Address Hashing in Intel Processors

John D. McCalpin, PhD
mccalpin@tacc.utexas.edu
IXPUG Fall Conference, 2018-09-25

Outline & Caveats

- This is a stupidly complex and advanced topic
- What I want to get across today:

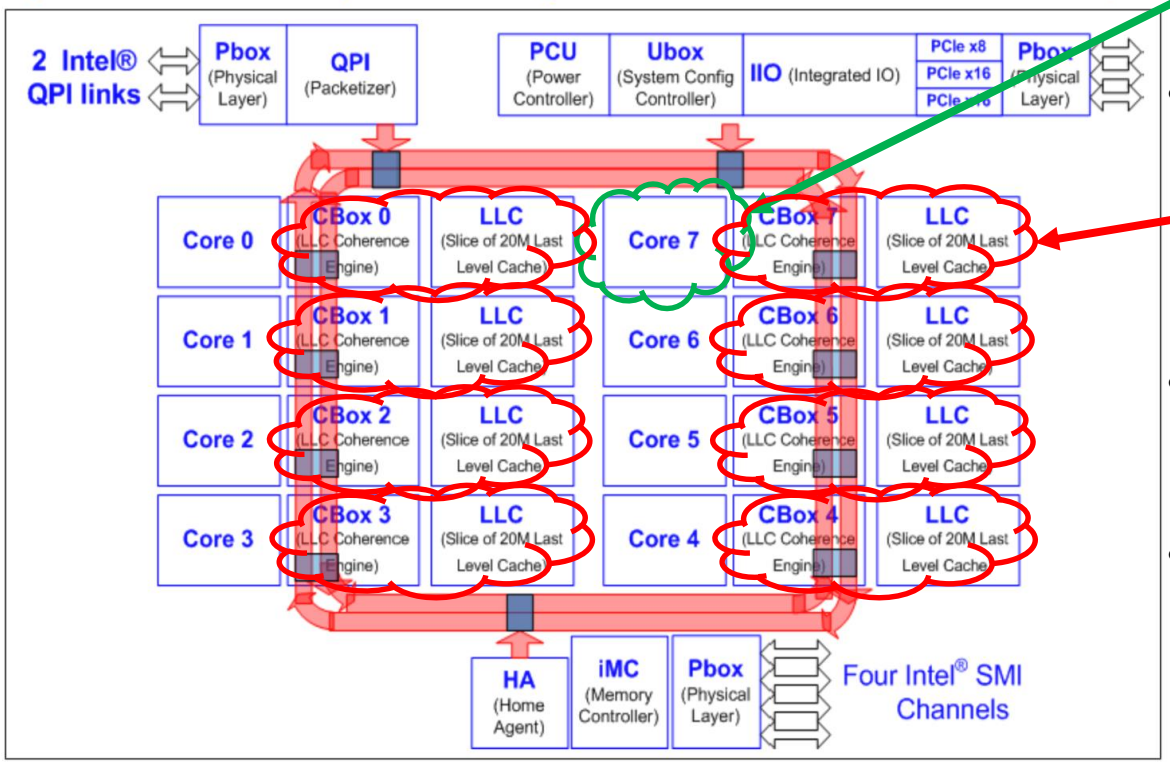
1. Address Hashing is required with distributed caches
2. The address hashes used in Intel processors are undocumented, but some properties can be derived
3. Non-power-of-2 hashing is hard, and as processors get more cores & caches, more “surprises” occur
4. Software mitigations are possible, but won’t happen without Intel’s help

Part 1: Background

WHAT IS “ADDRESS HASHING” AND WHY IS IT USED?

Review: **distributed**, **shared** L3 cache

Figure 1-1. Uncore Sub-system Block Diagram of Intel Xeon Processor E5-2600 Family



- Each “Core” box includes a core and its private L1 and L2 caches
- Each cache line address is “owned” by exactly one CBox unit and cached in exactly one L3 “slice” using an **undocumented** mapping
- **KEY 1**: Consecutive cache lines are mapped to all CBox units in many permutations
- **KEY 2:** each CBox/LLC only needs to handle the average traffic from one core.

(Figure from Intel document 327043-001)

Distributed, Shared, and ***Inclusive***

- "Inclusive" (pre-SKX processors) is a bit more subtle
  - **All** cache lines that are in **any** L1 Instruction Cache, **any** L1 Data Cache, or **any** L2 Cache on the chip, **must also** have a valid entry in the shared L3 cache.
- Implications:
  - **KEY 3:** An L3 cache miss guarantees that the cache line is not in any L1 or L2 cache on the chip, so there is no need to check those caches
  - Lines chosen as "victims" in the L3 cache **must be evicted from all L1 and L2 caches** before the L3 entry can be released and re-used
    - Rare (???) in practice due to L3's large size and high associativity

What about SKX?

- Intel Xeon Scalable Processors (Skylake Xeon) have a shared L3 cache that is **not** inclusive.
- The non-inclusive L3 cache is augmented by a *"Snoop Filter"* that tracks cache lines held in L1 and/or L2 caches
  - The Snoop Filter is **inclusive** – all lines held in any L1 or L2 must have a valid entry in the Snoop Filter
  - Any entry victimized from the Snoop Filter **must be evicted from all L1 and L2 caches** on the chip before freeing the Snoop Filter entry
- Basic mechanisms are the same – sizes have changed
  - Quantitative differences can lead to qualitative changes (Part 3)

Part 2

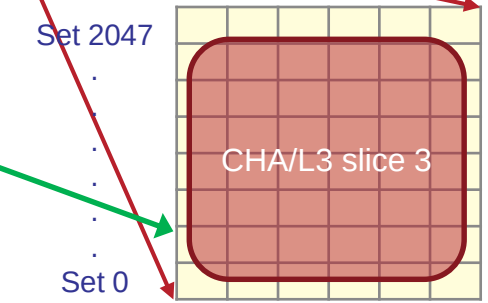
ADDRESS HASH PROPERTIES

Address Hashing components

- Physical Addresses must be mapped to one of the CHA/L3 “slices” on the chip, AND mapped to a “set” within that slice
- “***Slice Hash***” (or “Slice Select”) is the mapping of physical addresses to CHA/L3 “slices” (numbered 0..N-1)
- “***Set Hash***” is the mapping of physical addresses to “sets” within a slice.
- The “***Slice Hash***” will be my first topic, since it is relatively easy to measure (using hardware performance counters)

28c SKX CHA locations

UPI	PCIe	PCIe	Rlink	UPI	PCIe
0	4	9	14	19	24
IMC	5	10	15	20	IMC
1	6	11	16	21	25
2	7	12	17	22	26
3	8	13	18	23	27



Measuring the Slice Select by Address

- Use CHA performance counter event LLC_LOOKUPS.DATA_READ
- Allocate some 2MiB pages and use `/proc/[pid]/pagemap` to obtain the physical address for (the base of) each page
- For each of the 32,768 cache lines in a 2 MiB page:
  - Read the counter value on all CHAs
  - (load; fence; flush) 1000 times
  - Read the counter value on all CHAs
  - If exactly 1 CHA shows > 95% of expected accesses, record it and move to the next cache line address
    - Else repeat (up to 100 times before aborting)
- Save results in a file (name includes Physical Address of base of page)

How much to measure?

