physical address of the base of the 2MiB-aligned segment (using `/proc/self/pagemap`)
    - Has this 2MiB page already been mapped?
        - Yes: skip to next 2MiB virtual address in array
        - No: Continue with Mapping
    - Mapping: For each cache line in the 2MiB region:
        - Read the LLC_LOOKUPS.READ counter in each CHA
        - Repeat 1000x: (read address, MFENCE, flush address, MFENCE)
        - Read the LLC_LOOKUPS.READ counter in each CHA
        - Perform "sanity checking" on results
            - Pass: record the matching CHA number and move to next cache line, else
            - Fail: Repeat measurements for current cache line
    - After completing the mapping of the 32768 cache lines in one 2MiB page, save the results (32768 one-byte values), with the physical address of the base of the 2MiB page in hexadecimal as part of the file name.

The entire program was run repeatedly so that it would be allocated as many different 2MiB pages as possible, continuing until the collection of results showed adequate coverage of the physical address bits of interest.

The "sanity checking" for these measurements consists of a variety of *ad hoc* tests looking to see if any fail.

- Compute min, avg, max delta LLC_LOOKUPS.READ over the CHAs and compute 3 tests:
    - If (max/1000 > 0.95): g_1 = pass; else g_1 = fail
    - If (min/1000 < 0.20): g_2 = pass; else g_2 = fail
    - If (avg/1000 < 0.40): g_3 = pass; else g_3 = fail
- Check to see how many CHAs report >0.95 of the expected value
    - If 0: g_4 = fail (equivalent to g_1)

[4] Note that the L3/CHA slice numbers from the hardware performance counters do not match the numbers of the co-located cores in most processor models. See [6][7] for review and discussion of the mappings in a variety of Intel processors.

  - If 1: g_4 = pass
  - If >1: g_4 = fail
- If g_1, g_2, g_3, g_4 all pass, then accept results, otherwise immediately repeat testing on this address
- After every 100 fails, sleep for 1 second.
- After 10 sleeps, abort the program.

Except on the Xeon Phi 7250 systems, the program typically repeats less than 1% of the measurements and very rarely requires the one-second "sleep" delay. Subsequent analysis showed that these *ad hoc* rules resulted in zero errors in more than 3 billion measurements (over 100000 2MiB pages).

The Xeon Phi systems tested had too much "noise" in the measurements to obtain reliable results in a practical time frame, so the results here were derived from the full mapping presented in [9][10], but re-cast to be consistent with the formulation used here for the Xeon Scalable Processors. A small set of mappings (28 2MiB pages) were collected in September 2018 on TACC systems. These did not provide enough data on their own to allow inversion of the address mapping function, but were in perfect agreement with the results of [9][10].

A summary of the systems tested and the mapping data gathered is presented in Table 1.

Code Name	L3/CHA slices	Model(s) tested	2MiB pages mapped
KNL	38	Xeon Phi 7250	28
SKX	14	Xeon Gold 6132 & Gold 5120	22097
SKX & CLX	16	Xeon Gold 6142 & Silver 4216	756
SKX	18	Xeon Gold 6150	268
SKX	20	Xeon Gold 6148	3187
SKX	22	Xeon Gold 6152	26422
SKX	24	Xeon Platinum 8160	33028
SKX & CLX	26	Xeon Platinum 8170 & 8260	1018
SKX & CLX	28	Xeon Platinum 8180 & 8280	7859
ICX	28	Xeon Gold 6330	5600

Table 1: Summary of address to L3/CHA mapping data collected. "KNL" is "Knights Landing" (Xeon Phi x200). "SKX" is "Skylake Xeon" (1st generation Xeon Scalable Processor). "CLX" is "Cascade Lake Xeon" (2nd generation Xeon Scalable Processor). "ICX" is "Ice Lake Xeon" (3rd generation Xeon Scalable Processor). No differences were found between SKX and CLX processors with the same number of L3 slices.

3.2. **Analytical Formulation**

The analytical formulation used to represent these address hashing functions is a slight simplification of that used in prior publications[5], particularly [5]. For each of the systems tested, the data showed the following properties:

- The sequence of L3/CHA numbers in aligned, power-of-two block sizes shows a relatively small number of unique patterns.
- For a suitable choice of the power-of-two block size, all observed sequences of L3/CHA numbers are "binary permutations" of each other.

[5] See Appendix A for a detailed description of the differences between the formulations.

- In each case, it is possible to derive a "base sequence" that applies to the block starting at address zero, and to compute the binary permutation number of all subsequent blocks using XOR-reduction of subsets of the high-order address bits.

These observations and definitions lead to a simple structure that will be used here to encapsulate the mapping from physical address to L3/CHA slice number.

Some nomenclature conventions:

- "*m*" is the number of bits in sequence length (*i.e.*, SeqLength = 2^m)
- The physical address *A* can be divided into three parts:

bits >(*m*+5)	bits (*m*+5):6	bits 5:0
"sequence number"	**"index"**	(Unused here)
Used to compute binary permutation number	Cache line number inside sequence – number that gets permuted	(Inside cache line)

- "*M*" is the set of *m* permutation selector masks (introduced in Section 2)
  - "*M(A)*" means "applying" the vector of "*m*" selector masks to an address to obtain a composite binary permutation number "*p*"
  - Figure 1 provides a concrete implementation of the meaning of this operator
- "*perm*" is the binary permutation number for an address
  - Note: The computed binary permutation number will be the same for all elements of a sequence because the permutation selector mask bits are all zero for the address bits within (and below) the sequence.
- "*S*" is the "base sequence" of 2^m slice numbers
- "*index*" will refer to the cache line number within a sequence (address bits (*m*+5):6), or equivalently the index of a slice number in the base sequence *S*

The full mapping of address to slice is then:

$$S(M(A) \oplus index)$$

In steps:

1. Compute the *binary permutation number* by applying the permutation selector masks to the address
2. Permute the *index* using that binary permutation number
3. Select the base sequence element at the *permuted index*

The 16-slice processors tested have a very simple structure that serves as a first concrete example. Every 16 cache line addresses are mapped onto L3s [0…15] using a binary permutation of the sequence [0…15]. Address bits [5:0] are offsets within each cache line, address bits [9:6] form the index within each sequence, and address bits [37:10] contribute to the permutation selectors. The base sequence is the identity permutation [0...15], i.e., the first 16 cache line addresses in memory are mapped to L3/CHA slices [0…15] in ascending order. There are a maximum of 16 binary permutations of this base sequence (all of which are observed in the data), so we need four permutation select equations. In the mask notation, the permutation associated with the sequence whose base address is *A* (for addresses below 2^{38}) is computed by:

$$p_0 = \bigoplus(\text{0x1b5f575400} \cdot A)$$
$$p_1 = \bigoplus(\text{0x2eb5faa800} \cdot A)$$
$$p_2 = \bigoplus(\text{0x3cccc93000} \cdot A)$$
$$p_3 = \bigoplus(\text{0x31aeeb1000} \cdot A)$$

Then the binary permutation applied to the base sequence is

$$perm = \sum_{i=0}^{3} p_i \times 2^i$$

Applying these masks to address 0 yields permutation 0 – consistent with the assertion that the sequence at address 0 is the base sequence. The second 16-cache-line sequence in memory starts at address 16*64=1024, or 0x400 in hex, which only overlaps with the mask for p_0, so the second 16 cache-line sequence is permutation 1 of the base sequence. After compensating for slight differences in formulation and address ranges used (see Appendix A), comparing these results to those of [8] shows that the current mask for p_0 matches theirs for 2/4/8 slices, the current mask for p_1 is the same as theirs for 4/8 slice configurations, and the current mask for p_2 is the same as theirs for 8 slice processors.

