High Core Count Performance Optimization in ClickHouse

Jiebin Sun (jiebin.sun@intel.com)

Linux Performance & Innovation/STG

Yaqi Zhao(yaqi.zhao@intel.com)

XCS SW ENABLING INFRASTUCTURE

Agenda

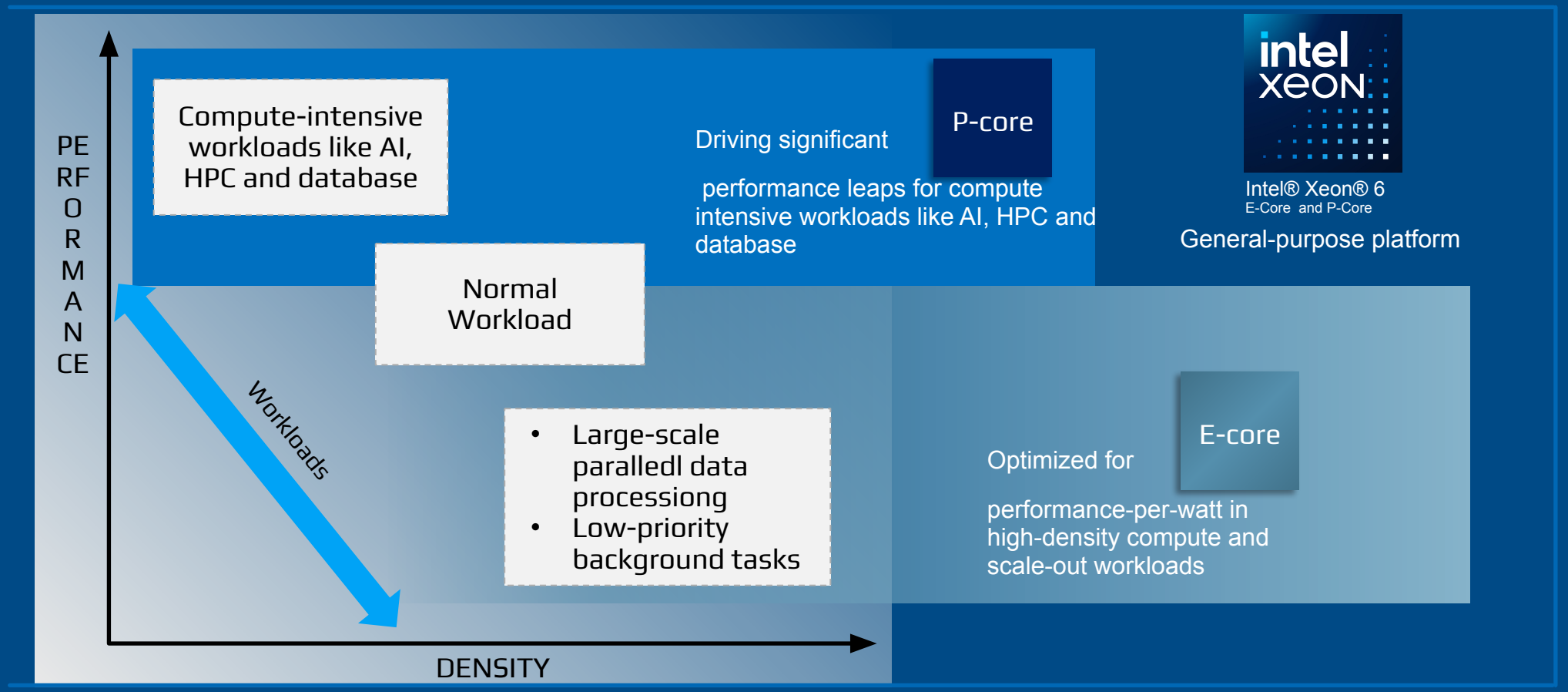
- Introduction
- Brief Introduce to Intel® Xeon® processors
- HCC Optimization Methodology
  - Reduce lock contentions in critical path.
  - Reduce memory bandwidth/path length.
  - Increase the thread level parallelism.
  - Search algorithm.
- Q&A

Introduction

- Focus on high core count (HCC) performance optimization in Linux kernel & ClickHouse & base lib.

Merged performance PR Link	Benchmark	Test Platform	Performance Improvement
Optimize the mutex with shared_mutex in the memory tracker (#72375)	Clickbench	2 x 256 vCPUs	Q8: 77%, Q24: 19.5%, Q26: 19.5%, overall geomean: 6.8%
Change the default threshold to enable hyper threading (#70111)	Clickbench	2 x 80 vCPUs	overall geomean: 13.2% @32 vCPUs, 7.6% @48 vCPUs
Add thread pool and cancellation to support parallel merge with key (#68441)	Clickbench	2 x 80 vCPUs	Q8: 10.3%, Q9: 7.6%
Rewrite the AST/Analyzer of sum(column +/- literal) function (#57853)	Clickbench	1 x 144 vCPUs	Q29: 5.9x @144 vCPUs, 21.8x @16 vCPUs
Limit the array index of FixedHashTable by min/max (#62746)	Clickbench	2 x 80 vCPUs	Q7: 2.1x
Redesign the iterator in ColumnSparse::filter (#64426)	Clickbench	2 x 80 vCPUs	Q10: 9.6%, total cycles reduced to 79.2%
Rewrite the ColumnSparse::Iterator (#64497)	Clickbench	2 x 80 vCPUs	Total cycle reduced to 97% and total instructions reduced to 88.3%
Release more num_streams if data is small (#53867)	Clickbench	2 x 80 vCPUs	Q39: 3.3x, Q36: 2.6x
Convert hashsets in parallel before merge (#50748)	Clickbench	2 x 112 vCPUs	Q5: 2.64x
Optimize the merge of singlelevelhash (#52973)	SELECT COUNT(DISTINCT Title) FROM hits_v1	2 x 80 vCPUs	2.35x
Maintain per-thread timer_id rather than create/delete frequently (#48778)	Clickbench/SSB	2 x 112 vCPUs	Clickbench Q4: 17.5%, Q5: 8.3%. SSB: 18% overall geomean (#49965)
Optimize the SIMD StringSearcher by searching first two chars (#46289)	Clickbench	2 x 80 vCPUs	Q20: 35%, overall geomean: 4.1%

Infrastructure Refresh – Intel® Xeon® 6



Reimagine and Optimize Your Datacenter

Intel® Xeon® 6 with Performance cores and Efficient cores

Up to 128 performance cores (P-cores) , The best CPU for general purpose AI & host CPU deployments in AI accelerated systems

Up to 144 efficient cores (E-cores) specializing in data services, networking, media and microservices workloads

Customers might be afraid to refresh and modernize their data infrastructure because of the cost required and overall time commitment.