- Mapping a few hundred 2 MiB pages is enough to determine the high-level characteristics of the hash
- Deriving the *formula* for the slice select hash requires mapping a large fraction of the memory of a socket
  - E.g., >30,000 2 MiB pages for a socket with 96 GiB
  - At 7 seconds per 2 MiB page, this requires about 2.5 days
- I mapped a few hundred 2 MiB pages on 16c, 18c, 20c, 22c, 26c, 28c Xeon Scalable Processors (and KNL)
- I mapped >1/2 of memory on 14c and 24c processors
  - 1 billion (2^{30}) measurements on the 24c processors

Block sizes to obtain uniform distributions

- How many cache lines are required in a block to obtain *asymptotically uniform distribution* across the slices?
- On the 24-core Xeon Platinum 8160, every naturally-aligned block of 512 cache lines assigns
  - 21 cache lines to each of slices 0-15
  - 22 cache lines to each of slices 16-23

Non-Uniformity of Distribution across Slices

- On the 24-core Xeon Platinum 8160, every naturally-aligned block of 512 cache lines assigns
  - 21 cache lines to each of of slices 0-15
  - 22 cache lines to each of slices 16-23
- 512 lines / 24 slices = 21.3333 (uniform distribution)
- *Fraction lost* = 22 / 21.3333 – 1 = 3.125%
- Because they are assigned more addresses, slices 16-23 *must* start overflowing when you get to within ~3% of the nominal (aggregate) L3 capacity

How many permutations are observed?

- For 24 slices, there are > Avogadro's Number of possible permutations ($24! = 6.2 \times 10^{23}$)

- On the Xeon Platinum 8160, I collected 2 million (2^{21}) 512-line (aligned) sequences of slice assignments, but only 512 *unique* sequences were found

- The observed sequences have a distinctive structure…

Permutation Structure

- All 512 unique sequences are “binary permutations” of each other
  - This explains why the same slices are overloaded in all sequences – if the number “23” occurs 22 times in the base sequence, it will occur 22 times in every permutation of that sequence
- What is a “*binary permutation*”?
  - Let s[0..n-1] be the vector of slice assignments of the block starting at address zero, then

$$p_n[i] = s[i \oplus n]$$

So what does

$p_n[i] = s[i \oplus n]$

actually mean?

Start with base Slice Select Sequence s(n) for 4 slices

0	1	2	3

Permutation p_0: Identity Operator

0	1	2	3

Permutation p_1: swap elements within aligned pairs

1	0	3	2

Permutation p_2: swap pairs within aligned quads

2	3	0	1

Permutation p_3: perform both swaps

3	2	1	0

$p_{n/2}$ swaps the first and last halves
p_{n-1} reverses the order of the sequence

How are the permutations selected?

- A binary permutation of length 2^n has n 1-bit equations
  - Each is an XOR-reduction of a *different subset* of the high-order address bits
- Simple (made-up) example for n=4 (16-line sequences)
  - Let a_i be bit i of the physical address, and b_j be bit j of the permutation number
    - $b_0 = a_{18} \oplus a_{17} \oplus a_{14} \oplus a_{11} \oplus a_{10}$
    - $b_1 = a_{19} \oplus a_{18} \oplus a_{15} \oplus a_{12} \oplus a_{11}$
    - $b_2 = a_{19} \oplus a_{16} \oplus a_{13} \oplus a_{12}$
    - $b_3 = a_{17} \oplus a_{14} \oplus a_{13}$
- These are not documented, and must be derived by measurement, but all processors seem to use this approach

Address Hash Properties by Core Count

	14c	16c	18c	20c	22c	24c	26c	28c	KNL
# cache lines to get uniform distribution	16384	16	512	256	16384	512	16384	4096	2048
#permutations observed	2048	16	2048	256	*4096?*	512	*8192?*	*4096?*	4096
Fraction lost	2.54%	0.0%	1.95%	1.56%	3.66%	3.13%	2.20%	2.54%	7.62%

- Values in black have been studied extensively
- Insufficient data was collected to provide definitive results on some systems with longer-than-expected sequence lengths
- Models used: Xeon Gold 6132, 6142, 6150, 6148, 6152, Xeon Platinum 8160, 8170, 8180, and Xeon Phi 7250

Address Hash Properties by Core Count

	14c	16c	18c	20c	22c	24c	26c	28c	KNL
# cache lines to get uniform distribution	16384	16	512	256	16384	512	16384	4096	2048
#permutations observed	2048	16	2048	256	*4096?*	512	*8192?*	*4096?*	4096
Fraction lost	2.54%	0.0%	1.95%	1.56%	3.66%	3.13%	2.20%	2.54%	7.62%

- 16c is trivial – every group of 16 cache line addresses has an allocation pattern that is one of the 16 binary permutations of the set 0,1,2,3,…,15
- All the others are quite complex!
  - Details for 24c are in Appendix
  - NOTE: KNL uses all 38 CHAs, even if co-located cores are disabled!

Xeon Scalable Processors and Xeon Phi x100

Number of cache lines allocated to each CHA in each (aligned) sequence of the length specified in the bottom row

(Note that each row represents 2 CHAs to make the table easier to read)

Color-coding indicates CHAs with most (orange), medium (yellow), or least (blue) lines allocated

There is a lot of common structure in these patterns…

	14c	16c	18c	20c	22c	24c	26c	28c	KNL
CHA 0-1	1192	1	29	13	772	21	638	149	52
CHA 2-3									58
CHA 4-5									52
CHA 6-7									58
CHA 8-9	1200								52
CHA 10-11									58
CHA 12-13	1024								52
CHA 14-15									58
CHA 16-17			24	12	752	22	644	150	52
CHA 18-19									58
CHA 20-21					512				52
CHA 22-23									58
CHA 24-25							512	128	52
CHA 26-27									58
CHA 28-29									52
CHA 30-31									58
CHA 32-33									48
CHA 34-35									48
CHA 37-37									48
Seq Length	*16384*	*16*	*512*	*256*	*16384*	*512*	*16384*	*4096*	*2048*

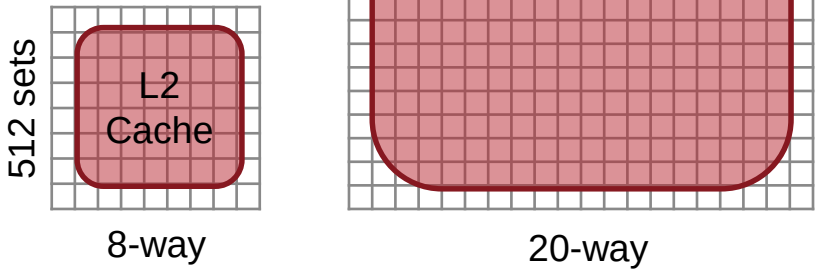
Part 3

ADDRESS HASH CONFLICTS

A change in ratios

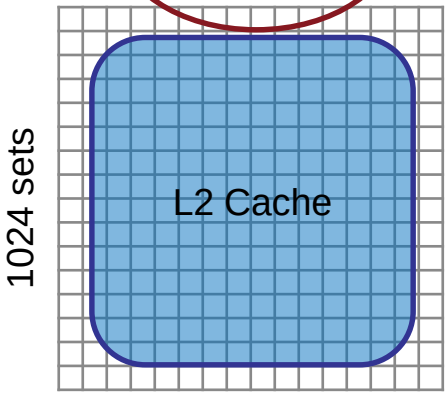
Xeon E5 v3:
small L2 included in large L3
4x #sets
2.5x associativity

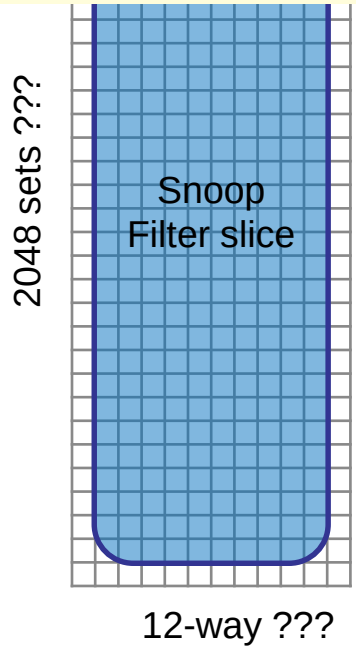
12 L2's = 2 2MiB pages



Xeon Platinum
large L2 tracked by Snoop Filter
2x #sets
??? associativity?

