OPTIMIZE YOUR PMDK APPLICATION'S PERFORMANCE WITH THE HELP OF INTEL® VTUNE™ AMPLIFIER PROFILER

Dmitry Ryabtsev

Sergey Vinogradov

Zhuowei Si

LEGAL DISCLAIMER & OPTIMIZATION NOTICE

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

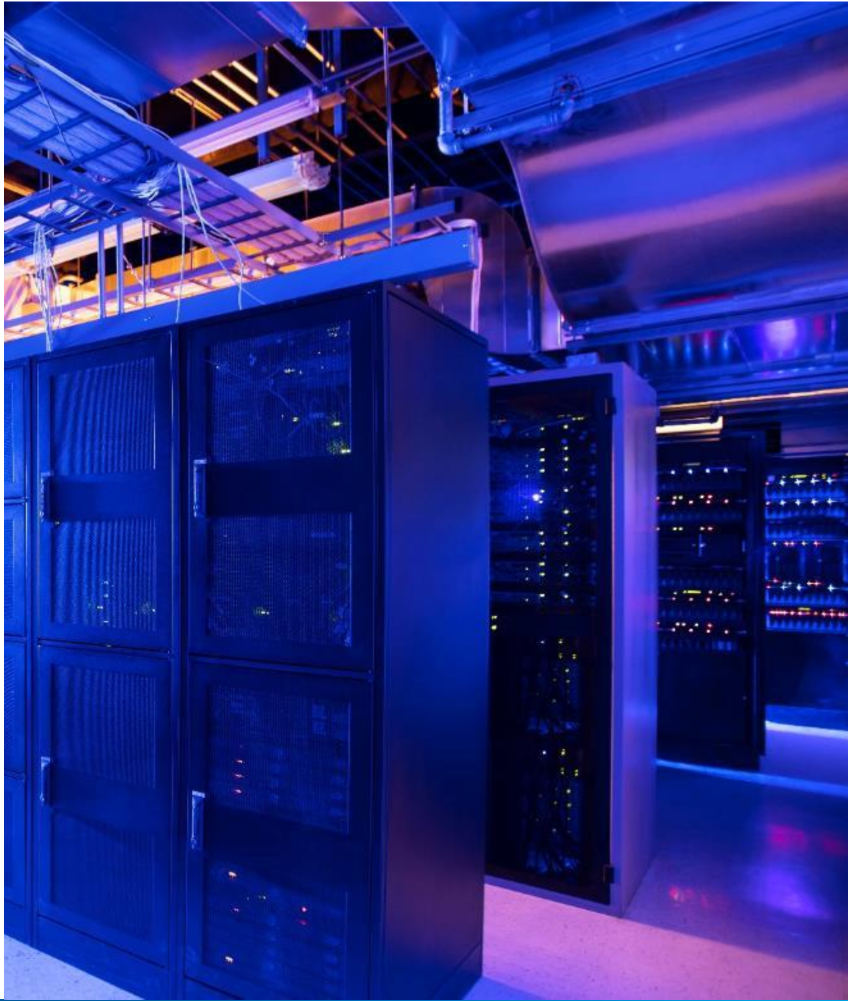
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

AGENDA

- Motivation
- Overview of VTune capabilities useful for PMEM profiling
  - Characterization
  - Data Profiling
  - Bandwidth monitoring
- Case Study: PMEMKV
- Case Study: Pelikan



Motivation

Intel® VTune™ Amplifier:

