Leveraging Modern Hardware in SAP HANA
How SAP and Intel collaborate to Lead the Edge

Dr. Ismail Oukid (SAP), Dr. Thomas Willhalm (Intel)
January 22, 2018
Agenda

- Simple Instruction Multiple Data
- Hardware Transactional Memory
- Non-Volatile Memory
- Outlook
SIMD Simple Instruction Multiple Data
Simple Instruction Multiple Data (SIMD)

Scalar processing
- traditional mode
- one instruction produces one result

SIMD processing
- with Intel® SSE, AVX
- one instruction produces multiple results
Use-Case: Pack Integers in Bit-Fields

**Example:** Packed 17-bit fields

Integers in the range \([0, 100000]\) need only **17 bits**

**Idea:** Store only 17 bits (saving 15 bits per value)
Decompress Unaligned Bit Fields (Example: Packed 17-Bit Fields)

1. Load a 128-bit segment of input data into SSE register

2. Shuffle compressed values to target “32-bit segment”

3. Shift values to align them to 32-bit boundary

4. Store uncompressed values
Search Unaligned Bit Fields (Example: Packed 17-Bit Fields)

1. Load a 128-bit segment of input data into SIMD register

   ... 110300 65536 1772 2702 2 42

2. Shuffle compressed values to target “32-bit segment”

   _mm_shuffle_epi8

3. Parallel compare to shifted search range

   _mm_shuffle_epi8

4. Store result of search as bit-vector

   0100
Full-table Scan is 1.63x faster with SSE

Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB 2009*
Problem: No “vector-vector shift” in SSE

Hint: How do you implement a fast multiplication by “2”?  
Solution: Use multiplication for shifting

Multiply to shift left

\[
\text{__m128i mult_msk} = \text{__mm_set_epi32(0x04,0x02,0x01,0x80)};
\]
\[
\text{__m128i mult_rslt} = \text{__mm_mullo_epi32(shift_rslt, mult_msk)};
\]

Shift right

\[
\text{__mm_slli_epi32(mult_rslt1_m128i, 7)};
\]
## Unpacking of Bit-Fields with Intel® AVX2

<table>
<thead>
<tr>
<th>Pseudo Code</th>
<th>SSE 4.1</th>
<th>AVX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector load v from input array</td>
<td>movdqu x8(%r10,%rcx,1),%xmm6</td>
<td>vmovdqu xmm8, xmmword ptr[rax+rcx*1+0x11]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vinserti128 ymm9, ymm8, xmmword ptr [rax+rcx*1+0x19], 0x01</td>
</tr>
<tr>
<td>byte shuffle v</td>
<td>pshufb %xmm1,%xmm6</td>
<td>vpshufb ymm10, ymm9, ymm1</td>
</tr>
<tr>
<td>vector shift v</td>
<td>pmulld %xmm2,%xmm6</td>
<td>vpsrlvd ymm11, ymm10, ymm0</td>
</tr>
<tr>
<td></td>
<td>psrld $0xe,%xmm6</td>
<td></td>
</tr>
<tr>
<td>vector and v</td>
<td>pand %xmm0,%xmm6</td>
<td>vpand ymm12, ymm11, ymm2</td>
</tr>
<tr>
<td>vector store v in output array</td>
<td>movdqa %xmm6,0x10(%r8)</td>
<td>vmovdqu ymmword ptr [r8+0x20], ymm12</td>
</tr>
</tbody>
</table>

New variable shift instruction

Double the number of data elements vs. Intel® SSE

Implementation takes advantage of new variable shift
Intel® AVX2 Unpacking - Performance

Bit-field unpacking runs up to 1.6x faster on average with Intel® AVX2

Source: Willhalm et al. Vectorizing Database Column Scans with Complex Predicates. ADMS@VLDB 2014.
Intel® Advanced Vector Extensions 512

Expands register size to 512 bits

New mask registers:
- Results of comparisons
- Masking operations

New instructions:
- vpcompressd – extract selected DWORDs
- Scatter – distribute values
Using Mask Registers for Predication

if (v5<v6) {v1 += v3;}

v5 = 0 4 7 8 3 9 2 0 6 3 8 9 4 5 0 1
v6 = 9 4 8 2 0 9 4 5 5 3 4 6 9 1 3 0

vpcmpd k7, k0, zmm5, zmm6, 0x6

k7 = 1 0 1 0 0 0 1 1 0 0 0 0 1 0 1 0

v3 = 5 6 7 8 5 6 7 8 5 6 7 8 5 6 7 8
v1 = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

vaddpd v1, k7, v1, v3

v1 = 6 1 8 1 1 1 8 9 1 1 1 1 6 1 8 1

- Instructions operate only on selected elements
- Enables vectorization of "branchy" code, e.g. processing of NULL values
VCOMPRESS

Compress values that are marked in mask:

src  h g f e d c b a

mask

dest  e b a

VCOMPRESS generates list of values that are selected by bits in mask ("bit-vector")