24 L2's = 12 2MiB pages





SNOOP FILTER ORGANIZATION AND SIZE IS NOT DOCUMENTED!

Snoop Filter Properties

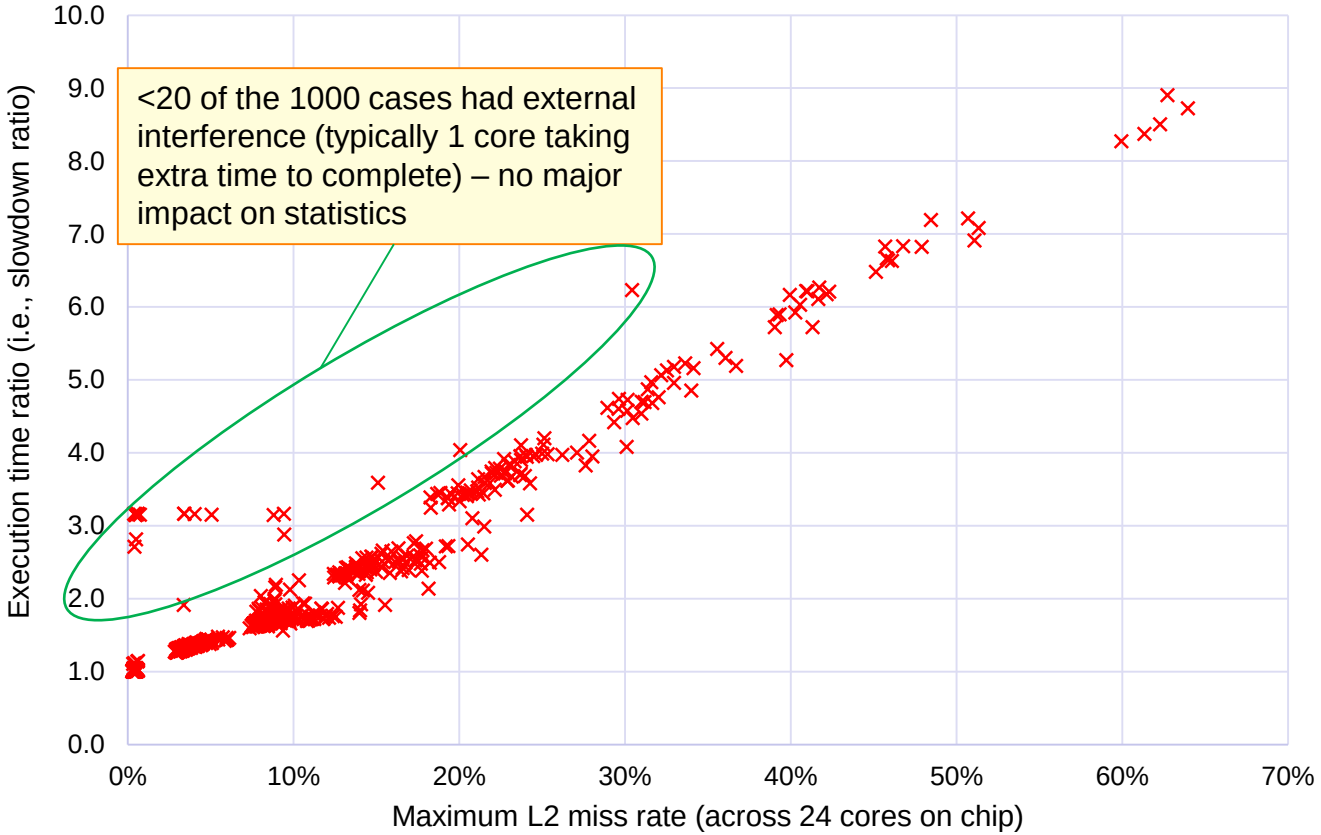
- The Snoop Filter is distributed and shared, using the same address hash as the L3 cache
- The Snoop Filter has enough capacity to track all lines in the L1 and L2 caches – but only if the addresses are distributed “uniformly enough” over the available entries
  - Each Snoop Filter can track more than its “fair share”: 1 MiB (=1/28 of 28 1MiB caches), but cannot track all 28 MiB if all cores load only cache lines mapping to one “slice”
- Conflicts can occur: evicting “live” data from L2 caches

Snoop Filter Conflict Demo: 2MiB pages

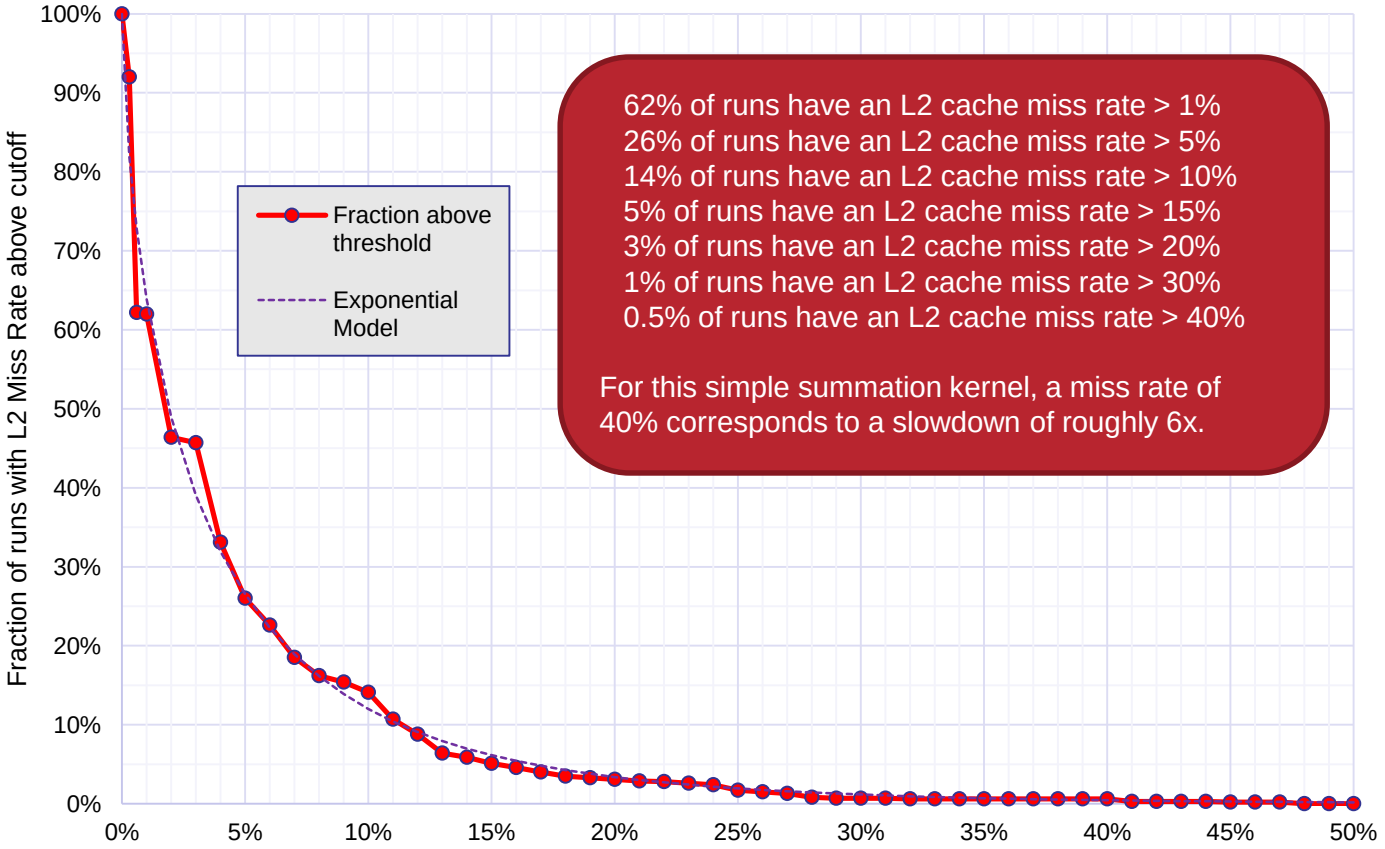
1. Create an array that is 95% of the size of the total L2 cache
2. Load the array into the L2 cache
3. Read the performance counters
4. Sum the array 1000 times
5. Read the performance counters & print results