By refreshing infrastructure with Intel® Xeon® 6900 with P-cores and Intel® Xeon® 6700 with E-cores, customers can consolidate server count and experience gains like:

- *Lower TCO*
- *Improved workload performance G2G and vs. AMD EPYC*
- *Greater efficiency and perf/watt*

Customer Challenges/Pain Points

- Rising compute costs
- Inability to justify investment and quickly achieve ROI
- Performance inefficiencies
- Pressure to operate more efficiently with better TCO/sustainability grades

Higher Performance & Efficiency

Refresh Servers from 2nd Gen Intel Xeon CPUs to Intel Eon 6 CPUs

up to **4.2x** higher performance[1]

up to **2.6x** higher perf/watt[1]

ClickHouse Hardware Benchmark

Ali Cloud ecs.c9i

up to **1.12** Higher performance vs AWS c8g.48xlarge

up to **1.3x** Higher perf vs AMD

阿里云

Machine	Relative time (lower is better)
AWS c8g.48xlarge (Graviton 4)†:	×1.25
AMD EPYC 9454P NVMe†:	×1.41
AWS c7g.metal, 500 GB EBS:	×1.45
AWS c7a.metal-48xl, 500 GB EBS:	×1.45
AWS c7i.metal-48xl, 500 GB EBS:	×1.54
AMD EPYC 9454P†:	×1.54

Learn more about SKU transitions, software readiness and refresh opportunities:https://xeonprocessoradvisor.intel.com

HCC Optimization Methodology

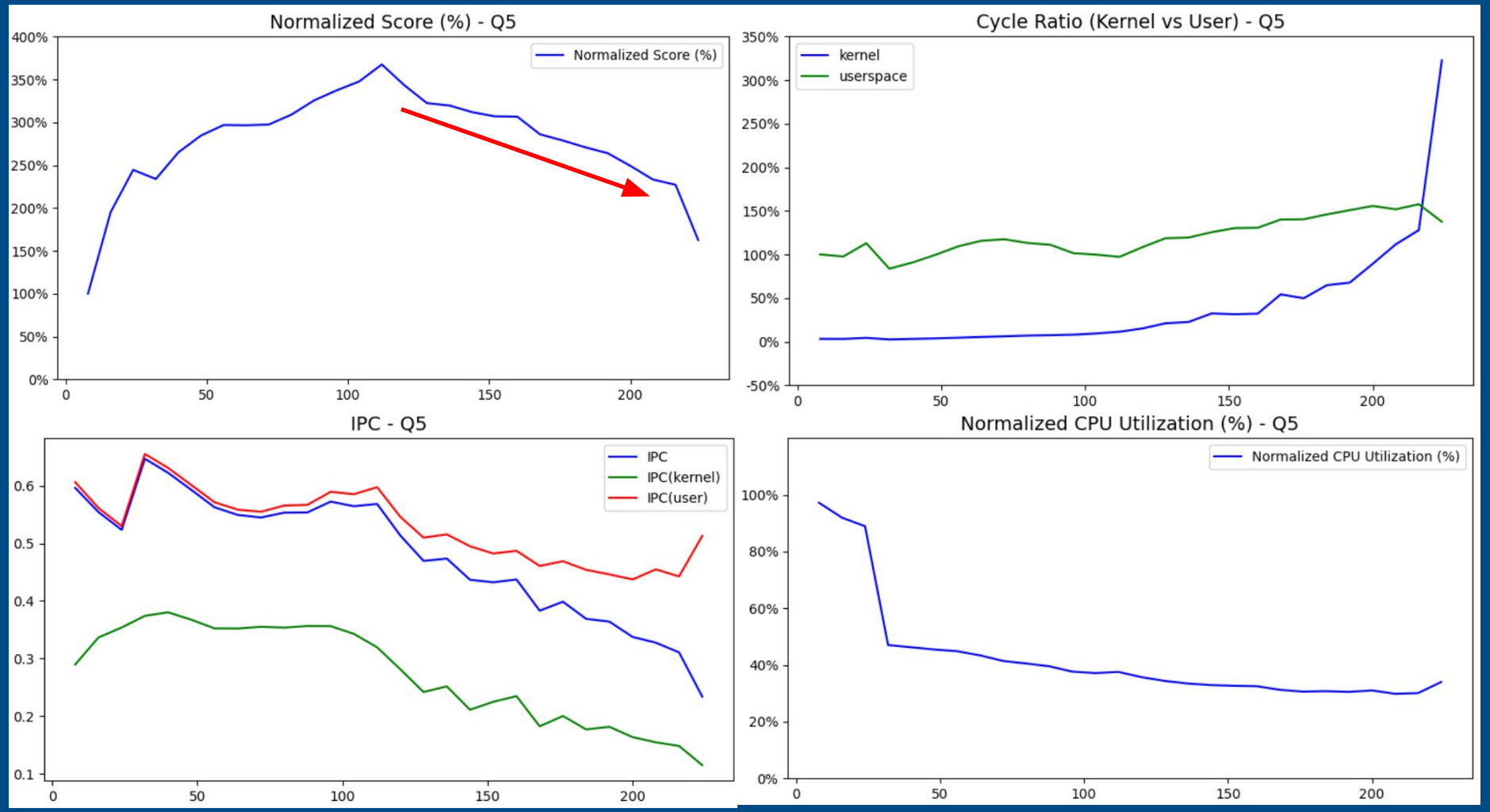
- Reduce lock contentions in critical path.
- Reduce memory bandwidth/path length.
- Increase the thread level parallelism.
- Search algorithm.

Reduce lock contentions in critical path

- Examples
  - Maintain per-thread timer_id rather than create/delete frequently (#48778). Overall geomean of SSB: 18% @ 2 x 80 vCPUs.
  - Optimize the mutex with shared_mutex in the memory tracker (#72375). Q8: 77%, Q24: 19.5%, Q26: 19.5%, overall geomean: 6.8% @ 2 x 256 vCPUs.

Timer -- The Problem

- Q5 of Clickbench is not scalable & kernel dominant. (SELECT COUNT(DISTINCT SearchPhrase) FROM hits)



Timer -- Hotspot Analysis

- Perf cycle hotspots in 224 threads:

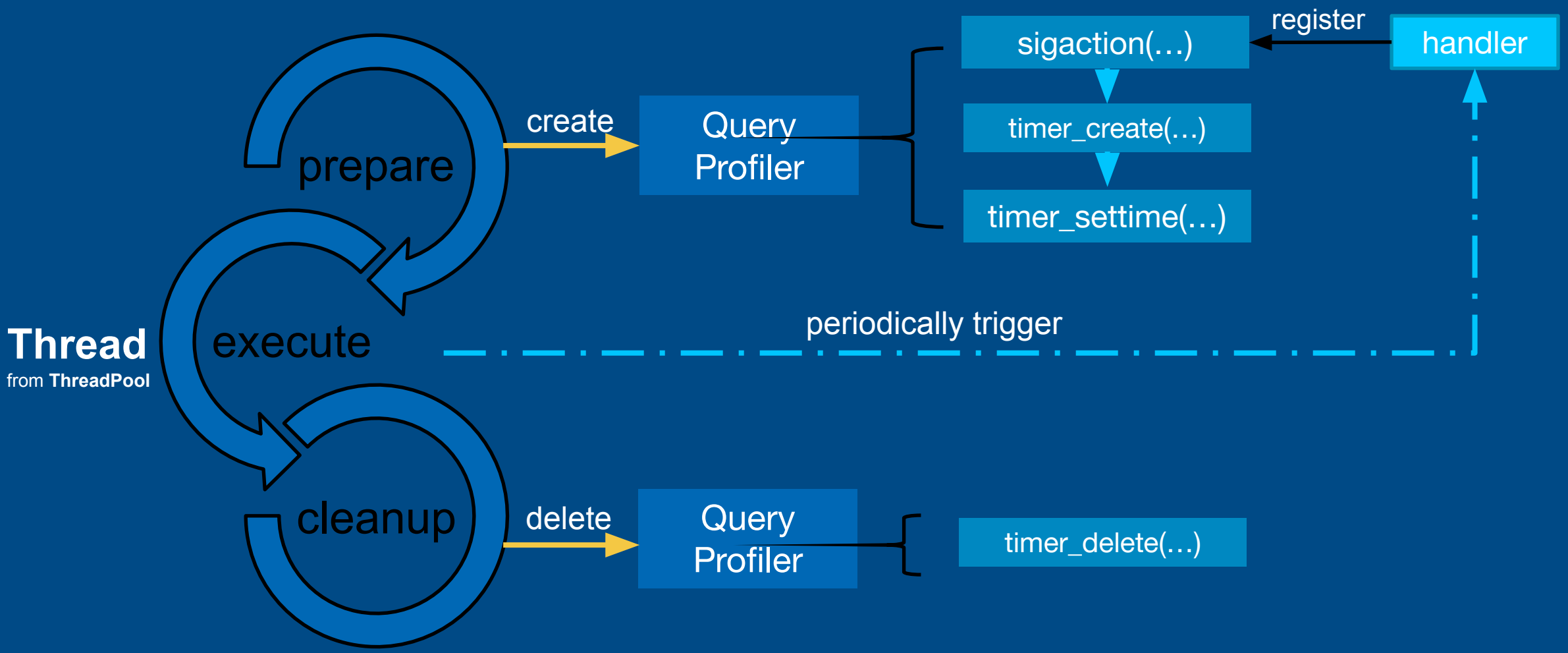
```
  Overhead  Shared Object       Symbol
-   42.57%  [kernel.kallsyms]   [k] native_queued_spin_lock_slowpath
   - 23.45% 0
      - 23.09% timer_delete@@GLIBC_2.3.3
         + 23.09% entry_SYSCALL_64_after_hwframe
   + 3.61% __libc_sigaction
   + 2.43% timer_settime@@GLIBC_2.3.3
   - 2.25% 0x40000000c
      + 2.25% timer_create@@GLIBC_2.3.3
   - 1.83% 0x40000000a
      + timer_create@@GLIBC_2.3.3
   - 0.75% __lll_unlock_wake
      + 0.75% entry_SYSCALL_64_after_hwframe
+   11.18%  clickhouse          [.] HashSetTable<long, HashTableCell<
+    9.83%  clickhouse          [.] DB::AggregateFunctionUniq<long, D
+    5.81%  clickhouse          [.] HashTable<long, HashTableCell<lon
+    5.23%  libc-2.28.so        [.] __memset_avx2_erms
+    3.55%  clickhouse          [.] memcpy
+    3.36%  clickhouse          [.] TwoLevelHashTable<long, HashTable
     0.70%  [kernel.kallsyms]   [k] copy_user_enhanced_fast_string
     0.63%  [kernel.kallsyms]   [k] down_read_trylock
+    0.57%  clickhouse          [.] HashTable<long, HashTableCell<lon
```

More detail callstack with LBR