Very useful to generate list of values in a column

- Filter-predicates
HTM Hardware Transactional Memory
Locks are blocking like traffic lights

Serialize execution only when necessary

Picture idea from Dave Boutcher
Intel® Transactional Synchronization Extensions

Transactionally execute lock-protected critical sections

Execute without acquiring lock
- Expose hidden concurrency

Hardware manages transactional updates – All or None
- Other threads can’t observe intermediate transactional updates
- If lock elision cannot succeed, restart execution & acquire lock
Intel® Transactional Synchronization Extensions

No Serialization and no communication if no data conflicts
Initial Analysis: B+Tree

Intel TSX provides significant gains with no application changes

- Outperforms RW lock on read-only queries
- Significant gains with increasing inserts (6x for 50%)

Source: HPCA 2014: Improving In-Memory Database Index Performance with Intel® Transactional Synchronization Extensions. Tomas Karnagel et al.
Up to 2x Performance Boost in OLTP When Running SAP HANA with TSX

Analysis: TSX Aborts in Delta Storage Index

Capacity Aborts

- Algorithmic level
  - Node/Leaf Search Scan
  - Causes random lookups
- Cache Associativity Limits
  - Aborts typically before cache size limits
  - Hyper-threads share the L1 cache
- Dictionary contributes to larger footprint

Data Conflicts

- Single dictionary
- Global memory allocator

Analysis leading to improvements in future HW generations
NVM  Non-Volatile Memory
Storage-Class Memory

SCM is byte-addressable Non-Volatile Memory

Access Latency in Cycles for a 4 GHz Processor [1]

SCM writes slower than reads

Limited write endurance

SCM is a merging point between memory and storage

Storage-Class Memory

Scale-up systems are constrained by the scalability limits of DRAM
→ SCM as potential remedy

Opportunities:

• Increased scalability
  o Larger memory modules means more memory available per server

• Significant cost savings
  o SCM is cheaper than DRAM

• Improved recovery times

Challenges:

• Higher (than DRAM) latency impacting performance

• New technology, standards still evolving…
  o Means slow, phased implementation with increased complexity and uncertain timelines
Sample Use Case: Reducing In-Memory Database Down-Time

Logging, warehousing, processing information: lifeline of companies

Information availability depends on database availability (9’s)

Minimize restart time to improve database availability

<table>
<thead>
<tr>
<th>Availability</th>
<th>Annual Downtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>97%</td>
<td>11 days</td>
</tr>
<tr>
<td>98%</td>
<td>7 days</td>
</tr>
<tr>
<td>99%</td>
<td>3 days 15 hrs</td>
</tr>
<tr>
<td>99.9%</td>
<td>8 hrs 48 min</td>
</tr>
<tr>
<td>99.99%</td>
<td>53 min</td>
</tr>
<tr>
<td>99.999%</td>
<td>5 min</td>
</tr>
<tr>
<td>99.9999%</td>
<td>32 sec</td>
</tr>
</tbody>
</table>

- Each restart for an IMDB can take up to 1 hour to load TBs of data to memory.
- Dell study shows millions of dollars lost per hour due to downtime**
- Existing HA solutions increase the price exponentially for every nine

### SAP HANA - Memory Architecture

<table>
<thead>
<tr>
<th>Function</th>
<th>Organization</th>
<th>Access type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch pad for intermediate results</td>
<td>Scratch pad space</td>
<td>Reads and writes</td>
</tr>
<tr>
<td>Portions of data that has been modified</td>
<td>~5% of data, less compressed</td>
<td>Reads and writes</td>
</tr>
<tr>
<td>Data in memory format</td>
<td>~95% of data, 10-20X compressed</td>
<td>Reads (scans)</td>
</tr>
</tbody>
</table>

Main store contains ~95% of the data in highly compressed format
Main Store is the perfect fit for SCM!

<table>
<thead>
<tr>
<th>Technology differentiators</th>
<th>Why Main is well suited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large capacity</td>
<td>Since 95% of data contained in main, HANA can scale up to larger datasets due to increased memory capacities</td>
</tr>
<tr>
<td>Persistence</td>
<td>Avoid loading data from storage and reduce downtime</td>
</tr>
<tr>
<td>Higher latencies</td>
<td>References are in form of scans. Hardware and software prefetchers can hide latencies for such reference patterns</td>
</tr>
</tbody>
</table>
Leveraging 3D XPoint™ PMEM technology for SAP HANA Main Store

- Primary data store is data volume (in SAN or local storage)

- Main is in 3D XPoint™ PM instead of DRAM and is now persistent

- On Restart: Main already in 3D XPoint™ PM, no need to load data from SAN

- On HW failure: Backup server loads data from data volume
Leveraging 3D XPoint™ PM for SAP HANA: Performance analysis

Promising initial results from a prototype (using HW emulation):

- Significant improvements in the restart time
  - >100x improvement measured

- Acceptable performance impact of higher (than DRAM) latencies, resulting in slightly lower performance
  - Measured (simulated) performance degradation was within the expected range for most workloads