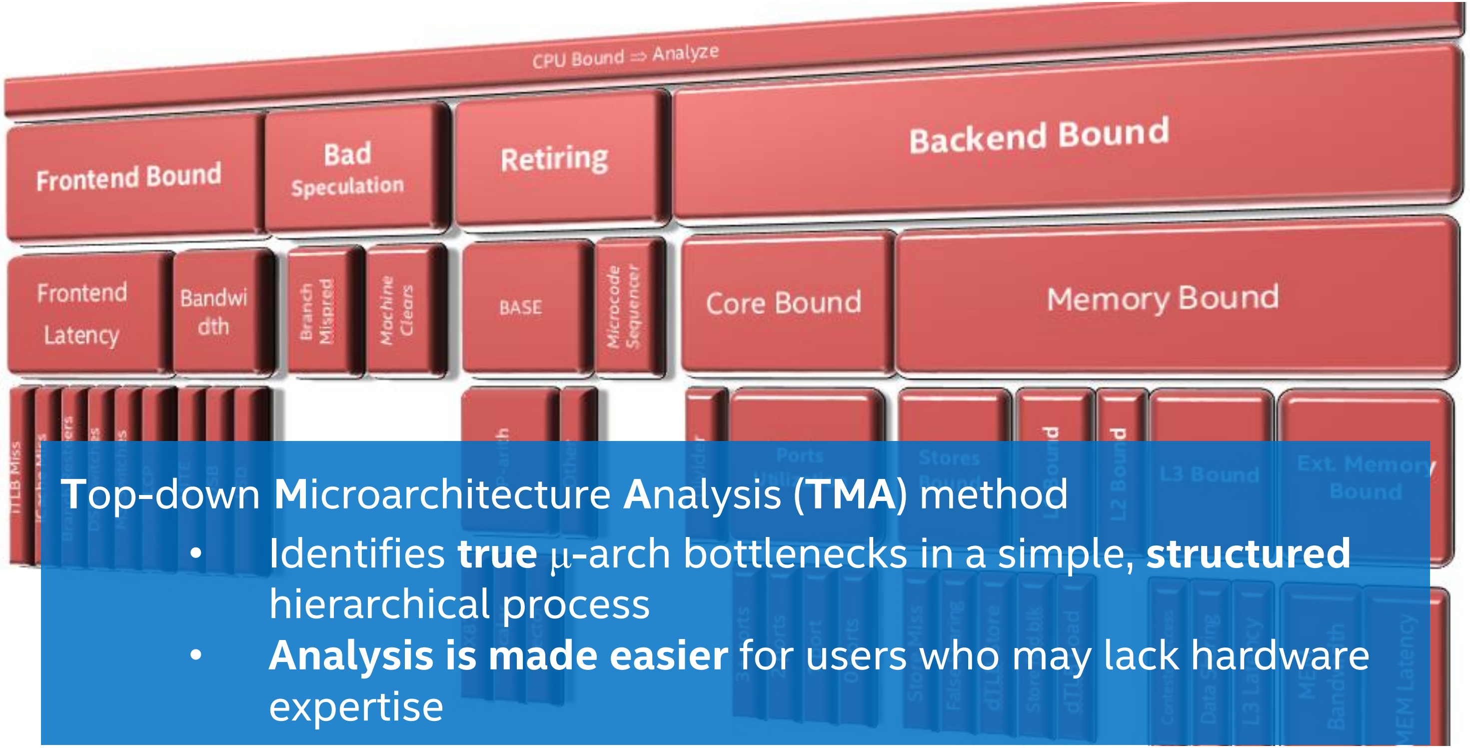
- Delivers insights for optimizing performance on modern Intel HW.
- Has specialized analyses for optimizing the use of memory.
  - A typical hotspot analysis shows code that is taking the most time.
  - **Memory Access** analysis lets you attribute performance events to memory objects.
- Support Intel® Optane™ DC PM specific performance events.

MAIN VTUNE CAPABILITIES FOR PMEM

3 key ingredients:

- **Application characterization from uArch perspective**
  - Is accessing memory really a bottleneck? How big it is?
- **Bandwidth analysis**
  - Is my workload approaching memory bandwidth limits? What code is responsible?
- **Data profiling**
  - Accessing what data structures is the most problematic?

APPLICATION CHARACTERIZATION FROM UARCH PERSPECTIVE



Top-down **M**icroarchitecture **A**nalysis (**TMA**) method

- Identifies **true** μ-arch bottlenecks in a simple, **structured** hierarchical process
- **Analysis is made easier** for users who may lack hardware expertise

Optimization Notice

APPLICATION CHARACTERIZATION FROM UARCH PERSPECTIVE

New metric in TMA: **“Persistent Memory Bound”**

Shows percentage of cycles CPU stalled while waiting from PMEM

Complements existing Memory Bound, DRAM Bound, etc. metrics

Memory Bound:	75.0%	of Pipeline Slots
L1 Bound:	1.4%	of Clockticks
L2 Bound:	0.1%	of Clockticks
L3 Bound:	3.1%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Persistent Memory Bound:	57.9%	of Clockticks
Store Bound:	7.7%	of Clockticks

DATA PROFILING

Attributes performance metrics not only to the code but to the **data structures** as well

Works by:

- Instrumenting memory API
  VTune provide the ***instrumentation and tracing technology (ITT)*** *APIs to trace custom memory allocation API*
- Capturing data address for PEBS facility on each memory event sample
- Correlating above two things

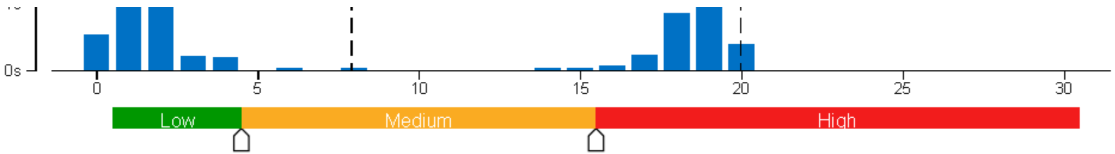
Latest VTune version supports PMDK memory allocation API

Optimization Notice

BANDWIDTH ANALYSIS

VTune is able to measure both regular DRAM and PMEM bandwidth. Based on uncore events.

Automatically correlates with the code running at the same time – easy to see who is responsible for high bandwidth consumption



Top Memory Objects with High Bandwidth Utilization

This section shows top memory objects, sorted by LLC Misses, that were accessed when bandwidth utilization was high for the domain selected in the histogram area.

Memory Object	LLC Miss Count
array.c:95 (366 MB)	37.4%
array.c:132 (366 MB)	36.0%
array.c:134 (366 MB)	15.8%

Optimization Notice

CASE STUDY

PMEMKV: FILLRANDOM BENCHMARK ON STREE ENGINE

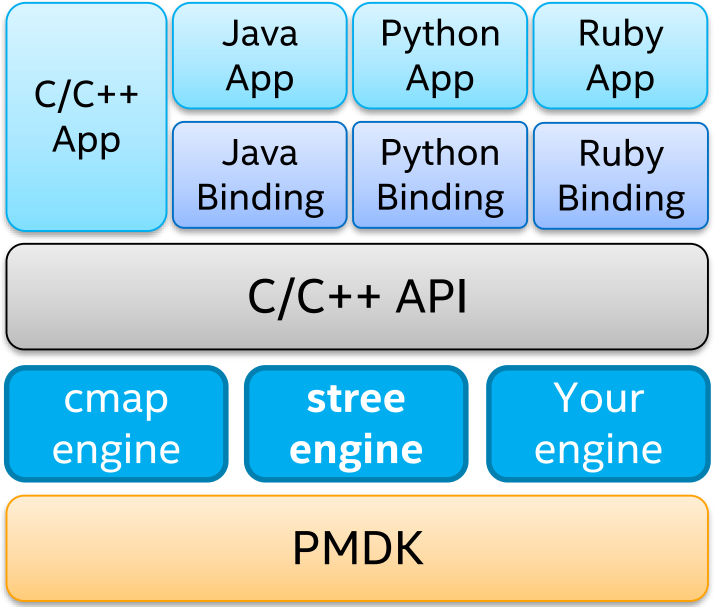
PMEMKV

- Embedded Key/Value storage optimized for persistent memory.
- **pmemkv_bench** – db_bench ported from RocksDB.