- Run the code at least 1000 times (different executions get different physical addresses assigned)
- Expected L2 miss rate is 0%
- Actual L2 miss rate is sometimes not 0%
- When L2 miss rate is not zero, the L2 miss count matches the Snoop Filter Eviction count

Xeon Platinum 8160: 95% L2 Read Time vs Max L2 Miss Rate: 1000 trials on 2MiB pages

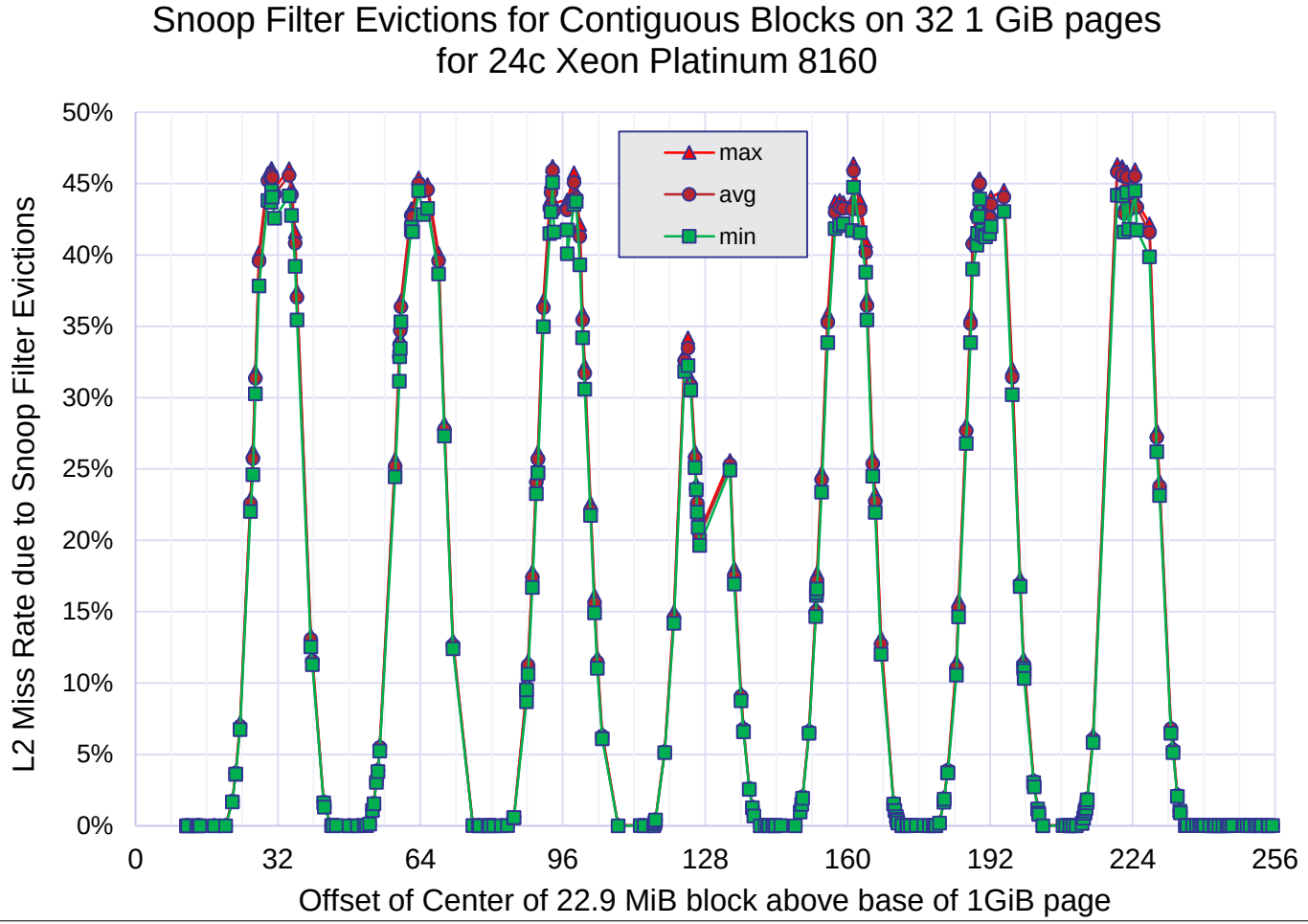


Xeon Platinum 8160: Cumulative Probability Function of L2 Miss Rates for 1000 trials of repeated summation test for 22.9MiB array



Snoop Filter Conflict Demo: 1GiB pages

- 1 GiB pages provide contiguous Physical Addresses for L2-containable arrays
- Methodology similar to 2MiB case, but
  - No ensembles needed (no higher-order address bit changes in array)
  - Start of the array was shifted within the 1 GiB page
- Repeated on 32-64 different 1GiB pages for each offset
  - Results were the same on every 1GiB page
  - The pattern repeats 4 times in every 1GiB page (every 256 MiB)
- Conflicts on contiguous addresses give hints about Set Hash
  - The details are left as an exercise to the reader….


Snoop Filter Evictions for Contiguous Blocks on 32 1 GiB pages
for 24c Xeon Platinum 8160
L2 Miss Rate due to Snoop Filter Evictions
50%
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
max
avg
min
0
32
64
96
128
160
192
224
256
Offset of Center of 22.9 MiB block above base of 1GiB page

Snoop Filter Conflict Summary for 95% L2 read test

	14c	16c	18c	20c	22c	24c	26c	28c	KNL*
Max L2 miss rate on 1 GiB pages	0.0%	0.0%	23.5%	11.1%	1.3%	46.1%	0.1%	0.0%	25.2%

*KNL core L2_REQUESTS.MISS undercounts, use CHA or IMC events

Snoop Filter Conflict Summary for 95% L2 read test

	14c	16c	18c	20c	22c	24c	26c	28c	KNL*
Max L2 miss rate on 1 GiB pages	0.0%	0.0%	23.5%	11.1%	1.3%	46.1%	0.1%	0.0%	25.2%
Max L2 miss rate on 2 MiB pages (1000 trials)	0.0%	0.0%	35.1%	38.6%	13.2%	43.4%	12.2%	15.7%	50.6%