The non-power-of-2 case with the simplest structure is the 20-slice configuration. In this case the base sequence has a length of 256 elements, with all 256 possible binary permutations observed. The base sequence in this case is nonlinear, mapping the 8-bit index onto a permutation of the allowed L3/CHA slice numbers of [0…19]. The base sequence contains the values [0...15] 13 times each and the values [16...19] 12 times each. Note that the binary permutation operator can only rearrange the values within the base sequence, so *every* permutation will display the same non-uniformity with respect to bulk allocations across the L3/CHA slices.

3.3. **Derivation of Sequence Lengths, Permutation Select Masks, and Base Sequences**

When this work was initiated, it was by no means clear that all Intel processors of interest would contain address hashing functions with the same sorts of properties, or that the measurement approach would result in reliable identification of the address to L3/CHA slice mappings. The workflow presented here does not resemble the original exploratory process, and instead represents a somewhat optimized process for obtaining the parameters of the family of hash functions observed to be in use.

For each slice count, the first major step is to identify the length of the base sequence. Due to the nature of the XOR reductions and the address-based binary permutations, we find that all power-of-two sequence lengths *longer* than some minimum value can also serve as base sequences – indeed it is possible that the hardware implementation uses a single long sequence length for all slice counts, but that the internal symmetries of the base sequence allow it to be subdivided into smaller, functionally equivalent, base sequences. For each system, data was collected on aligned 2MiB pages, corresponding to blocks of 32768 cache lines. For every processor configuration, each 32768-cacheline block was found to be a binary permutation of every other 32768-cacheline block for that processor. The same was found for 16384-cacheline blocks. For block sizes less than 16384 cachelines some processors deviated from the perfect binary permutation structure. Results are summarized in Table 2, showing base sequence lengths from 2^4 to 2^{14}.

Code Name	L3/CHA slices	Model(s) tested	Base Sequence Length	Unique patterns observed
KNL	38	Xeon Phi 7250	4096	4096
SKX	14	Xeon Gold 6132	16384	*2048*
SKX & CLX	16	Xeon Gold 6142 & Silver 4216	16	16
SKX	18	Xeon Gold 6150	4096	*2048*
SKX	20	Xeon Gold 6148	256	256
SKX	22	Xeon Gold 6152	16384	*4096*
SKX	24	Xeon Platinum 8160	512	512
SKX & CLX	26	Xeon Platinum 8170 & 8260	16384	*8192*
SKX & CLX	28	Xeon Platinum 8180 & 8280	4096	4096
ICX	28	Xeon Gold 6330	16384	*4096*

Table 2: Minimum sequence length such that all observed sequences are binary permutations of each other. For the five cases in blue italics, multiple binary permutations result in the same output sequence.

The formulation of the hash makes it possible to derive the masks for the permutation selectors from a relatively small number of experiments. For each address bit above the top of the sequence, we need to find (at least) one pair of addresses that differ only in that bit. For the measurements used here, single-bit address differences in the address bits from the top of the sequence to bit 20 can be found within every 2MiB page. For address bits 21 and higher, the base addresses of the 2MiB pages are compared to find at least one pair that differs in only that bit.

Any pair of sequences with starting addresses that differ in only one particular bit will be binary permutations of each other, and the permutation number that converts one sequence to the other depends only on the starting address bit that differs. (Note that the binary permutation is its own inverse, so the permutation is the same in either direction.)

Table 3 provides an illustration of how the permutations associated with single-bit address changes can be separated into contributions to each of the permutation select bits, using the 26-slice SKX/CLX processor as an example. The row values are expanded from the observed permutations using:

$$perm_j = \sum_{i=0}^{13} p_{ij} \times 2^i, \quad 20 \le j \le 37$$

Here j is the row index (corresponding to bits of the physical address) and i is the column index (corresponding to the individual bits of the binary permutation number).

The values are then gathered by columns to create address masks (Table 4) that can be applied as described in Section 3.2 to obtain the permutation number used for the sequence starting at that address:

$$M_i = \sum_{j=20}^{37} p_{ij} \times 2^j, \qquad 0 \le i \le 13$$

Address bit	Single-address-bit permutation	p13	p12	p11	p10	p9	p8	p7	p6	p5	p4	p3	p2	p1	p0
37	9686	1	0	0	1	0	1	1	1	0	1	0	1	1	0
36	11558	1	0	1	1	0	1	0	0	1	0	0	1	1	0
35	9061	1	0	0	0	1	1	0	1	1	0	0	1	0	1
34	11931	1	0	1	1	1	0	1	0	0	1	1	0	1	1
33	1393	0	0	0	1	0	1	0	1	1	1	0	0	0	1
32	68	0	0	0	0	0	0	0	1	0	0	0	1	0	0
31	1172	0	0	0	1	0	0	1	0	0	1	0	1	0	0
30	10784	1	0	1	0	1	0	0	0	1	0	0	0	0	0
29	3815	0	0	1	1	1	0	1	1	1	0	0	1	1	1
28	1691	0	0	0	1	1	0	1	0	0	1	1	0	1	1
27	11897	1	0	1	1	1	0	0	1	1	1	1	0	0	1
26	1225	0	0	0	1	0	0	1	1	0	0	1	0	0	1
25	2047	0	0	0	1	1	1	1	1	1	1	1	1	1	1
24	3915	0	0	1	1	1	1	0	1	0	0	1	0	1	1
23	3190	0	0	1	1	0	0	0	1	1	1	0	1	1	0
22	1514	0	0	0	1	0	1	1	1	1	0	1	0	1	0
21	11313	1	0	1	1	0	0	0	0	1	1	0	0	0	1
20	12191	1	0	1	1	1	1	1	0	0	1	1	1	1	1

Table 3: Example analysis for the 16384-cacheline sequences used by the 26-slice SKX/CLX processors. For each address bit (column 1), column 2 shows the binary permutation number that relates a sequence at one address to the sequence at the address differing only in that bit. The remaining columns show how each bit of the binary permutation number depends on each of the single-address-bit changes.

Permutation Selector	Address Mask
p13	0x3C48300000
p12	0x0
p11	0x1469B00000
p10	0x36BFF00000
p9	0xC7B100000
p8	0x3A03500000
p7	0x24B6500000
p6	0x2B2FC00000
p5	0x1A6AE00000
p4	0x269AB00000
p3	0x41F500000
p2	0x39A2900000
p1	0x3433D00000
p0	0xE3F300000

Table 4: The values in Table 3 are gathered by column to generate these masks for the 26-slice SKX/CLX processors. The masks are used as described in Section 3.2 to compute the binary permutation number applied to the base sequence for that address.

The formulation of the base sequence used here is unambiguous – the "base sequence" is the sequence that starts at physical address zero, where the XOR-based permutation selectors must return permutation 0 – the identity permutation. It is not generally possible to test at physical address zero, but the "self-inverse" property of the binary permutation operator means that the base sequence can be retrieved from *any* sequence by simply applying the computed binary permutation number based on the sequence's starting address. The permutation masks are ambiguous in some cases, as will be discussed in Section 4.2.