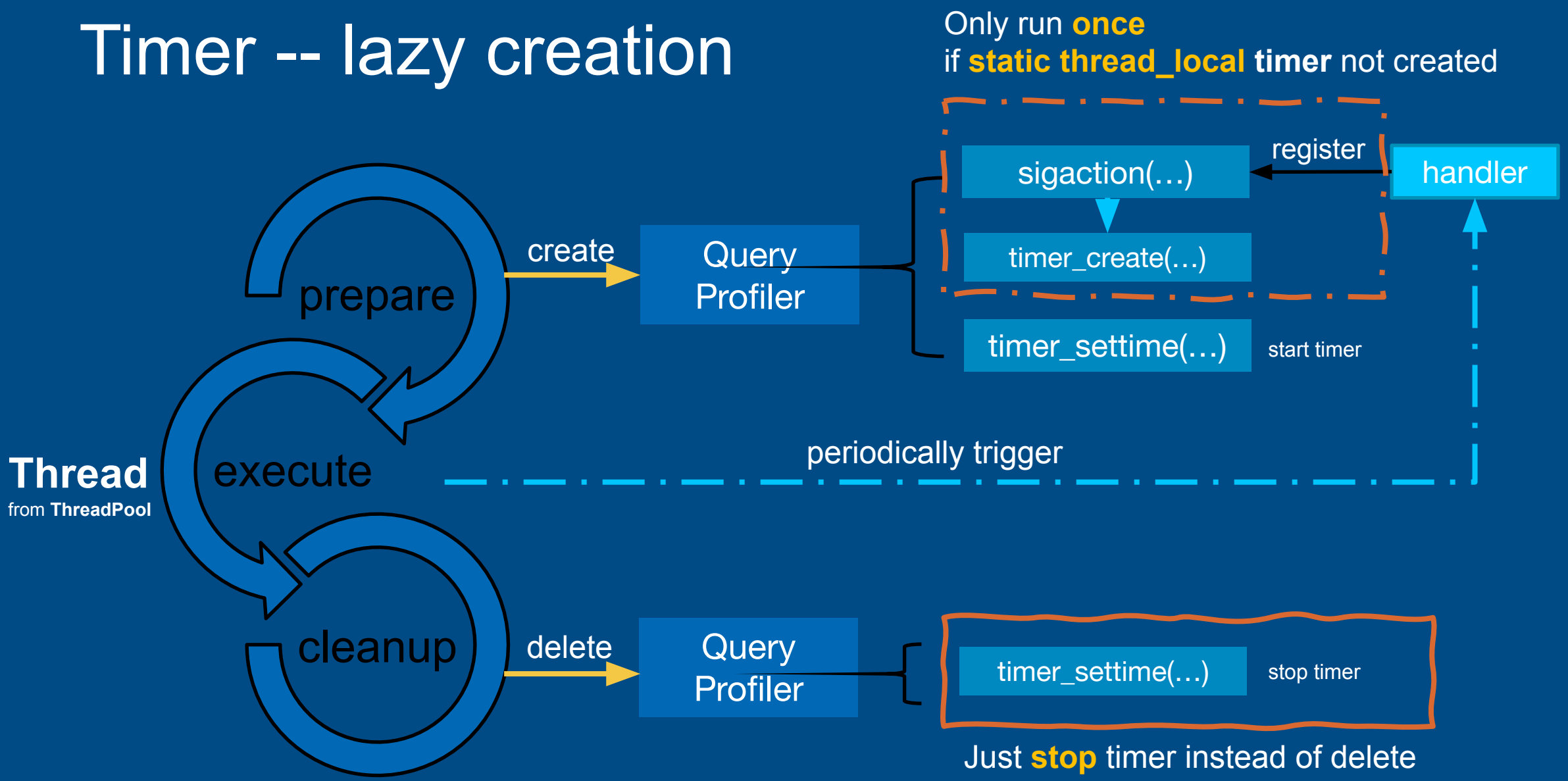
```
  Overhead  Shared Object       Symbol
-   42.87%  [kernel.kallsyms]   [k] native_queued_spin_lock_slowpath
   - 33.90% ThreadPoolImpl<ThreadFromGlobalPoolImpl<false> >::worker
      - 33.21% std::__1::__invoke_void_return_wrapper<void, true>::__call<DB::UniqExactS
         - 22.18% DB::UniqExactSet<HashSetTable<long, HashTableCell<long, HashCRC32<long
            - 22.17% DB::ThreadStatus::detachFromGroup
               - 11.23% DB::QueryProfilerBase<DB::QueryProfilerCPU>::~QueryProfilerBase
                  + DB::QueryProfilerBase<DB::QueryProfilerCPU>::tryCleanup
               - 7.19% DB::QueryProfilerBase<DB::QueryProfilerReal>::~QueryProfilerBase
                  + DB::QueryProfilerBase<DB::QueryProfilerReal>::tryCleanup
               + 3.60% DB::ThreadStatus::finalizePerformanceCounters
         + 10.76% DB::ThreadStatus::attachToGroupImpl
   + 4.04% std::__1::__invoke_void_return_wrapper<void, true>::__call<DB::UniqExactSet<H
   + 1.92% DB::ThreadStatus::attachToGroupImpl
   + 1.73% DB::ThreadStatus::detachFromGroup
```