*KNL core L2_REQUESTS.MISS undercounts, use CHA or IMC events

Snoop Filter Conflict Summary for 95% L2 read test

	14c	16c	18c	20c	22c	24c	26c	28c	KNL*
Max L2 miss rate on 1 GiB pages	0.0%	0.0%	23.5%	11.1%	1.3%	46.1%	0.1%	0.0%	25.2%
Max L2 miss rate on 2 MiB pages (1000 trials)	0.0%	0.0%	35.1%	38.6%	13.2%	43.4%	12.2%	15.7%	50.6%
Max L2 miss rate on 2 MiB pages (all trials)	0.0% (1000)	0.0% (10000)	45.2% (10000)	41.0% (20000)	15.2% (10000)	60.3% (20000)	25.0% (10000)	26.5% (6000)	62.1% (10000)

*KNL core L2_REQUESTS.MISS undercounts, use CHA or IMC events

Speculation

- Occasional Snoop Filter Conflicts with 2 MiB pages may be unavoidable with the current design
  - Filling 28 MiB of L2 cache requires 14 2 MiB pages, but the Snoop Filter may only be 11-way or 12-way associative?
- Snoop Filter Conflicts for contiguous physical address ranges should be avoidable
  - Impossible to evaluate without documentation of the class of hash functions in use….
  - Reducing the conflicts for contiguous physical addresses might improve the conflicts with 2 MiB pages?

Workload Implications

- The “surprise” of Snoop Filter Conflicts is expected to impact jobs with:
  - High L2/L3 cache re-use, AND
  - High L2/L3 access rate, AND
  - (if multi-node) Static load distribution with synchronization
- We saw it first with HPL (High Performance LINPACK)
  - About 1 of 200 single-node runs were $\geq 10\%$ slower than the median for that node
  - All runs using > 200 nodes had at least 1 slow node
- Currently collecting performance counter data across all jobs to look for occurrence in “real” applications….
  - We will review the data before SC18

Mitigations?

- In some cases, conflicts can be eliminated with 1 GiB pages
  - TACC's TOP500 HPL benchmark boosted by 28% (same HW)
  - Details in my SC18 presentation on November 13 in Dallas.
- Given formulas for the "Slice Hash" and "Set Hash", it should be possible to predict conflicts for groups of 2 MiB pages
  - This could be used to create a routine to sort the free page list to minimize conflicts for groups of ~10-30 2 MiB pages
    - E.g., `zonesort` on our KNL systems in MCDRAM Cache mode
  - Intel does not (yet) appear enthusiastic about such a project

Summary

- Address Hashes are a fundamental piece of Intel manycore processor designs
- The hashes are undocumented, but can be studied
- At high core counts, hash conflicts occur for “bad” combinations of physical addresses on 2 MiB pages, causing unexpected eviction of data from L2 caches
  - Occurrence can be rare and difficult to identify
  - Software workarounds may be possible
- Additional “surprises” occur when attempting to re-use data held in the aggregate L2+L3 cache – analysis in progress….

Appendix 1

SLICE SELECT EQUATIONS FOR XEON PLATINUM 8160 (24-CORE)

Slice Hash on Intel Xeon Platinum 8160

- The 24-core Xeon Platinum 8160 has a 512-element base slice select sequence
- That sequence is permuted by a 9-bit binary permutation operator computed by XORing the address bits selected by the “1” bits in the table:

Select bit	bit 36	bit 35	bit 34	bit 33	bit 32	bit 31	bit 30	bit 29	bit 28	bit 27	bit 26	bit 25	bit 24	bit 23	bit 22	bit 21	bit 20	bit 19	bit 18	bit 17	bit 16	bit 15
8	1	0	1	0	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1	0	0	1
7	1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0	1	0	1	1	0	1
6	1	1	0	1	0	0	0	0	0	1	0	1	1	1	0	0	0	1	1	1	1	1
5	1	1	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	0	0	1	1	0
4	1	1	1	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	0	0	1	1
3	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1	0	0	1	1	0	0	0
2	0	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1	0	0	1	1	0	0
1	1	0	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1	0	0	1	1	0
0	0	1	0	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1	0	0	1	1

Slice Hash on Intel Xeon Platinum 8160

An alternate approach is to AND the physical address with each of 9 “masks”, compute a popcount on the result, and use the parity of the popcount as the corresponding permutation select bit:

```
Bit 8 = (popcountq(ADDR & 0x15b9648000))%2;
Bit 7 = (popcountq(ADDR & 0x1f65d68000))%2;
Bit 6 = (popcountq(ADDR & 0x1a0b8f8000))%2;
Bit 5 = (popcountq(ADDR & 0x18bca30000))%2;
Bit 4 = (popcountq(ADDR & 0x1c5e518000))%2;
Bit 3 = (popcountq(ADDR & 0x1b964c0000))%2;
Bit 2 = (popcountq(ADDR & 0x0dcb260000))%2;
Bit 1 = (popcountq(ADDR & 0x16e5930000))%2;
Bit 0 = (popcountq(ADDR & 0x0b72c98000))%2;
```

These masks are valid for memory addresses up to 96 GiB (maybe 128?) – more testing would be required to determine the higher-order bits in use for larger memory sizes.

Xeon Platinum 8160 Base Slice Select Sequence

- The 512-character sequence below (8 lines of 64 characters per line) is the base slice select sequence for the 24-core Xeon Platinum 8160 processor
- In the sequence, the letter “a” represents CHA/L3 number 0, the letter “b” represents CHA/L2 number 1, etc., up to “x” representing CHA/L3 number 23.

```
adkjhunwfgpmcritbcligvmxexovdqjsadkjxevovgxmsbqlbcliwfupuhwntark
onehrktalibcupwfpmfgqlsbsjqdvoxeonehjsdqlqbsmxgvpmfgitcrkratnwhu
itcrpmfgnwhukjadjsdqonehmxgvlqbsqlsbpmfgvoxesjqdrktaonehupwftirc
wfupbclitarkehonxevoadkjsbqlvgxmgvmxbclidqjsexovhunwadkjcritfwpu
bslqgfmpexovdajkatkrhunwfwpucbilrctiwfupuhwndajkqdsjxevovgxmcbil
xmvgilcbsjqdnohewnuhrktatircmpgfpufwitcrkratnoheovexjsdqlqbsmpgf
jkdaovexmpgflqbsitcrpufwnohekratrktawnuhmpgftircqlsbxmvgnohesjqd
henoqdsjcbilvgxmwfuprctidajkuhwnhunwatkrcbilfwpugvmxbslqdajkexov
```

Appendix 2

ASIDE: WHY NOT USE 4K PAGES FOR L2-CONTAINED DATA?