Given the base sequence and permutation selector masks, all measured data can be validated.

4. Results

4.1. Permutation Selector Masks

Table 5 and Table 6 summarize the address hash information for each of the systems tested, including the permutation selector masks used to compute the permutation of the base sequence to be used for each sequence starting address. The "Low addr bit" is the first address bit above the top of each sequence. The "High addr bit" depends on the range of physical addresses used in the measurements and indicates the maximum address for which these permutation selectors are valid. In each of the cases for which the number of unique patterns found is less than the sequence length, one or more of the permutation selector masks have zero values and correspond to one possible set of permutation bits that are not used – see Section 4.2 for discussion of these zero values.

For the SKX 18-slice processor, the collected data was enough to obtain the dependence of the permutations on address bits 18-33 and 36 directly (i.e., via single-address bit differences). The dependence of the permutations on address bit 34 was inferred from sequences that differed in 2 address bits (36 and 34). None of the measurements included pages with address bits 37 or 35 set, so those dependencies remain unknown. The masks in Table 5 include contributions for address bits 36 and 34-18, but not 35 or 37, so they are labelled as only being valid to address bit 34.

Code Name	SKX	SKX & CLX	SKX	SKX	SKX
#slices	14	16	18	20	22
Low addr bit	20	10	18	14	20
High addr bit	37	37	*34*	37	37
Base Sequence Length	16384	16	4096	256	16384
Unique patterns	2048	16	2048	256	4096
p0	0x3880c00000	0x1b5f575400	*0x4c8fc0000*	0x3ecbad4000	0x3880c00000
p1	0x263a700000	0x2eb5faa800	*0x105380000*	0x35cf7c000	0x3433d00000
p2	0x1d14c00000	0x3cccc93000	*0x62b8c0000*	0x387242c000	0xf1d600000
p3	0x41f500000	0x31aeeb1000	*0x41f500000*	0xe2f28c000	0x41f500000
p4	0x1025400000	0x0	*0x46d780000*	0x1c5e518000	0x1025400000
p5	0x2cd5100000	0x0	*0x4d5140000*	0x38bca30000	0x2cd5100000
p6	0x1d90300000	0x0	*0x15d80c0000*	0xfb2eb4000	0x1d90300000
p7	0x0	0x0	*0x133f480000*	0x1f65d68000	0x1209a00000
p8	0x3a03500000	0x0	*0x1203500000*	0x0	0x3a03500000
p9	0xc7b100000	0x0	*0x433280000*	0x0	0xc7b100000
p10	0x0	0x0	*0x0*	0x0	0x0
p11	0x1469b00000	0x0	*0x1469b40000*	0x0	0x1469b00000
p12	0x0	0x0	*0x0*	0x0	0x0
p13	0x3c48300000	0x0	*0x0*	0x0	0x3c48300000

Table 5: Address hash properties and permutation selector masks for the first half of the systems tested. Masks for permutation selector bits beyond the length of the base sequence are in grey. The SKX 18-slice results do not include contributions from address bits 37 or 35.

Code Name	SKX	SKX & CLX	SKX & CLX	ICX	KNL
#slices	24	26	28	28	38
Low addr bit	15	20	18	20	18
High addr bit	37	37	37	37	37
Base Sequence Length	512	16384	4096	16384	4096
Unique patterns	512	8192	4096	4096	4096
p0	0x2b72c98000	0xe3f300000	0x32770c0000	0x3880c00000	0x32770C0000
p1	0x16e5930000	0x3433d00000	0x3433d40000	0x390100000	0x2BBAC80000
p2	0x2dcb260000	0x39a2900000	0x39a2900000	0x38bea00000	0x39A2900000
p3	0x1b964c0000	0x41f500000	0x3857680000	0x41f500000	0x1B964C0000
p4	0x1c5e518000	0x269ab00000	0x1ad2880000	0x1025400000	0x055B940000
p5	0x38bca30000	0x1a6ae00000	0x1a6ae40000	0x2cd5100000	0x05E3F80000
p6	0x1a0b8f8000	0x2b2fc00000	0x2b2fc40000	0x1d90300000	0x1767FC0000
p7	0x1f65d68000	0x24b6500000	0x24b6540000	0x25aa600000	0x3B3F480000
p8	0x15b9648000	0x3a03500000	0x3a03500000	0x3a03500000	0x258A4C0000
p9	0x0	0xc7b100000	0xc7b100000	0xc7b100000	0x3033280000
p10	0x0	0x36bff00000	0xaf7c80000	0x0	0x2936EC0000
p11	0x0	0x1469b00000	0x28218c0000	0x1469b00000	0x1469B40000
p12	0x0	0x0	0x0	0x0	0x0
p13	0x0	0x3c48300000	0x0	0x3c48300000	0x0

Table 6: Address hash properties and permutation selector masks for the second half of the systems tested. Masks for permutation selector bits beyond the length of the base address sequence are in grey.

4.2. **Ambiguity in Permutation Selector Masks**

In Section 3.3 it was claimed that the formulation of the base sequence used here was "unambiguous" – corresponding to the sequence of values starting at address zero. For the processors with fewer observed patterns than available permutations, however, the derived permutation selector masks are not unique – i.e., when comparing sequences whose addresses vary in a single bit, multiple matching permutation numbers are found. Using any of these matching permutation numbers for each single-bit address change will result in a functionally correct solution, but with different permutation select masks.

The entries in Table 5 and Table 6 were produced by selecting the *smallest-numbered* matching permutation for each single-bit address change. When there are 2^q matching permutations for each single-bit address change, choosing the smallest permutation numbers results in zero values for all address bits in q of the permutation selectors – e.g., the column under p12 in Table 3. Choosing the *larger* of the 2 matching permutation numbers would result in all "1" values in p12 in Table 3.

While it is conceivable that the ambiguity is due to deliberate "skipping" of permutation selector bits (as suggested by the structure of the current results in Table 5 and Table 6), it seems more likely that the *apparent* absence of some permutation numbers is due to correlations of the base sequence with the binary permutation operator, with multiple permutation numbers acting to produce identical output sequences. Intuitively, it seems more likely that the single-address-bit permutation numbers are approximately uniformly distributed across the matching permutation numbers, leading to a very large set of possible combinations of models that match the observations. The potential for disambiguation of these cases remains an open question.

Due to the linearity of the binary permutation operator, in each ambiguous case the multiple matching permutations for each single-address-bit change are related to each other by the same set of "identity permutations" that map the base sequence back to itself. Direct application of all the possible binary permutation operators on each of the base sequences shows a striking commonality in these "identity permutations", as reported in Table 7. While it may not be surprising that the 14-slice and 28-slice hashes share several identity permutations (given the common prime factor of 7), it does seem surprising that the 18, 22, and 26-slice hashes are also so closely related. It seems likely that these common identity permutations imply important structural similarities across the family of base sequence generators in use.

Processor	slices	SeqLength	Identity Permutations						
SKX/CLX	26	*16384*					6371		
SKX	22	*16384*			1269		6371	7190	
ICX	28	*16384*			1269		6371	7190	
SKX	18	4096		1139		6245		7190	
SKX	14	*16384*	134	1139	1269	6245	6371	7190	7312

Table 7: Permutations (other than 0) mapping the base sequence onto itself. If the base sequence for the SKX 18-slice system is extended to 16384 elements, the values 6245 and 7190 (plus four more values > 8192) become identity permutations.