- Source: Mihnea et al. SAP HANA Adoption of Non-Volatile Memory. VLDB 2017.
World’s first Intel Persistent Memory Demo at Sapphire 2017

- High capacity for scalability
- Data persistence without disk I/O
- Lower cost
- Less downtime
Next step: Mutable data structures on SCM
SCM Performance Implications

Sequential memory access patterns hardly affected

Random memory access patterns significantly affected

Need to keep DRAM next to SCM
SCM Programming Challenges

- Database developers are used to:
  - Ordering operations at the logical level (e.g., write undo log, then update primary data)
  - Fully controlling when data is made persistent (e.g., log durability must precede data durability)

- NVM invalidates these assumptions:
  - Little control over when data is made persistent
  - Writes need to be ordered at the system level
  - New failure scenarios

How to persist data in SCM?
Example: Array Append Operation

```c
void push_back(int val){
    m_array[m_size] = val;
    sfence();
    clwb(&m_array[m_size]);
    sfence();
    m_size++;
    sfence();
    clwb(&m_size);
    sfence();
}
```

What is in SCM?

<table>
<thead>
<tr>
<th>m_size</th>
<th>m_array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2017</td>
</tr>
</tbody>
</table>

Pros:
- Low-level optimizations possible

Cons:
- Programmer must reason about the application state
  → Harder to use and error prone

```c
void push_back(int val){
    TXBEGIN {
        m_array[m_size] = val;
        m_size++;
    } TXEND
}
```

à la software transactional memory

Pros:
- Easy to use and to reason about

Cons:
- Overhead due to systematic logging
- Low-level optimizations not possible
SCM-Based Data Structures: State-of-the-Art

Literature focuses mostly on tree-based data structures
→ Failure-atomic updates
→ Reduce SCM writes

Literature timeline

CDDS-Tree (FAST'11) → wB-Tree (VLDB'15) → NV-Tree (FAST'15) → FPTree (SIGMOD'16) → WORT (FAST'17) → HiKV (ATC'17)
FP Tree Design Goals

- Persistence
- Fast Recovery
- Near DRAM-Performance
- High scalability

<table>
<thead>
<tr>
<th>NV-Tree</th>
<th>NV-Tree, wBTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Selective Persistence</td>
<td>2. Unsorted Leaves</td>
</tr>
<tr>
<td>3. Fingerprinting</td>
<td>4. Selective Concurrency</td>
</tr>
</tbody>
</table>

FPTree
1. Selective Persistence

Inner nodes in DRAM for better performance

Leaves in SCM to ensure durability

Inner nodes rebuilt from leaves upon recovery in $O(#\text{entries})$

Recovery is up to 100x faster than a full rebuild
2. Unsorted Leaves

Counter

1. 3 4 7 14
   a b c

2. 3 4 7 14
   a b c

3. 3 4 5 7 14
   a d b c

4. 4 4 5 7 14
   a d b c

Sorted leaf

Potential corruption!

Unsorted leaf

Failure

Many writes!

- Failure-atomicity + fewer writes

But…linear search instead of binary search
3. Fingerprinting

A fingerprint is a 1-byte hash of a key

Fingerprints act as a filter to limit the number of key probes
3. Fingerprinting

Expected number of probed keys is **one** for leaf sizes up to 64
Hardware Transactional Memory (HTM)

HTM and SCM are apparently incompatible

Transactions keep read and write sets in L1 cache

FLUSH → Abort!
4. Selective Concurrency

Transient – Hardware Transactional Memory

Persistent – Fine-grained locking
4. Selective Concurrency: Insertion

Selective Concurrency solves the incompatibility of HTM and SCM

Transient

Persistent

1. Find and lock leaf
2. Modify leaf
3. Update parents
4. Unlock leaf

XBEGIN
FLUSH
XEND

© 2018 SAP SE or an SAP affiliate company. All rights reserved. | PUBLIC
The FPTree is competitive with a Transient B+-Tree

Only ~3% of data in DRAM
FPTree Performance Evaluation: Multi-Threaded

The FPTree scales nearly linearly
Outlook
Extended Hardware ingredients for Optimizations

Intel QuickAssist Technology:
Hardware acceleration for high-compute workloads (crypto, compression)

NIC HW acceleration:
Intelligent or Programmable NICs

FPGA acceleration:
Processor companion chip or as a PCIe add-on board to accelerate platform or workload functions
Summary

➢ SIMD - Novel use-cases for databases
➢ TSX – fine-grained locking with coarse-grained implementation
➢ Storage-Class Memory – Merging storage and memory
➢ More exciting topics are ahead

Innovative approaches are possible when hardware and software are changed at the same time
Thank you.

Contact information:

Ismail Oukid
Ismail.Oukid@sap.com

We are hiring!

- Full-time positions
- PhD positions
- Student jobs (master/bachelor theses, internships, working student)

Contact: students-hana@sap.com
“The price of reliability is the pursuit of utmost simplicity.”

– C.A.R Hoare