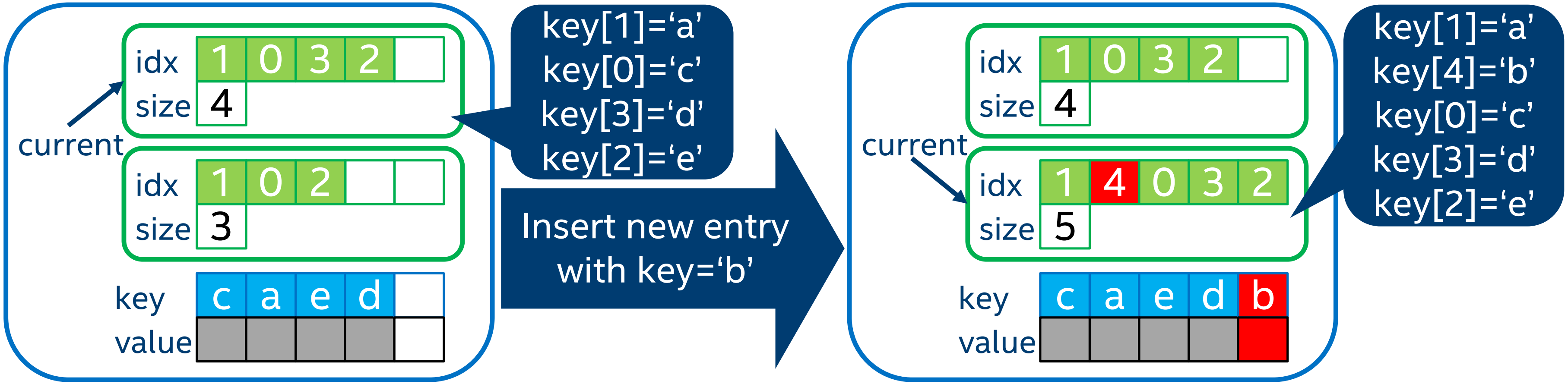
Benchmark

- Fillrandom benchmark using stree engine.
- 4 000 000 entries inserted.
- Key size -16 bytes, Value size – 200 bytes.

```
pmemkv_bench --engine=stree --num=4000000 --value_size=200 --benchmarks=fillrandom
```



INSERT INTO B+TREE LEAF NODE



- Insert key-value pair to the end.
- Merge current idx array with index of new element to the second idx array.
- Switch current pointer to merged idx array.

```
class leaf_node_t {
private:
   value_type entries[slots];
   leaf_entries_t v[2];
   uint32_t current;
};
```

Optimization Notice

VTUNE UARCH CHARACTERIZATION

Clockticks:	86,458,898,535	
Instructions Retired:	35,338,687,890	
CPI Rate:	2.447	
MUX Reliability:	0.899	
Retiring:	11.3%	of Pipeline Slots
Front-End Bound:	12.2%	of Pipeline Slots
Bad Speculation:	3.1%	of Pipeline Slots
Back-End Bound:	73.4%	of Pipeline Slots
Memory Bound:	69.8%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
L2 Bound:	0.6%	of Clockticks
L3 Bound:	18.8%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	5.0%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Persistent Memory Bound:	49.1%	of Clockticks
Store Bound:	28.2%	of Clockticks
Store Latency:	24.8%	of Clockticks
False Sharing:	0.0%	of Clockticks
Split Stores:	0.0%	of Clockticks
DTLB Store Overhead:	0.3%	of Clockticks
Core Bound:	3.6%	of Pipeline Slots

Our workload is memory bound

All data stored in persistent memory
We are persistent memory bound

Optimization Notice

WHAT IS WRONG WITH OUR DATA STRUCTURES?

VTune Memory Access analysis can aggregate performance data per memory objects

leaf_node_t Memory objects

Current field: Offset inside **leaf_node_t**

Grouping: Memory Object / Offset / Function / Allocation Stack

Memory Object / Offset / Function / Allocation Stack	Hardware Event Count by Hardware Event Type	
	MEM_LOAD_RETIRED.L3_MISS...	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_4
db_bench.cc:629 (16 KB)	10,257,175	1,802,391
16656	1,090,763	87,261
persistent::internal::leaf_node_t<pstring<(unsigned l	1,050,7[illegible]5	7,021
persistent::inte[illegible]al::leaf_node_t<pstring<(unsigned l	40,028	0
persistent::interna[illegible]f_node_t<pstring<(unsigned long)20>, pstring<(unsigned long)200[illegible]igned long)63>::consistent		2,006
16136	560,392	21,063
4000	240,168	1,003
3752	170,119	7,021
8464	160,112	2,006
5984	160,112	3,009
1768	140,098	3,009
6480	140,098	1,003

Functions that access the field

There are a lot of LLC misses

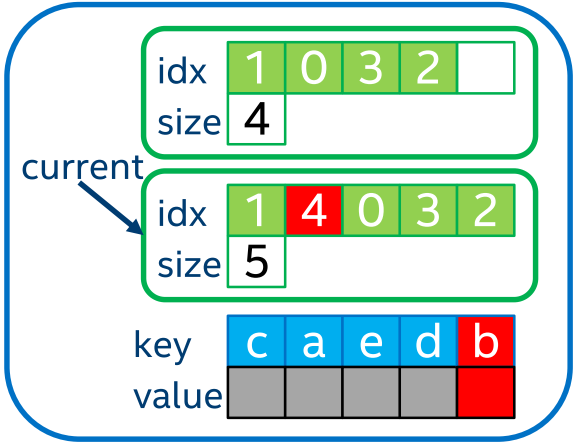
READ-AFTER-FLUSH PATTERN IN LEAF NODE

Problem:

- Reading **current** value causes a lot of LLC misses.
- Read-Modify-Flush pattern when updating **current** value.