4.3. **Base Sequences**

The full set of base sequences derived here are provided as text files in the same repository as this document: https://hdl.handle.net/2152/87595. Appendix B contains a listing of the accompanying files. Several of the shorter sequences are encoded as text in Appendix C.

The initial elements of each of the derived base sequences are presented in Table 8. Some features jump out:

- All sequences begin with slice 0.
- Five cases show a linear mapping of index to slice number over the first 14-16 entries.

- The five cases that start with the linear mapping are the also the only five cases for which the first "L" entries form a permutation of the indices [0…L-1].
- The SKX 18/20 slice results match exactly in the first 26 entries.
- The SKX 22/26/28 slice results match exactly in the first pass through their respective range of slice numbers, with entries at index 16 and above replaced by binary permutation 1 of the indices in that range.
- The 16-slice processor uses the trivial base sequence composed of the sequence of slice numbers [0…15].
- Unlike previous generational changes, the 3rd generation Xeon Scalable Processor (ICX) 28-slice uses a different address hash than the 1st and 2nd gen Xeon Scalable Processors (SKX/CLX) with 28 slices.

index	SKX 14	SKX 22	SKX 26	ICX 28	KNL 38	SKX 18	SKX 28	SKX 24	SKX 20	SKX 16
0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	19	11	1	3	11	1
2	2	2	2	2	2	2	2	10	2	2
3	3	3	3	3	17	9	3	9	9	3
4	4	4	4	18	15	7	4	7	7	4
5	5	5	5	19	28	12	5	20	12	5
6	6	6	6	20	13	5	6	13	5	6
7	7	7	7	21	30	14	7	22	14	7
8	8	8	8	8	5	1	8	5	1	8
9	9	9	9	9	22	10	9	6	10	9
10	10	10	10	10	7	3	10	15	3	10
11	11	11	11	11	20	8	11	12	8	11
12	12	12	12	26	10	6	12	2	6	12
13	13	13	13	27	25	13	13	17	13	13
14	4	14	14	18	8	4	14	8	4	14
15	11	15	15	25	27	15	15	19	15	15
16	1	17	17	1	1	1	17	1	1	
17	0	16	16	0	18	10	16	2	10	
18	3	19	19	3	3	3	19	11	3	
19	2	18	18	2	16	8	18	8	8	
20	5	21	21	19	14	6	21	6	6	
21	4	20	20	18	29	13	20	21	13	
22	7	5	23	21	12	4	23	12	4	
23	6	14	22	20	31	15	22	23	15	
24	9	1	25	9	4	0	25	4	0	
25	8	10	24	8	23	11	24	23	11	
26	11	19	15	11	26	10	27	14	18	
27	10	16	20	10	37	17	26	21	17	
28	13	7	3	27	11	7	1	3	7	
29	12	12	16	26	24	12	16	16	12	
30	5	17	9	19	33	17	11	9	17	
31	10	18	18	24	34	14	18	18	18	
32	2	18	18	16	16	8	18	0	8	
33	3	19	19	17	3	3	19	3	3	
34	0	16	16	14	18	10	16	10	10	
35	1	17	17	15	1	1	17	9	1	
36	6	14	22	6	31	15	22	23	15	
37	7	5	23	7	12	4	23	4	4	

Table 8: Initial elements of each of the base sequences by processor and number of L3/CHA slices. The outlines indicate the number of L3/CHA slices "L" for each processor. Slice numbers that equal the index number in the first "L" entries are highlighted in green.

5. Analysis

5.1. Common Features of the Permutation Selector Masks

Review of the permutation selector mask values in Table 5 and Table 6 shows some significant commonalities. The SKX 14-slice, SKX 22-slice, and ICX 28-slice processors have identical mask values for 9 of the 11 or 12 non-zero selectors. Starting with the SKX 14-slice mask values, Figure 2 shows how the values are transformed to the values for ICX 28-slice, SKX 22-slice, SKX 26-slice, and SKX 28-slice, using only a handful of constants.

c1	c2	c3	c4/c5/c6
0x0025aa600000	0x0037a3c00000	0x0036bff00000	0x003c48300000
			0x000000040000
			0x000000080000

perm bit	SKX-14	ICX-28	SKX-22	SKX-26	SKX-28
0	Base	<- match	<- match	<-- XOR c3	<- ^c4^c5^c6
1	Base	<-- XOR c1	<-- XOR c2	<- match	<- XOR c5
2	Base	<-- XOR c1	<-- XOR c2	<-- XOR c3	<- match
3	Base	<- match	<- match	<- match	<- ^c4^c6
4	Base	<- match	<- match	<-- XOR c3	<- ^c4^c6
5	Base	<- match	<- match	<-- XOR c3	<- XOR c5
6	Base	<- match	<- match	<-- XOR c3	<- XOR c5
7	*null*	<-- XOR c1	<-- XOR c2	<-- XOR c3	<- XOR c5
8	Base	<- match	<- match	<- match	<- match
9	Base	<- match	<- match	<- match	<- match
10	*null*	*null*	*null*	<-- XOR c3	<- ^c4^c6
11	Base	<- match	<- match	<- match	<- ^c4^c5^c6
12	*null*	*null*	*null*	*null*	*null*
13	Base	<- match	<- match	<- match	<- XOR c4

Figure 2: Starting with the permutation selector mask values for SKX 14-slice, the values for four other configurations can be computed using only a small number of constants.

Only one minor commonality was seen between the mask values used for KNL and those used by any of the new processors – the mask used for permutation bit 11 matches that from the first four entries in Figure 2 except for having address bit 18 set. (This is the same as XOR'ing the value with the constant c5 from Figure 2.)

No commonalities were seen between the SKX 16-slice masks and those used by the other configurations. This is not surprising – the 16-slice case has a trivial base sequence and does not require that the permutation selector masks be chosen with consideration of the repeat patterns in the base sequence.

The SKX 20-slice and SKX 24-slice share mask values only with each other and show a pattern similar to that seen in Figure 2.

- Three of the mask values are identical (for permutation bits p4, p5, p7)
- SKX 20-slice p8 mask value is zero (due to the 256-element sequence length)
- SKX 24-slice p8 mask value is 0x15b9648000
    - The SKX 24-slice mask values for permutation bits 0, 1, 2, 3, 6 are equal to the corresponding SKX 20-slice values XOR'd with the constant 0x15b9648000

In the available address bits, SKX 18-slice masks match those of SKX 14, SKX 22, and ICX 28 for three permutation select bits (p3, p5, p8) and differ in only address bit 18 for a fourth (p11).

5.2. **Bulk Distribution of Addresses across the Slices**

While the statistical properties of the hash functions can, of course, be derived from the complete representation, many analyses were performed directly on the collected data before the complete representation was available. These provide insights into the some of the performance implications of the hashing functions used and may be useful in identifying a common underlying family of hash functions, of which these are specific instantiations

As noted in Section 3.2, the binary permutation functions used in these address hashes simply reorder the allocation of cache line addresses to slices and cannot change the overall counts. For almost all configurations, the minimum (aligned, power-of-2) block size that results in asymptotically uniform distribution of lines across the slices is the same as the base sequence block size.