Address bits for accessing a 4 MiB array of doubles with 4 KiB pages

	47	…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual	**Base address plus 8 * array index**																						0	0	0

Address bits for accessing a 4 MiB array of doubles with 4 KiB pages

	47	…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual	Base address plus 8 * array index																						0	0	0
Virtual	**Base address + 512 * array index**													**Cache line number within 4 KiB page**						**Byte offset within cache line**					

Address bits for accessing a 4 MiB array of doubles with 4 KiB pages

	47	…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual	Base address plus 8 * array index																						0	0	0
Virtual	Base address + 512 * array index													Cache line number within 4 KiB page						Byte offset within cache line					
Physical	**Pseudo-Random 4KiB Physical Page Address**													Cache line number within 4 KiB page						Byte offset within cache line					

Address bits for accessing a 4 MiB array of doubles with 4 KiB pages

	47	…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual	Base address plus 8 * array index																						0	0	0
Virtual	Base address + 512 * array index													Cache line number within 4 KiB page						Byte offset within cache line					
Physical	Pseudo-Random 4KiB Physical Page Address													Cache line number within 4 KiB page						Byte offset within cache line					
L2 Cache											**L2 Cache Index** (Sandy Bridge - Broadwell)									Byte offset within cache line					

Address bits for accessing a 4 MiB array of doubles with 4 KiB pages

	47	…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual	Base address plus 8 * array index																						0	0	0
Virtual	Base address + 512 * array index													Cache line number within 4 KiB page						Byte offset within cache line					
Physical	Pseudo-Random 4KiB Physical Page Address													Cache line number within 4 KiB page						Byte offset within cache line					
L2 Cache											L2 Cache Index (Sandy Bridge - Broadwell)														
L2 Cache											**Page color**			Cache line number within 4 KiB page											

Consider repeated accesses to an 80 KiB array
→ pages 0..19 →

The contiguous virtual addresses are mapped to pseudo-random 4KiB page physical addresses

3 bits above the 4KiB page boundary determine the “page color”

Each box is labelled with its virtual page number and colored by its “page color”

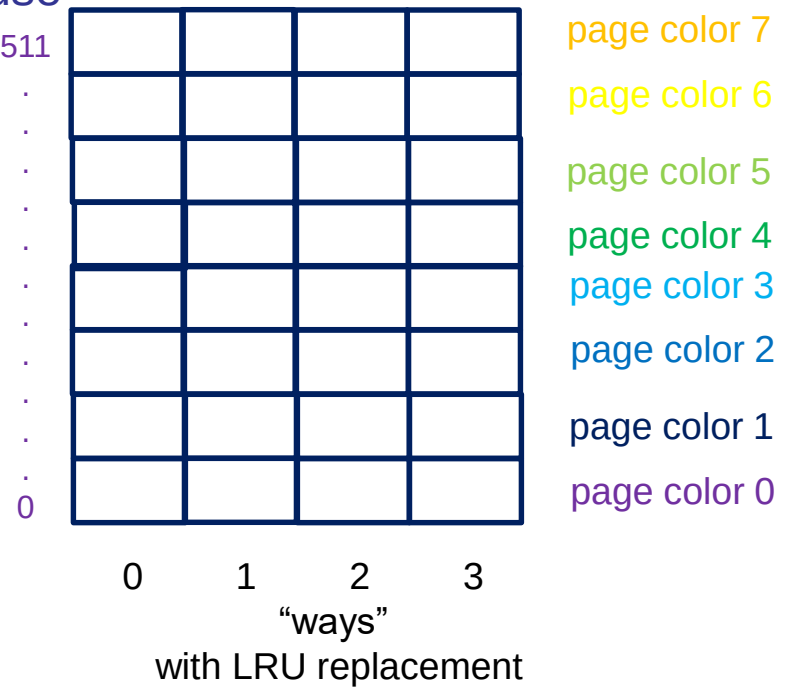
		1st	2nd
2	19	Miss	Hit
0	18	Miss	Hit
1	17	Miss	Miss – evict 6
1	16	Miss	Miss – evict 17
6	15	Miss	Hit
2	14	Miss	Hit
4	13	Miss	Hit
5	12	Miss	Hit
3	11	Miss	Hit
5	10	Miss	Hit
1	9	Miss	Miss – evict 16
1	8	Miss	Miss – evict 9
4	7	Miss	Hit
1	6	Miss	Miss - evict 8
0	5	Miss	Hit
4	4	Miss	Hit
4	3	Miss	Hit
5	2	Miss	Hit
0	1	Miss	Hit
7	0	Miss	Hit

Attempting to use 62.5% of the cache and still only getting a 75% hit rate!

L2 cache indices **eviction**

Example L2 cache indexing

128 KiB, 4-way set-associative
128 KiB / 4-way = 32KiB of addresses
32 KiB = 8 pages = 8 colors



Xeon Platinum 8160: 95% L2 Read test, max L2 Miss Rate, 1000 trials, 4KiB vs 2MiB pages

For both page sizes, execution time is most strongly correlated with the maximum L2 miss rate across all cores (r^2 ~ 0.94).

For 2MiB pages, L2 misses are almost all due to Snoop Filter Evictions (r^2 > 0.9999, slope = 0.999).

For 4KiB pages, L2 misses are dominated by page coloring conflicts (see Appendix 2), with only 30% of L2 Misses due to Snoop Filter Evictions.

Switch from 4KiB pages to 2MiB pages

	47	…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual	Base address plus 8 * array index																						0	0	0
Virtual	Base address + 512 * array index													Cache line number within 4 KiB page						Byte offset within cache line					
Physical	**Pseudo-Random 4KiB Physical Page Address**													Cache line number within 4 KiB page						Byte offset within cache line					
Physical	**Physical address**				**Cache line number within 2 MiB page**															Byte offset within cache line					