QueryProfiler is the problem!

Timer -- QueryProfiler in ClickHouse

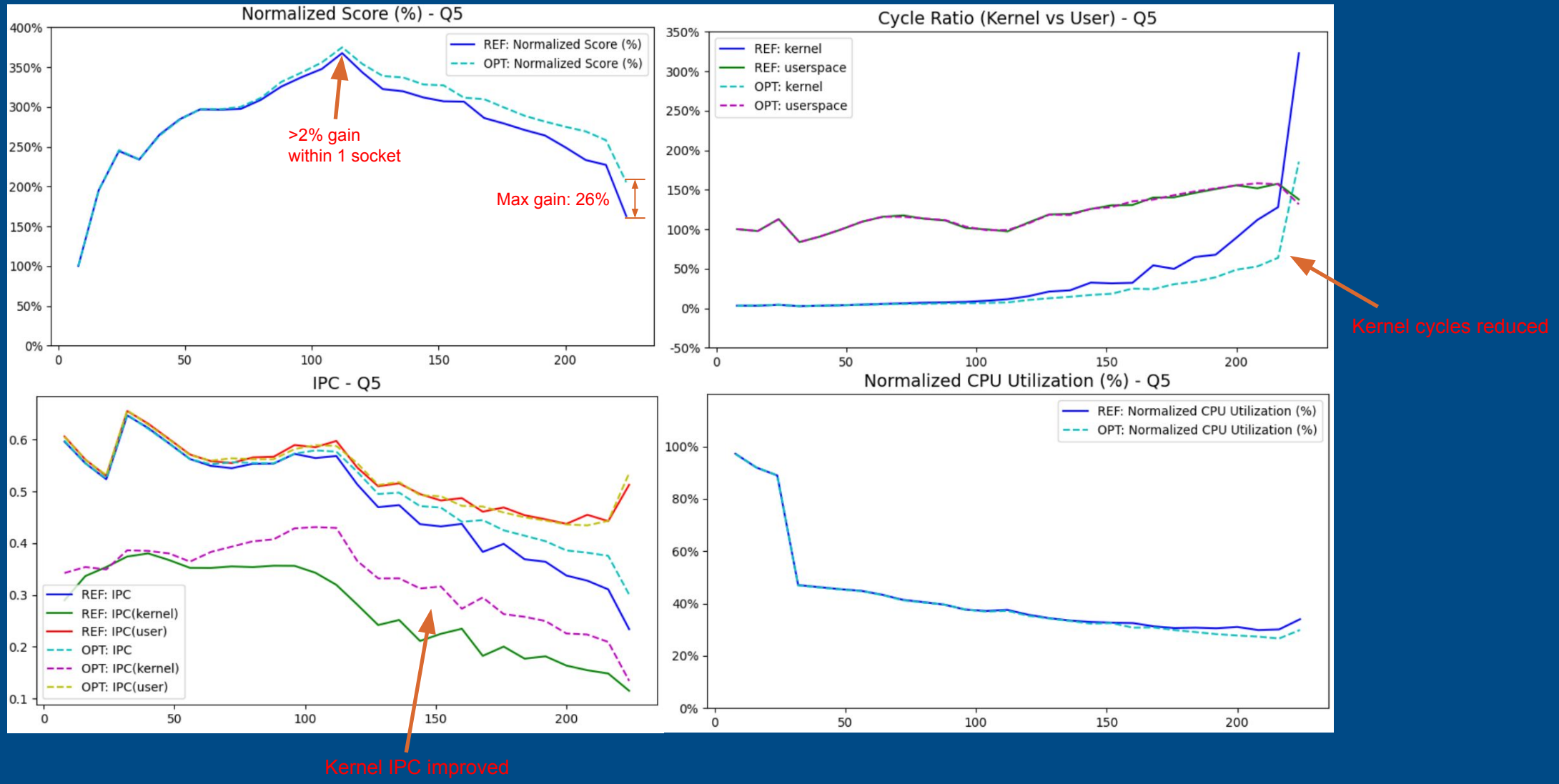


Timer -- lazy creation



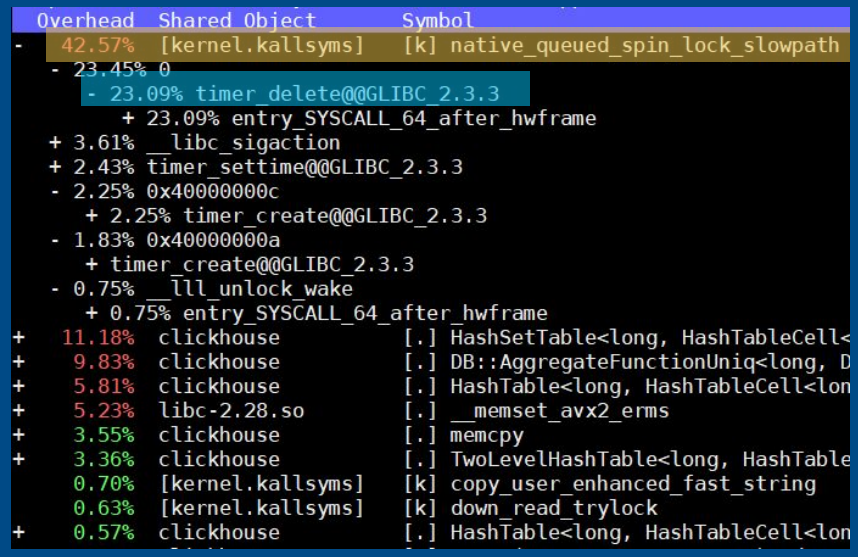
https://github.com/ClickHouse/ClickHouse/pull/48778