Table 9 shows the number of cache line addresses assigned to each slice for any permutation of the base sequence, or for the minimum block size that provides the same relative distribution. As expected, the power-of-two slice count (16) shows perfectly uniform distribution across every naturally aligned sequence of 16 addresses. The remainder of the SKX/CLX/ICX configurations show a small degree of non-uniformity, with the most heavily allocated slices receiving 1.6% to 3.7% more cache lines than with a uniform distribution. The KNL platform is slightly more uneven, with the 16 most heavily loaded CHAs receiving 7.6% more cache lines than would be expected for a maximally uniform distribution. For most systems, the shortest block size that provides asymptotically uniform distribution is the same as the base sequence length. KNL is one exception – the first and last halves of the base sequence have the same counts of CHA allocations, so the table shows results for 2048-cacheline half-sequences[6]. The 18-slice SKX/CLX processors are also an exception, with each 512-element aligned sequence showing the same bulk allocation across the slices.

The color-coding in Table 9 suggests certain commonalities in structure across the hash functions. The structure can be modeled as an iterative decomposition into one, two, or three power-of-two groups, with the members of each group receiving the same number of cache line addresses. For the SKX/CLX processors, the groups are mapped to the L3/CHA numbers in descending order of size (e.g., 8, 4, 2 for the 14-slice configuration). For the KNL and ICX processors, the decomposition follows the same pattern but the mapping onto slice numbers is swizzled in two slightly different ways. Note that the *ratios* of cache lines allocated to each group of slices are the same for the SKX/CLX 28-slice processor and the ICX 28-slice processor, but the ICX 28-slice processor requires a 4x longer sequence to achieve this asymptotically uniform distribution.

In comparing the results of Table 9 with the discussion of re-used permutation selector mask values in Section 5.1, we see correlations between the number of slices and the mask values. Specifically, two of the three configurations that divide the number of slices into two different factors of two (SKX 20-slice, SKX 24-slice) have closely related mask values, and the configurations that divide the slices into three different factors of two (SKX 14-slice, SKX 22-slice, SKX 26-slice, SKX 28-slice, and ICX 28-slice[7]) have common mask values (though with fairly complex relationships). The SKX 18-slice does not follow these relationships – it has a non-linear initial base sequence (like SKX 20-slice and SKX 24-slice), but shares mask values in common with the set of configurations that divide the slices into three factors of two and use base sequences that are initially linear.

[6] Note that the binary permutation operator cannot move any elements of the sequence from the lower half to the upper half unless it swaps *all* the items in the lower and upper halves. This ensures that if any aligned power-of-two subset of the base sequence has asymptotically uniform distribution across the slices, all larger aligned power-of-two sequence sizes will also retain this property.

[7] KNL does not share these mask values, but it was a standalone product released a year earlier than any of the others and may have represented an earlier stage of development of the address hashing functions.

	1st and 2nd gen Xeon Scalable Processors (Skylake Xeon / Cascade Lake Xeon)									
L3/CHA slice number	**14 slice**	**16 slice**	**18 slice**	**20 slice**	**22 slice**	**24 slice**	**26 slice**	**28 slice**	**KNL**	**ICX 28 slice**
0	1192	1	*29*	13	772	21	638	149	52	596
1	1192	1	*29*	13	772	21	638	149	52	596
2	1192	1	*29*	13	772	21	638	149	58	596
3	1192	1	*29*	13	772	21	638	149	58	596
4	1192	1	*29*	13	772	21	638	149	52	596
5	1192	1	*29*	13	772	21	638	149	52	596
6	1192	1	*29*	13	772	21	638	149	58	596
7	1192	1	*29*	13	772	21	638	149	58	596
8	1200	1	*29*	13	772	21	638	149	52	600
9	1200	1	*29*	13	772	21	638	149	52	600
10	1200	1	*29*	13	772	21	638	149	58	600
11	1200	1	*29*	13	772	21	638	149	58	600
12	1024	1	*29*	13	772	21	638	149	52	512
13	1024	1	*29*	13	772	21	638	149	52	512
14		1	*29*	13	772	21	638	149	58	596
15		1	*29*	13	772	21	638	149	58	596
16			*24*	12	752	22	644	150	52	596
17			*24*	12	752	22	644	150	52	596
18				12	752	22	644	150	58	596
19				12	752	22	644	150	58	596
20					512	22	644	150	52	596
21					512	22	644	150	52	596
22						22	644	150	58	600
23						22	644	150	58	600
24							512	128	52	600
25							512	128	52	600
26								128	58	512
27								128	58	512
28									52	
29									52	
30									58	
31									58	
32									48	
33									48	
34									48	
35									48	
36									48	
37									48	

Table 9: Counts of cache lines allocated to each L3/CHA slice for the base sequence, or for the minimum block size yielding the same distribution pattern. Colors indicate relative degree of loading across the slices.

5.3. **Introduction to Base Sequence Generating Functions**

Very few non-power-of-two base sequences have been transformed into compact generator functions that are plausibly related to the actual hardware implementation. The 6-slice case was studied by [5], and includes a simple nonlinear function to map from a 7-bit permuted index number to a slice number in the range of [0…5]. The structure of the generator function is easier to see in the 6-slice case, so we begin by reviewing that case.

Let i_0 through i_6 be the (permuted) address bits above the top of the sequence, then[8]:

$$s_2 = (i_0 \oplus i_5) \cdot (i_2 + (i_3 \cdot (i_4 + i_5)))$$
$$s_1 = i_1 \cdot \overline{s_2}$$
$$s_0 = i_0 \oplus i_1 \oplus i_2 \oplus i_3 \oplus i_4 \oplus i_6$$

Here $\oplus$ is the binary XOR operator, $\cdot$ is the logical AND operator, $+$ is the logical OR operator, and the overbar is the logical NOT operator. The slice number used is $4s_2 + 2s_1 + s_0$ (in ordinary integer arithmetic).
The formulation for s_2 is carefully structured to be "true" 34.375% of the time – close to the 33.333% of the time the high-order bit of the slice number would be set if the sequence displayed a perfectly uniform distribution across the slices. The formulation for s_1 ensures that s_2 and s_1 cannot both be set at the same time, preventing the function from generating references to the non-existent slice numbers 6 or 7 in its 3-bit output. The formula for s_0 is a simple alternating swizzle. The slight overabundance of "true" values of s_2 is responsible for the slight overrepresentation of slices 4 and 5 in the 128-element sequence – slices 0 to 3 are allocated 21 lines each and slices 4 and 5 are allocated 22 lines each in the base sequence (and in all its binary permutations).

The permutation selector masks used to derive the above differ slightly from those used here (i.e., type (c) as discussed in Appendix A) but the overall structure of the equations is minimally changed.