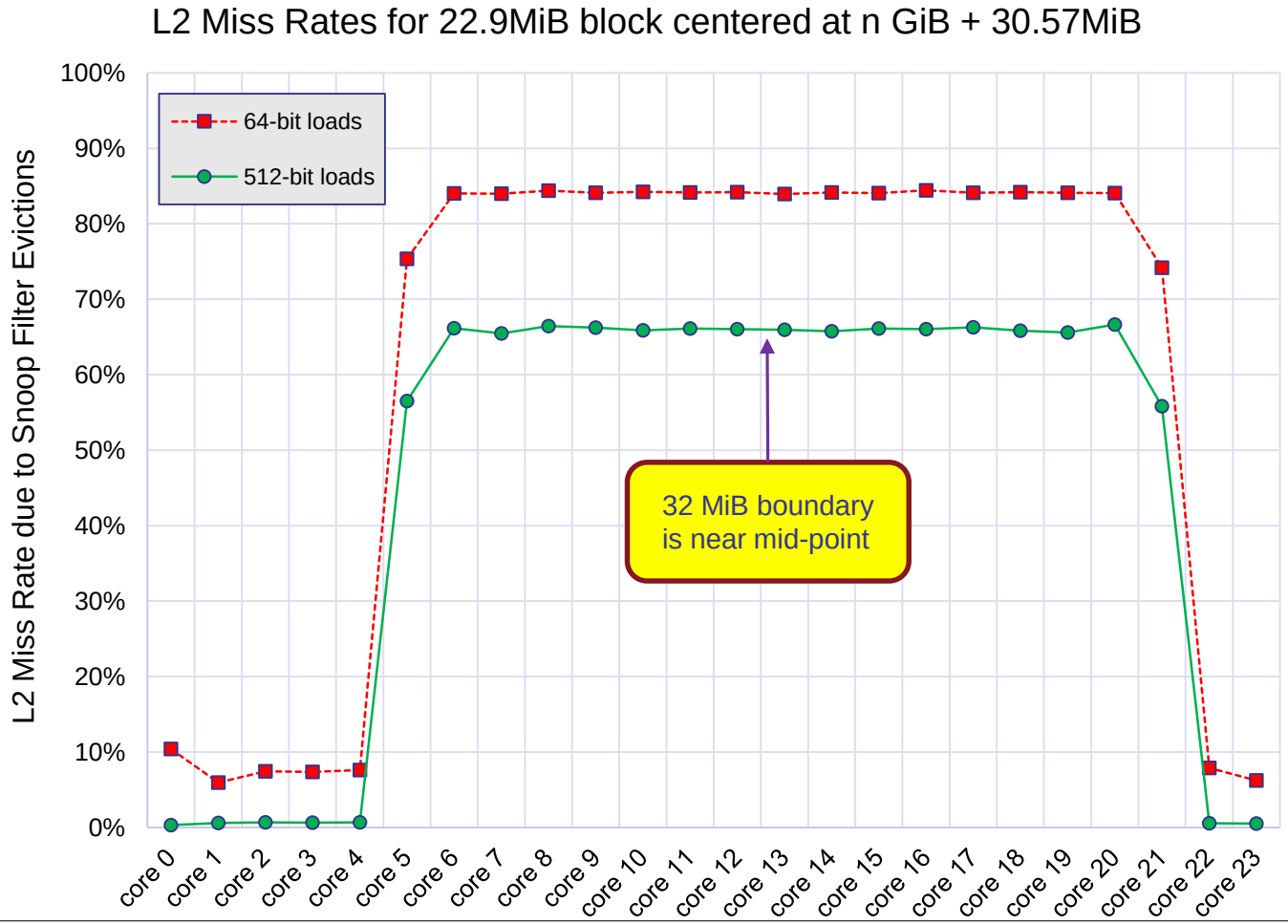
Switch from 4KiB pages to 2MiB pages

	47	…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual	Base address plus 8 * array index																						0	0	0
Virtual	Base address + 512 * array index													Cache line number within 4 KiB page						Byte offset within cache line					
Physical	**Pseudo-Random 4KiB Physical Page Address**													Cache line number within 4 KiB page						Byte offset within cache line					
Physical	**Physical address**				Cache line number within 2 MiB page															Byte offset within cache line					
L2 Cache	(not used)				(not used)						**L2 Cache Index**									Byte offset within cache line					