Solution:

- Introduce shadow copy **p_current** to avoid flushes.
- Application reads and modifies **current** value. But never flush it.
- Changes of **current** value propagated to **p_current**.
- After restart **current** value restored from **p_current**.




```
void insert(…) {
  …
  update(current);
  p_current = current;
  flush(p_current);
  …
}
```

Optimization Notice

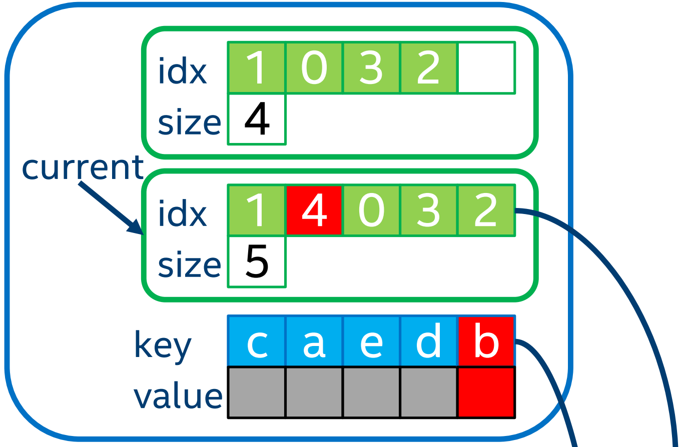
FALSE SHARING IN LEAF NODE

Problem:

- Reading **current** value still cause a lot of LLC misses.
- **current** shares the same cache line with **leaf_entries_t v[2].**

Solution:

- Change layout of the **leaf_node_t.**
- Keep **leaf_entries_t v[2]** and **current** in different cache lines.
- allignas(64) for **v[2]** array and **entries** array.




```
class leaf_node_t {
private:
  uint32_t p_current;
  value_type entries[slots];
  leaf_entries_t v[2];
  uint32_t current;
};
```

Optimization Notice

PERFORMANCE IMPROVEMENTS IN PERSISTENT B+TREE

Version	Ops/Sec	Descriptions
0	168066	Initial version
1	170586	Introduce shadow copy to avoid read-after flush pattern
2	178589	Change layout to avoid false sharing

Throughput of stree engine on fillrandom benchmark was increased by ~10K Ops/Sec.

```
class leaf_node_t {
private:
  uint32_t current;
  alignas(64) value_type entries[slots];
  leaf_entries_t v[2];
  alignas(64) uint32_t p_current;
};
```

Optimization Notice

CASE STUDY

PELIKAN ON PERSISTENT MEMORY

- **Pelikan** is Twitter's unified cache server.
- Can store data in persistent memory.
  - Data stored in memory mapped files managed by libpmem (PMDK).
  - Retains contents through graceful shutdown.
- **Cuckoo storage engine** is a cuckoo-based hash-array with a fixed size items.
  - Instrumented with VTune ITT API
  - Cuckoo items recognized by VTune as a memory objects.

Optimization Notice

MEMORY ANALYSIS FOR CUCKOO ENGINE (1/2)

Elapsed Time: 141.880s

CPU Time:	140.458s
Memory Bound:	78.5%
L1 Bound:	2.3%
L2 Bound:	0.7%
L3 Bound:	4.2%
DRAM Bound:	15.9%
NUMA: % of Remote Accesses:	0.0%
Persistent Memory Bound:	56.2%
Persistent Memory Bandwidth Bound:	0.0%
Local Persistent Memory:	100.0%
Remote Persistent Memory:	0.0%
Loads:	43,653,309,560
Stores:	18,466,553,980
LLC Miss Count:	581,040,670
Average Latency (cycles):	35

Workload is memory bound

Persistent memory bound

- Latency bound.
- Caching might improve the latency and increase performance

Optimization Notice

MEMORY ANALYSIS FOR CUCKOO ENGINE (2/2)

```
struct item {
 proc_time_i expire;//offset = 0
 uint8_t     klen;  //offset = 4
 uint8_t     vlen;  //offset = 5
 char        data[1];
};
```

Functions that access expire field

- Can we cache (expire, klen, vlen) in DRAM?
- Only 6 bytes per item.