5.4. Compact Generating Function for the 24-slice SKX/CLX Processors

Following the example of the 6-slice generating function, the base sequence for the 24-slice SKX/CLX processors was reduced to a compact set of equations. The 512-line base sequence uses 9 (permuted) address bits, which are converted to a 5-bit output value that is constrained to the range [0…23] and are computed from:

$$s_4 = (i_0 \oplus i_5 \oplus i_6) \cdot \big((i_2 \oplus i_7 \oplus i_8) + i_4 + i_5\big) \cdot \Big(\big((i_2 \oplus i_7) + i_3 + i_8\big) \oplus \big((i_2 \oplus i_7) \cdot i_3 \cdot i_8\big)\Big)$$
$$s_3 = (i_1 \oplus i_6 \oplus i_7) \cdot \overline{s_4}$$
$$s_2 = i_2 \oplus i_3 \oplus i_6$$
$$s_1 = i_0 \oplus i_1 \oplus i_2 \oplus i_6$$
$$s_0 = i_0 \oplus i_2 \oplus i_3 \oplus i_4 \oplus i_8$$

The s_4 equation is significantly more complex than in the 6-slice case, but it has the same probability of being "true" – 34.375% of the time (176 of 512 vs 44 of 128). The equation for s_3 ensures that the two high-order bits cannot be set at the same time, preventing the selector from generating slice numbers in the range of [24…31] in its 5-bit output. This similarity also leads to the same degree of non-uniformity in the bulk allocations – each 512-line permutation of the base sequence assigns 21 addresses to each of slices [0…15] and 22 to each of slices [16…23]. The remaining terms are simple XOR swizzles.

5.5. Approximating a Base Sequence – SKX 14-slice

The observations in Table 8 suggest that the structure of the base sequences may be amenable to additional simplification – especially if some errors in the prediction are tolerable. Taking the SKX 14-slice results as an example, the base sequence begins with $S_i = i$ for $0 \le i < 14$, then the 15th and 16th elements of the sequence are from a different pattern (with slice numbers of 4 and 11). Reviewing the next few 16-element sub-blocks of the base sequence shows that either 14 or 15 of the entries in every 16-element sub-block can be correctly predicted by a binary permutation of the first 16 values. In every sub-block the predicted slice numbers derived from permuting the first 14 of the 16 elements were always correct, while the predicted values obtained from permuting the last two elements were usually (but not always) wrong. These must have been produced by higher-order terms in the base sequence generator.

Given this encouraging result from manual investigation, the process was automated to include all 1024 16-element sub-blocks of the base sequence. For each sub-block, the "best" permutation of the first 16 elements was

[8] The formulation follows that of [5] but parentheses have been added to the first equation to clarify the order of operations.

found and recorded. Results showed 768 sub-blocks contained two mispredictions each, 168 contained one misprediction each, and the remaining 88 were correct in all 16 elements. The total number of mispredictions using this approach is 1704 of 16384, or about 10.4%. With these permutation numbers, the amount of data required to specify the base sequence was reduced from 65536 bits (16384 values at 4 bits each) to 4160 bits (16*4 bits for the first 16 entries, plus 1024*4=4096 bits for the permutation numbers of the 1024 sub-blocks).

On additional inspection, it was found that the optimum permutation for each sub-block could be computed by XOR-reduction of the sub-block number. This allows an approximation to the 16384-element base sequence to be represented very compactly as:

$$p_0 = \bigoplus(0\text{x}1\text{d}5 \cdot blk)$$
$$p_1 = \bigoplus(0\text{x}2\text{aa} \cdot blk)$$
$$p_2 = \bigoplus(0\text{x}24\text{c} \cdot blk)$$
$$p_3 = \bigoplus(0\text{x}2\text{c}4 \cdot blk)$$

Where "*blk*" is the sub-block number in the base sequence – i.e., address bits 19:10. The binary permutation applied to the first 16 elements of the base sequence is:

$$\sum_{i=0}^{3} p_i \times 2^i$$

This formulation provides correct slice numbers for 89.6% of the elements of the base sequence while requiring only about 104 bits of storage (16*4 bits for the base sequence and 4*10 bits for the sub-block permutation selector masks). If one drops storage of the base sequence and simply uses with $S_i = i$ for all 16 elements, then the results will be correct for 87.5% of the locations, with illegal slice numbers of 14 and 15 for the remainder, but at a total data requirement of only 40 bits (4 10-bit selector masks).

After generating a 16384-element "approximate base sequence" using this approach, the permutation selector masks from Table 5 are used to generate permutations for higher addresses, as in previous sections.

6. Summary

A methodology has been presented for converting address-to-L3/CHA-slice measurements into a compact representation based on a "base sequence" of 16 to 16384 slice numbers plus a set of "permutation selector" masks. For each cache line address, the permutation selector masks are applied to the physical address to compute a permuted index. The element of the base sequence at the permuted index contains the slice number for that address. Complete base sequences (as lists) and permutation selector masks for all physical addresses below 256 GiB[9] are provided for Xeon Scalable Processors (1st and 2nd generation) with 14, 16, 18, 20, 22, 24, 26, 28 slices, for Xeon Scalable Processors (3rd generation) with 28 slices, and for Xeon Phi x200 Processors with 38 slices.

A relatively compact set of 5 binary equations is provided to reproduce the base sequence for the Xeon Scalable Processors (1st and 2nd generation) with 24 slices, and a simple approximation is derived that provides correct slice identification for almost 90% of addresses on the 14-slice models.

Many characteristics of the results presented here suggest that the base sequences used in these systems are generated by nonlinear functions of a single "family". From an implementation perspective, it is likely that there is a single "root" generating function, with model-specific mask values selecting the desired address bits to

[9] Except for the SKX 18-slice processor, where the masks are only valid for addresses in the first 32 GiB.

participate in each of the clauses of the operator. It is hoped that the availability of this data will encourage further study in the topic.

Caveat Emptor:
The mappings reported appear to be correct and applicable across different systems using the same processors, but it remains unknown whether these processors can configure alternate address mapping functions – either at manufacturing time or in later microcode updates.

7. Acknowledgments

This work has been supported by grants from the National Science Foundation, including award numbers 1663578 and 1854828. Special thanks to the Dell Customer Solutions Center for providing access to six of the processor configurations reported here. Thanks to Yuval Yarom for helping me understand the mapping inversion procedure, and to Gabriel Rodríguez Álvarez for additional test results and many helpful technical discussions.

8. References

[1] J. D. McCalpin, "HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, Nov. 2018, pp. 225–237. doi: 10.1109/SC.2018.00021.

[2] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World," in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2019, pp. 888–904. doi: 10.1109/SP.2019.00004.

[3] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *2013 IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2013, pp. 191–205. doi: 10.1109/SP.2013.23.

[4] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors," in *2015 Euromicro Conference on Digital System Design*, Madeira, Portugal, Aug. 2015, pp. 629–636. doi: 10.1109/DSD.2015.56.

[5] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," 2015/905, Sep. 2015. [Online]. Available: https://eprint.iacr.org/2015/905

[6] J. D. McCalpin, "Mapping Core and L3 Slice Numbering to Die Location in Intel Xeon Scalable Processors," Texas Advanced Computing Center, Austin, Texas, USA, TR-2021-01b, Feb. 2021. [Online]. Available: http://dx.doi.org/10.26153/tsw/13119

[7] J. D. McCalpin, "Mapping Core, CHA, and Memory Controller Numbers to Die Locations in Intel Xeon Phi x200 ('Knights Landing', 'KNL') Processors," Texas Advanced Computing Center, Austin, Texas, USA, TR-2021-02. [Online]. Available: http://dx.doi.org/10.26153/tsw/13120

[8] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters," in *Research in Attacks, Intrusions, and Defenses*, vol. 9404, H. Bos, F. Monrose, and G. Blanc, Eds. Cham: Springer International Publishing, 2015, pp. 48–65. doi: 10.1007/978-3-319-26362-5_3.