Timer -- The results

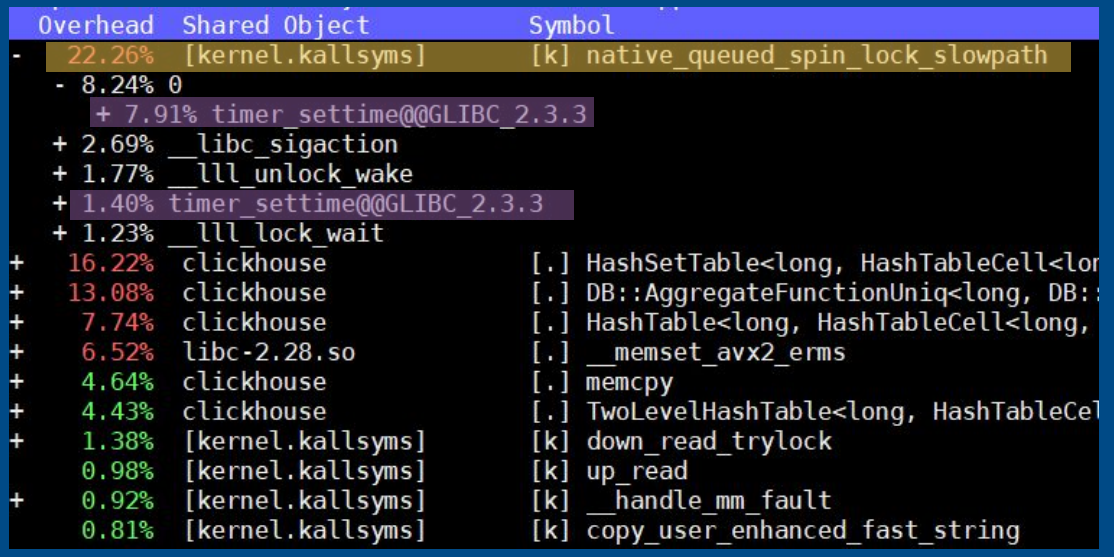


Timer -- Hotspot changes

- 2 x 112 vCPUs



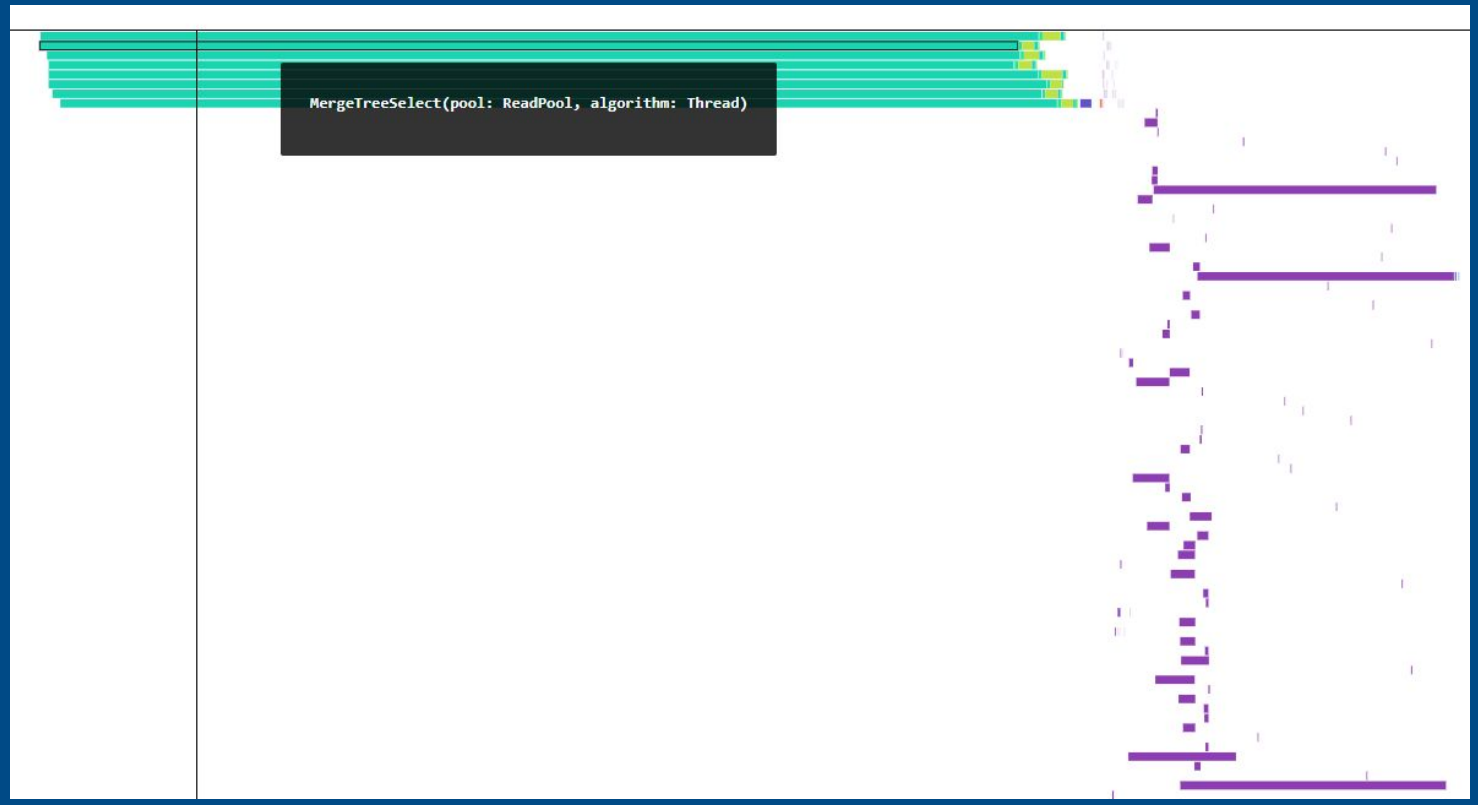
before



after

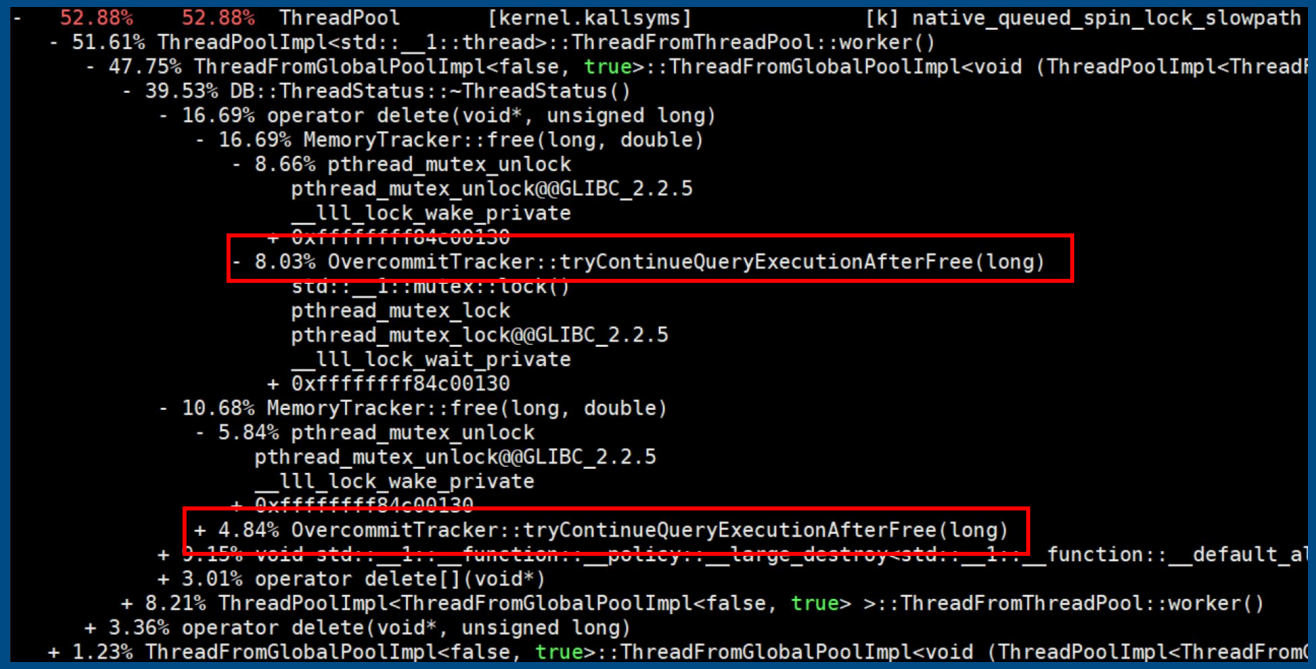
MemoryTracker -- Pipeline

- Q42 (SELECT DATE_TRUNC('minute', EventTime) AS M, COUNT(*) AS PageViews FROM hits WHERE CounterID = 62 AND EventDate >= '2013-07-14' AND EventDate <= '2013-07-15' AND IsRefresh = 0 AND DontCountHits = 0 GROUP BY DATE_TRUNC('minute', EventTime) ORDER BY DATE_TRUNC('minute', EventTime) LIMIT 10 OFFSET 1000)



MemoryTracker -- Hotspot

- 2 x 240 vCPUs



MemoryTracker – The Mutex

- The mutex will protect the cancellation_state read and the freed_memory update

Mutex

Protected

```
void OvercommitTracker::tryContinueQueryExecutionAfterFree(Int64 amount)
{
    DENY_ALLOCATIONS_IN_SCOPE;

    if (OvercommitTrackerBlockerInThread::isBlocked())
        return;

    std::lock_guard guard(overcommit_m);
    if (cancellation_state != QueryCancellationState::NONE)
    {
        freed_memory += amount;
        if (freed_memory >= required_memory)
            releaseThreads();
    }
}
```

MemoryTracker -- Optimization

- Due to the test, the freed_memory update part is much less than the read part.
- Mutex -> Shared_Mutex