Grouping: Memory Object / Offset / Function / Allocation Stack

Memory Object / Offset / ...	Loads	Stores	LLC Miss Count	Average Latency (cycles)
cuckoo.c:306 (214 B)	5,922,177,660	637,019,110	308,021,560	336
0	4,501,135,030	455,013,650	217,015,190	377
cuckoo_hit	3,612,108,360	0	182,012,740	446
cuckoo_displace	217,006,510	0	28,001,960	637
cuckoo_insert	266,007,980	0	7,000,490	749
__memmove_avx_ur	126,003,780	350,010,500	0	0
_sort_candidate	224,006,720	0	0	7
item_matched	0	0	0	0
_select_candidate	56,001,680	0	0	17
item_update	0	105,003,150	0	0
14	616,018,480	21,000,630	35,002,450	224
__memcmp_avx2_m	287,008,610	0	35,002,450	354
__memmove_avx_ur	0	21,000,630	0	0
hash_murmur3_32	329,009,870	0	0	7
4	49,001,470	0	14,000,980	350
item_matched	35,001,050	0	14,000,980	388
cuckoo_displace	7,000,210	0	0	7
item_datalen	7,000,210	0	0	0

SUMMARY

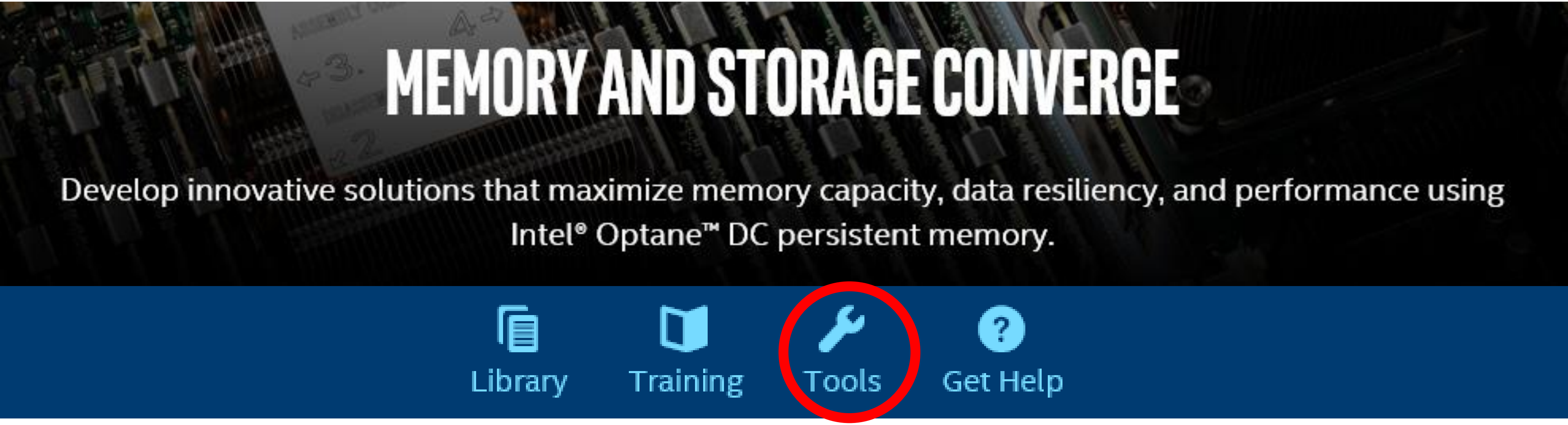
- Optimizing persistent memory bound applications is a challenging task.
- Hotspot might be blurred across multiple functions accessing the same memory.
- Intel VTune helps to identify inefficiency in data structures layout.
  - Memory Analysis allows to aggregate performance data per memory objects.

Contacts:

- If you need any help to enable or optimize your applications for persistent memory feel free to contact us:
  - Dmitry Ryabtsev Dmitry.Ryabtsev@intel.com
  - Sergey Vinogradov sergey.vinogradov@intel.com

SOFTWARE ANALYSIS TOOLS FOR INTEL® OPTANE™ DC PERSISTENT MEMORY DOWNLOADS & TECHNICAL ARTICLES

software.intel.com/persistent-memory



Click

THANK YOU

Questions

Optimization Notice