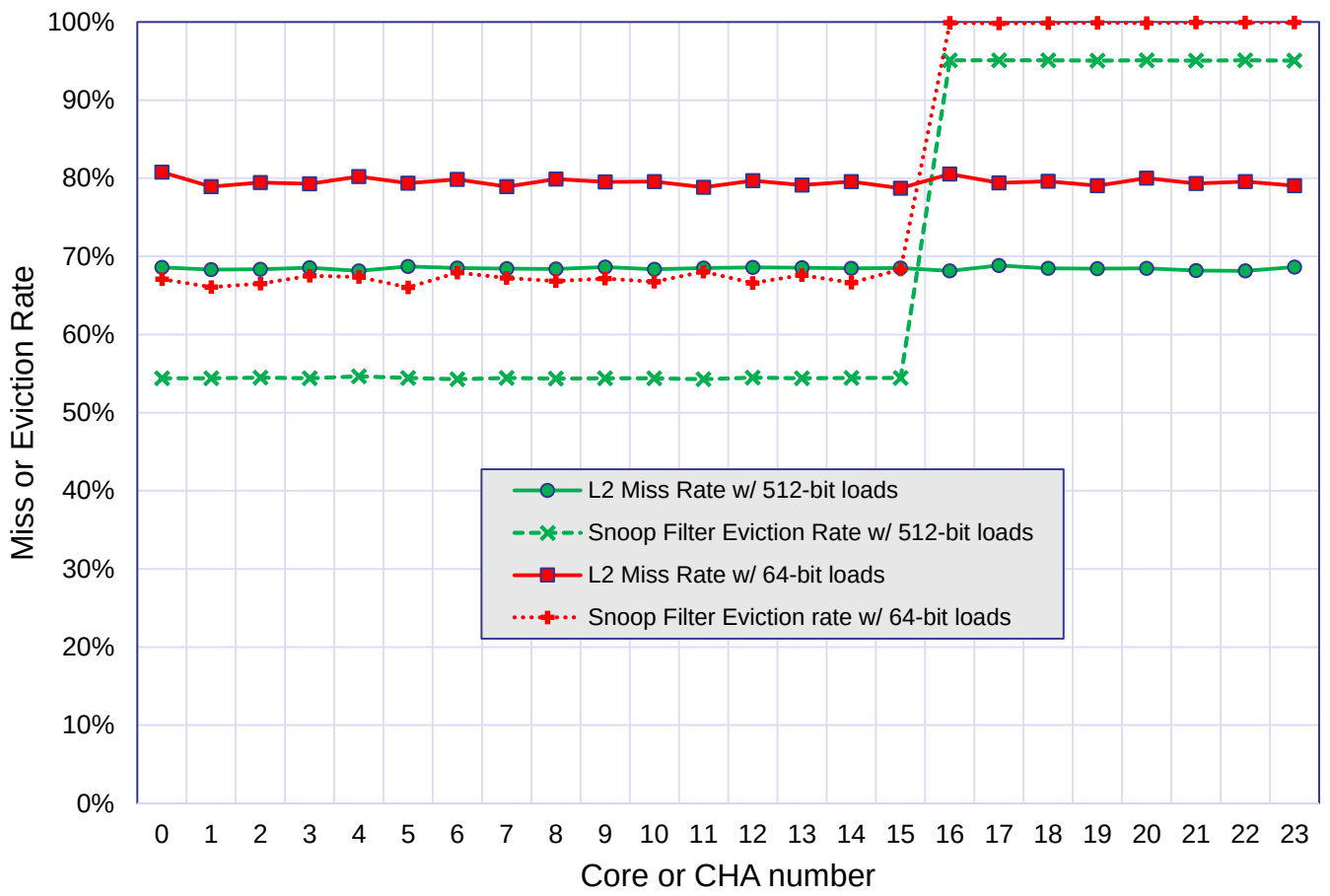
Appendix 3

ADDITIONAL RESULTS

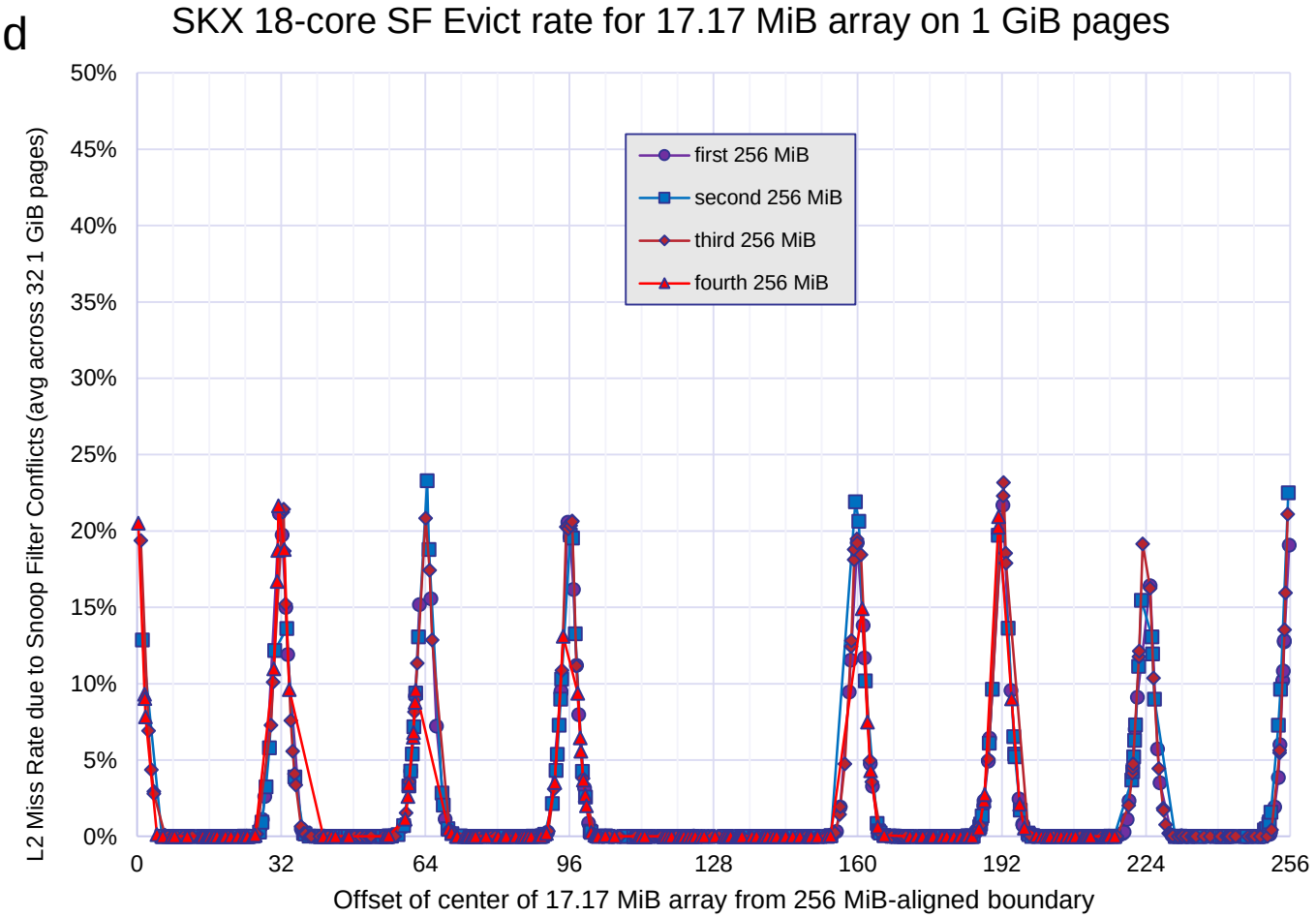
Xeon Platinum 8160



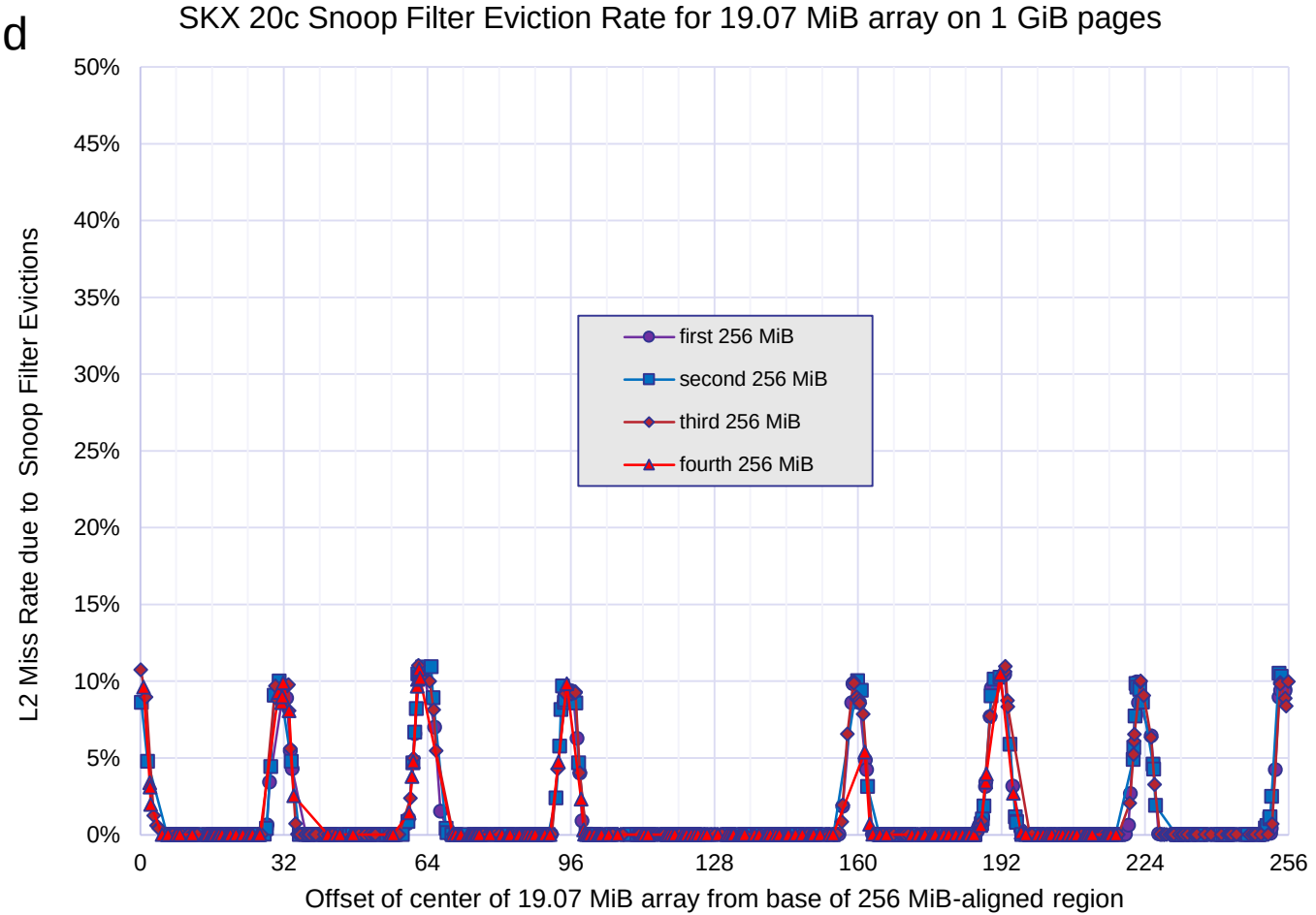
Xeon Platinum 8160



Xeon Gold 6150



Xeon Gold 6148



Spatial pattern of CHA allocations on Xeon Phi 7250

Background color indicates excess or deficit relative to uniform distribution

red +7.6%
beige -3.5%
green -10.9%

EDC	EDC	PCIe + DMI		EDC	EDC
CHA 0	CHA 12			CHA 13	CHA 29
CHA 4	CHA 16	CHA 28	CHA 1	CHA 17	CHA 33
CHA 8	CHA 20	CHA 32	CHA 5	CHA 21	CHA 37
IMC0	CHA 24	CHA 36	CHA 9	CHA 25	IMC1
CHA 2	CHA 14	CHA 26	CHA 3	CHA 15	CHA 27
CHA 6	CHA 18	CHA 30	CHA 7	CHA 19	CHA 31
CHA 10	CHA 22	CHA 34	CHA 11	CHA 23	CHA 35
EDC	EDC	MISC		EDC	EDC

Appendix 3

EXTRA SLIDES

References & Resources

- McCalpin's blog
  - https://sites.utexas.edu/jdm4372
- McCalpin's 2018-04-12 IXPUG Working Group presentation
  - https://www.ixpug.org/working-groups)
- McCalpin's GitHub projects, particularly:
  - https://github.com/jdmccalpin/SKX-SF-Conflicts
- "Mapping the Intel Last-Level Cache", Yarom, *et al.*
  - https://eprint.iacr.org/2015/905.pdf
- Intel Uncore Performance Monitoring manuals (for each generation)
- For sparse directories Lenoski & Weber, "Scalable Shared-Memory Multiprocessing", 1994 (ISBN-13 978-1493306145)
- Intel software developer forums

John D. McCalpin, PhD

mccalpin@tacc.utexas.edu

512-232-3754

For more information:

www.tacc.utexas.edu