[9] S. Kommrusch, M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño, "Optimizing Coherence Traffic in Manycore Processors using Closed-Form Caching/Home Agent Mappings," *IEEE Access*, pp. 1–1, 2021, doi: 10.1109/ACCESS.2021.3058280.

[10] S. Kommrusch, M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño, "Coherence Traffic in Manycore Processors with Opaque Distributed Directories," *ArXiv201105422 Cs*, Nov. 2020, Accessed: Jul. 26, 2021. [Online]. Available: http://arxiv.org/abs/2011.05422

[11] Intel Corporation, "Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual," Intel Corporation, 336274–001, Jul. 2017.

Appendix A: Equivalent Formulations

The literature contains a variety of slightly different formulations for the construction of the hash function, particularly with respect to the details of the permutation selector masks, *M*. Consider three structures for the selector masks for a hypothetical processor with 16 slices as presented in Figure 3. For version (a) (as used here), let M_a be the mask formulation from Figure 3(a), then the slice number is computed

$$S(M_a(A) \oplus index)$$

For version (b), augmenting the masks with scaled identity matrix causes the binary permutation operator to be absorbed into the $M_b(A)$ operation, so the slice number is computed independently for each cache line

$$S(M_b(A))$$

For version (c), the masks are augmented with any invertible binary matrix and the slice number is computed as in (b) – but with a base sequence function ($\tilde{S}$) that is permuted from the one used in (a) and (b):

$$\tilde{S}(M_c(A))$$

(a) Mask formulation as used here – address bits *within* a sequence ("index") are not used to compute the mask but are permuted using the binary permutation computed for the entire sequence.

	"sequence number" -- address bits above top of sequence														"index" -- address bits inside sequence			
addr bit -->	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
M_a0	0	1	0	1	0	1	1	1	0	1	0	1	0	1	0	0	0	0
M_a1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	0	0	0	0
M_a2	1	1	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0
M_a3	1	1	1	0	1	0	1	1	0	0	0	1	0	0	0	0	0	0

(b) Equivalent mask formulation – Masks extended with identity matrix and applied to each cache line address to obtain the permuted index value.

	"sequence number" -- address bits above top of sequence														"index" -- address bits inside sequence			
addr bit -->	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
M_b0	0	1	0	1	0	1	1	1	0	1	0	1	0	1	0	0	0	1
M_b1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	0	0	1	0
M_b2	1	1	0	0	1	0	0	1	0	0	1	1	0	0	0	1	0	0
M_b3	1	1	1	0	1	0	1	1	0	0	0	1	0	0	1	0	0	0

(c) Equivalent mask formulation – Mask extended with any non-singular pattern for "index" bits and applied to each cache line address to obtain permuted index value.

	"sequence number" -- address bits above top of sequence														"index" -- address bits inside sequence			
addr bit -->	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
M_c0	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	1
M_c1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0
M_c2	1	1	0	0	1	0	0	1	0	0	1	1	0	0	0	1	0	0
M_c3	1	1	1	0	1	0	1	1	0	0	0	1	0	0	1	0	0	0

Figure 3: Example of three equivalent structures of the permutation selector mask matrix. (a) is the formulation used here, (b) is a commonly used formulation that is exactly equivalent, (c) provides the same mapping, but with a modified base sequence function.

Appendix B: List of Associated Data Files

The following plain text files contain the complete base sequences and permutation selector masks derived in this report and are available at the same location as this report: https://hdl.handle.net/2152/87595
The READ_ME.txt file describes the naming conventions and file formats used in the data files.

READ_ME.txt
BaseSequence_ICX_28-slice.txt
BaseSequence_KNL_38-slice.txt
BaseSequence_SKX_14-slice.txt
BaseSequence_SKX_16-slice.txt
BaseSequence_SKX_18-slice.txt
BaseSequence_SKX_20-slice.txt
BaseSequence_SKX_22-slice.txt
BaseSequence_SKX_24-slice.txt
BaseSequence_SKX_26-slice.txt
BaseSequence_SKX_28-slice.txt
PermSelectMasks_ICX_28-slice.txt
PermSelectMasks_KNL_38-slice.txt
PermSelectMasks_SKX_14-slice.txt
PermSelectMasks_SKX_16-slice.txt
PermSelectMasks_SKX_18-slice.txt
PermSelectMasks_SKX_20-slice.txt
PermSelectMasks_SKX_22-slice.txt
PermSelectMasks_SKX_24-slice.txt
PermSelectMasks_SKX_26-slice.txt
PermSelectMasks_SKX_28-slice.txt

Appendix C: Base Sequences Encoded as Text

For the shorter sequences, it is practical to include the sequence values encoded as text ("a" == 0, "b" == 1, …). The 16-element sequence is simply the values [0…15]. Only a few cases are short enough for this to be a tolerable method of conveyance.

SKX/CLX 20-slice base sequence (256 elements)

```
alcjhmfobkdignepbkdignepalsrhmrsidkbpengjctqofqtjclaofmhidsrperskbidngpelajc
mhoflajcmhofsridrspecjalfohmtqbkqtgndibkepgnsralrshmepgndibkfohmcjalfohmcjal
qtgntqbkmhoflajcrspesridngpekbidqtoftqjcofmhjclapengidkbpengidkbofqtjctqgnep
bkdihmrsalsrhmfoalcjgnqtbktq
```

SKX/CLX 24-slice base sequence (512 elements)

This sequence should match the values from the equations in Section 5.4 for the indices [0…511].

```
adkjhunwfgpmcritbcligvmxexovdqjsadkjxevovgxmsbqlbcliwfupuhwntarkonehrktalibc
upwfpmfgqlsbsjqdvoxeonehjsdqlqbsmxgvpmfgitcrkratnwhuitcrpmfgnwhukjadjsdqoneh
mxgvlqbsqlsbpmfgvoxesjqdrktaonehupwftircwfupbclitarkehonxevoadkjsbqlvgxmgvmx
bclidqjsexovhunwadkjcritfwpubslqgfmpexovdajkatkrhunwfwpucbilrctiwfupuhwndajk
qdsjxevovgxmcbilxmvgilcbsjqdnohewnuhrktatircmpgfpufwitcrkratnoheovexjsdqlqbs
mpgfjkdaovexmpgflqbsitcrpufwnohekratrktawnuhmpgftircqlsbxmvgnohesjqdhenoqdsj
cbilvgxmwfuprctidajkuhwnhunwatkrcbilfwpugvmxbslqdajkexov
```

SKX/CLX 28-slice base sequence (4096 elements)