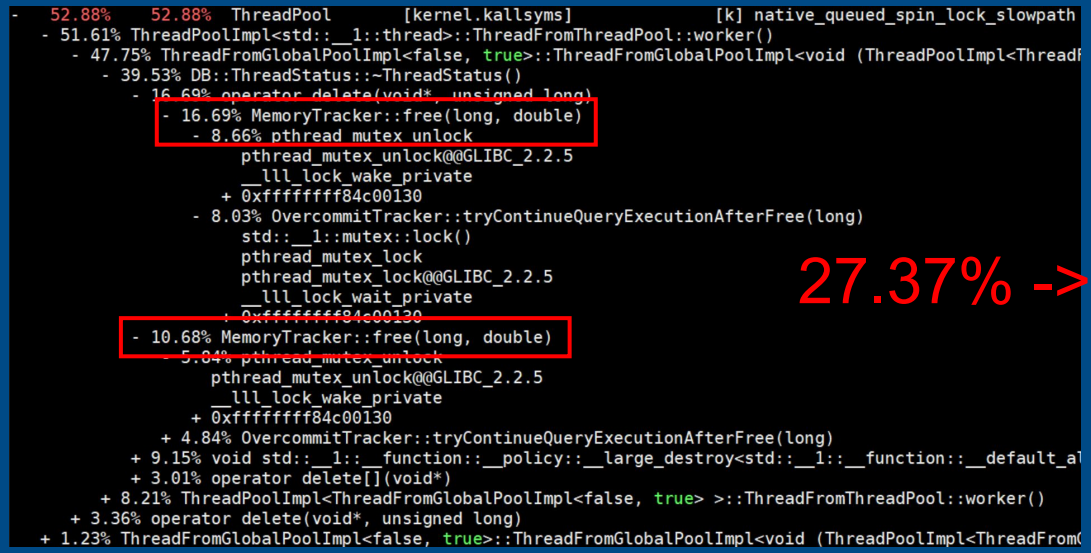
Read and check

write

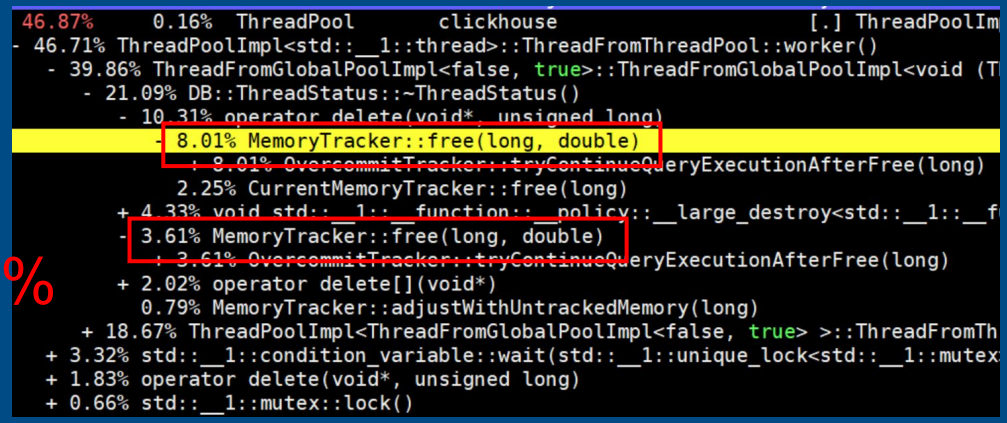
```
void OvercommitTracker::tryContinueQueryExecutionAfterFree(Int64 amount)
{
    DENY_ALLOCATIONS_IN_SCOPE;

    if (OvercommitTrackerBlockerInThread::isBlocked())
        return;

    std::lock_guard guard(overcommit_m);
    {
        std::shared_lock read_lock(overcommit_m);
        if (cancellation_state == QueryCancellationState::NONE)
            return;
    }

    std::lock_guard lk(overcommit_m);
    if (cancellation_state != QueryCancellationState::NONE)
    {
        freed_memory += amount;
        if (freed_memory >= required_memory)
            releaseThreads();
    }
}

void OvercommitTracker::onQueryStop(MemoryTracker * tracker)
{
    DENY_ALLOCATIONS_IN_SCOPE;

    {
        std::shared_lock read_lock(overcommit_m);
        if (picked_tracker != tracker)
            return;
    }

    std::lock_guard lk(overcommit_m);
    if (picked_tracker == tracker)
    {
        reset();
        cv.notify_all();
    }
}
```

MemoryTracker – The result

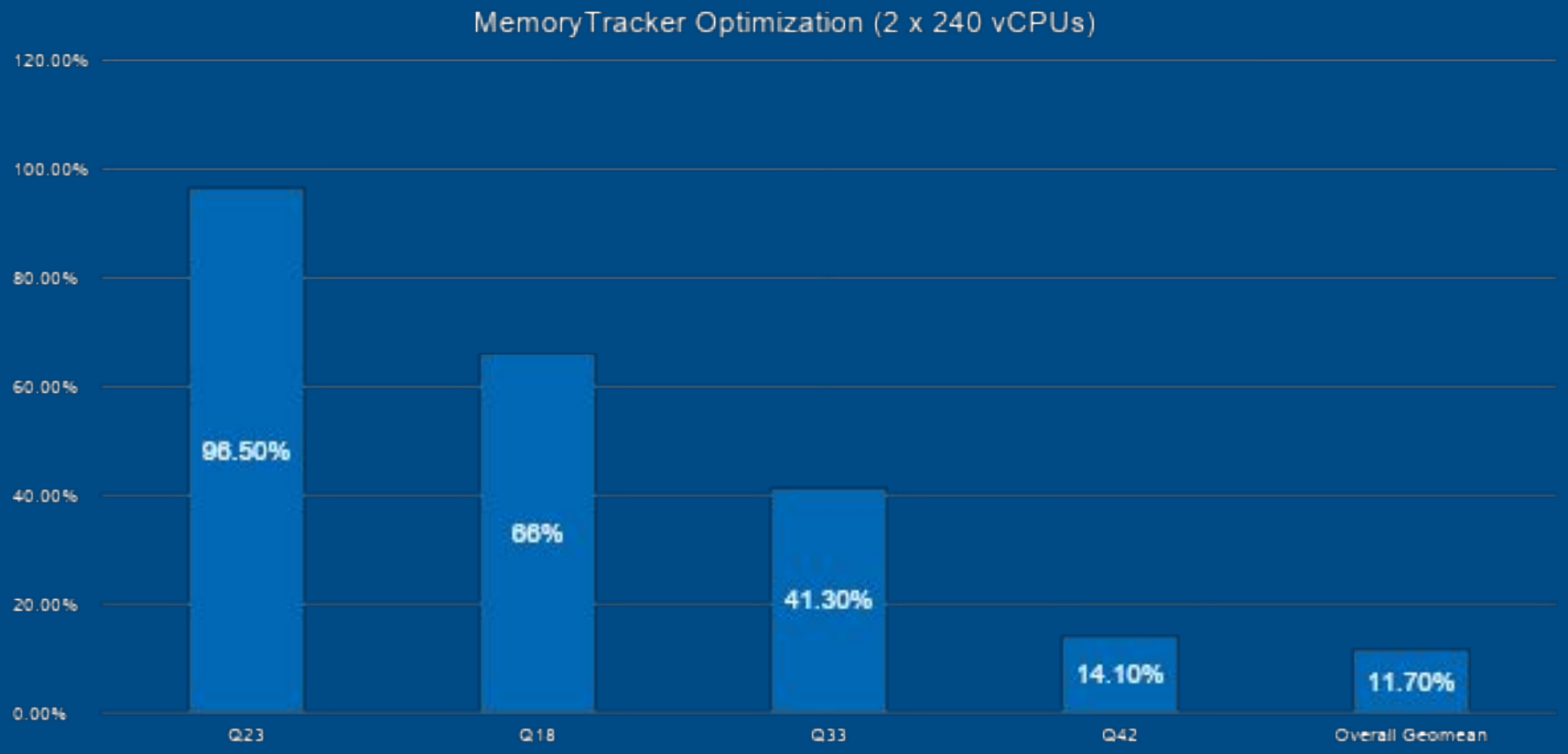
- Hotspot of memoryTracker::free before and after