Here “a” to “z” indicate slices 0 to 25, augmented by the “|” character for slice 26 and “{” for slice 27.

```
abcdefghijklmnoprqtsvuxwzy|{bqlsstqrwxuv{|yzsdqjdcbahgfelkjiponmnmpojilkfehgbadcmng
hyz{|uvwxqrstpofe|{zyxwvutsrqopmnklijghefcdabwxuvstqroxev{|yzhgfedcbaponmlkjiefghab
cdmnopijklvuxwrqtsvmxgzy|{|{zydcjitsrqxwvuklijopmncdabghefjilknmpobadcfehgyz{|abklq
rstuvwxbadcfehgjilknmpoqrstuvwxyz{|abkltsrqxwvu|{zydcjicdabghefklijopmnmnopijklefgh
abcdvmxgzy|{vuxwrqtsoxev{|yzwxuvstqrponmlkjihgfedcbaxwvutsrqpofe|{zyghefcdabopmnkli
jfehgbadcnmpojilkuvwxqrstmnghyz{|{|yzsdqjstqrwxuvlkjiponmdcbahgfeijklmnopabcdefghzy
|{bqlsrqtsvuxwcdabghefklijopmntsrqxwvu|{zyhwnuqrstuvwxyz{|ufwpbadcfehgjilknmpoponml
kjihgfedcbaklab{|yzwxuvstqrjqdszy|{vuxwrqtsmnopijklefghabcduvwxqrstirctyz{|fehgbadc
nmpojilkghefcdabopmnklijxwvutsrqtkra|{zyzy|{vexorqtsvuxwijklmnopabcdefghlkjiponmdcb
ahgfe{|yzgxmvstqrwxuvdcbahgfelkjiponmstqrwxuv{|yzghmnrqtsvuxwzy|{vexoabcdefghijklmn
opopmnklijghefcdabtkra|{zyxwvutsrqirctyz{|uvwxqrstnmpojilkfehgbadcvuxwrqtsjqdszy|{e
fghabcdmnopijklhgfedcbaponmlkjiwxuvstqrslqb{|yzyz{|ufwpqrstuvwxjilknmpobadcfehgklij
opmncdabghef|{zyhwnutsrqxwvufepozy|{vuxwrqtsmnopijklefghabcdponmlkjihgfedcbaghmn{|y
zwxuvstqrqrstuvwxyz{|irctbadcfehgjilknmpocdabghefklijopmntsrqxwvu|{zytkralkjiponmdc
bahgfe{|yzklabstqrwxuvzy|{jidcrqtsvuxwijklmnopabcdefghghefcdabopmnklijxwvutsrqhwnu|
{zyuvwxqrstufwpyz{|fehgbadcnmpojilkufwpyz{|uvwxqrstnmpojilkfehgbadcopmnklijghefcdab
hwnu|{zyxwvutsrqrqtsvuxwzy|{jidcabcdefghijklmnopdcbahgfelkjiponmstqrwxuv{|yzklabkli
jopmncdabghef|{zytkratsrqxwvuyz{|irctqrstuvwxjilknmpobadcfehghgfedcbaponmlkjiwxuvst
qrghmn{|yzvuxwrqtsfepozy|{efghabcdmnopijkltcri|{zyxwvutsrqopmnklijghefcdabnmpojilkf
ehgbadcarktyz{|uvwxqrststqrwxuv{|yzoxevdcbahgfelkjiponmabcdefghijklmnoprqtsvuxwzy|{
vmxgjilknmpobadcfehgyz{|unwhqrstuvwx|{zypofetsrqxwvuklijopmncdabghefefghabcdmnopijk
lvuxwrqtsbqlszy|{wxuvstqrsdqj{|yzhgfedcbaponmlkjisdqj{|yzwxuvstqrponmlkjihgfedcbamn
opijklefghabcdbqlszy|{vuxwrqtstsrqxwvu|{zypwfucdabghefklijopmnbadcfehgjilknmpoqrstu
vwxyz{|unwhijklmnopabcdefghzy|{vmxgrqtsvuxw{|yzoxevstqrwxuvlkjiponmdcbahgfefehgbadc
nmpojilkuvwxqrstarktyz{|xwvutsrqdcji|{zyghefcdabopmnklij|{zypwfutsrqxwvuklijopmncda
bghefjilknmpobadcfehgyz{|unwhqrstuvwxwxuvstqrsdqj{|yzhgfedcbaponmlkjiefghabcdmnopij
klvuxwrqtsbqlszy|{nmpojilkfehgbadcarktyz{|uvwxqrsttcri|{zyxwvutsrqopmnklijghefcdaba
bcdefghijklmnoprqtsvuxwzy|{vmxgstqrwxuv{|yzopefdcbahgfelkjiponm{|yzoxevstqrwxuvlkji
ponmdcbahgfeijklmnopabcdefghzy|{vmxgrqtsvuxwxwvutsrqtcri|{zyghefcdabopmnklijfehgbad
cnmpojilkuvwxqrstarktyz{|mnopijklefghabcdbqlszy|{vuxwrqtscdij{|yzwxuvstqrponmlkjihg
fedcbabadcfehgjilknmpoqrstuvwxyz{|unwhtsrqxwvu|{zypwfucdabghefklijopmnzy|{jqdsrqtsv
uxwijklmnopabcdefghlkjiponmdcbahgfe{|yzslqbstqrwxuvuvwxqrstefopyz{|fehgbadcnmpojilk
ghefcdabopmnklijxwvutsrqhgnm|{zyponmlkjihgfedcbagxmv{|yzwxuvstqrvexozy|{vuxwrqtsmno
pijklefghabcdcdabghefklijopmntsrqxwvu|{zylkbaqrstuvwxyz{|ijcdbadcfehgjilknmpoyz{|ij
cdqrstuvwxjilknmpobadcfehgklijopmncdabghef|{zylkbatsrqxwvuvuxwrqtsvexozy|{efghabcdm
nopijklhgfedcbaponmlkjiwxuvstqrgxmv{|yzopmnklijghefcdabhgnm|{zyxwvutsrqefopyz{|uvwx
qrstnmpojilkfehgbadcdcbahgfelkjiponmstqrwxuv{|yzslqbrqtsvuxwzy|{jqdsabcdefghijklmno
pghefcdabopmnklijxwvutsrqlkba|{zyuvwxqrstirctyz{|fehgbadcnmpojilklkjiponmdcbahgfe{|
yzgxmvstqrwxuvzy|{vexorqtsvuxwijklmnopabcdefghqrstuvwxyz{|ufwpbadcfehgjilknmpocdabg
hefklijopmntsrqxwvu|{zyhwnujqdszy|{vuxwrqtsmnopijklefghabcdponmlkjihgfedcbaslqb{|yz
wxuvstqrhgfedcbaponmlkjiwxuvstqrslqb{|yzvuxwrqtsjqdszy|{efghabcdmnopijklklijopmncda
bghef|{zyhgnmtsrqxwvuyz{|ufwpqrstuvwxjilknmpobadcfehgrqtsvuxwzy|{vexoabcdefghijklmn
opdcbahgfelkjiponmstqrwxuv{|yzgxmvirctyz{|uvwxqrstnmpojilkfehgbadcopmnklijghefcdabt
kra|{zyxwvutsrqefghabcdmnopijklvuxwrqtsnmhgzy|{wxuvstqropef{|yzhgfedcbaponmlkjijilk
nmpobadcfehgyz{|arktqrstuvwx|{zytcritsrqxwvuklijopmncdabghefstqrwxuv{|yzcdijdcbahgf
elkjiponmabcdefghijklmnoprqtsvuxwzy|{balkpwfu|{zyxwvutsrqopmnklijghefcdabnmpojilkfe
hgbadcunwhyz{|uvwxqrstfehgbadcnmpojilkuvwxqrstunwhyz{|xwvutsrqpwfu|{zyghefcdabopmnk
lijijklmnopabcdefghzy|{balkrqtsvuxw{|yzcdijstqrwxuvlkjiponmdcbahgfetsrqxwvu|{zytcri
cdabghefklijopmnbadcfehgjilknmpoqrstuvwxyz{|arktopef{|yzwxuvstqrponmlkjihgfedcbamno
pijklefghabcdnmhgzy|{vuxwrqts
```