27.37% -> 11.62%



MemoryTracker – The result

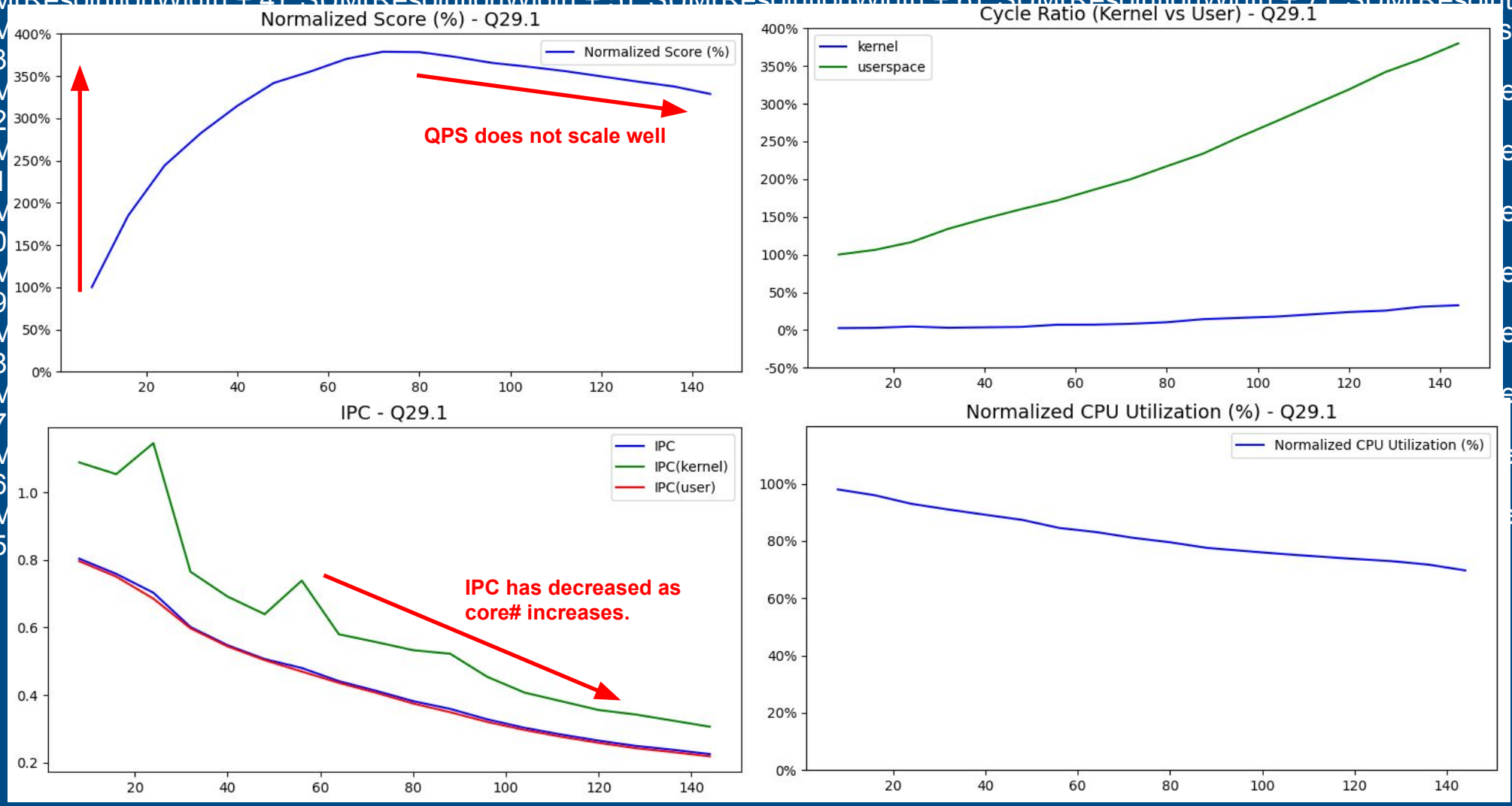


Reduce memory bandwidth/path length

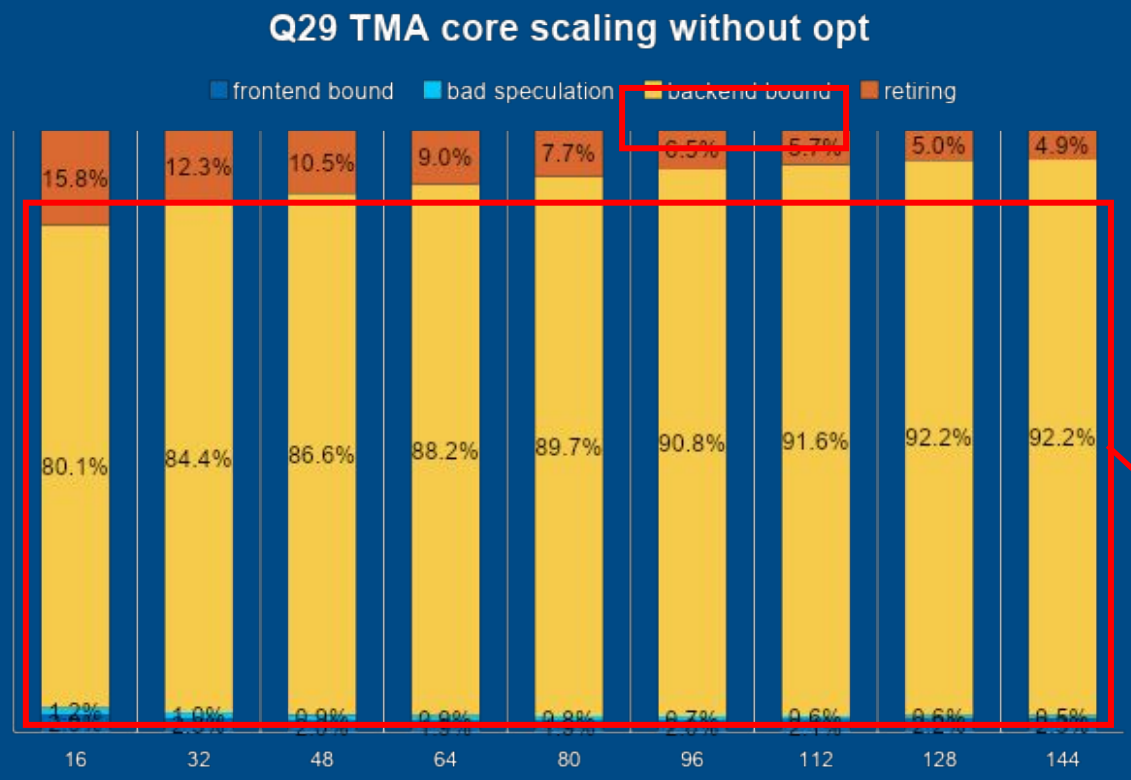
- Examples
  - Rewrite the frontend AST and Analyzer. Q29: 5.9x @144 vCPUs and 21.8x @16 vCPUs. The total memory bandwidth: 1.4% as before @144 vCPUs.
  - Limit the array index of FixedHashTable by min/max. Q7: 2.1x @ 2 x 80 vCPUs.
  - Redesign the iterator in ColumnSparse::filter. Q10: 9.6% @ 2 x 80 vCPUs.

Rewrite -- The Problem

Q29: SELECT SUM(ResolutionWidth), SUM(ResolutionWidth + 1), SUM(ResolutionWidth + 2), SUM(ResolutionWidth + 3), SUM(ResolutionWidth + 4), SUM(ResolutionWidth + 5), SUM(ResolutionWidth + 6), SUM(ResolutionWidth + 7), SUM(ResolutionWidth + 8), SUM
+ 13
SUM
+ 22
SUM
+ 31
SUM
+ 40
SUM
+ 49
SUM
+ 58
SUM
+ 67
SUM
+ 76
SUM
+ 85
solutionWidth
esolutionWidth
esolutionWidth
esolutionWidth
esolutionWidth
esolutionWidth
esolutionWidth
esolutionWidth
esolutionWidth



Rewrite -- The Problem: High Memory Bound



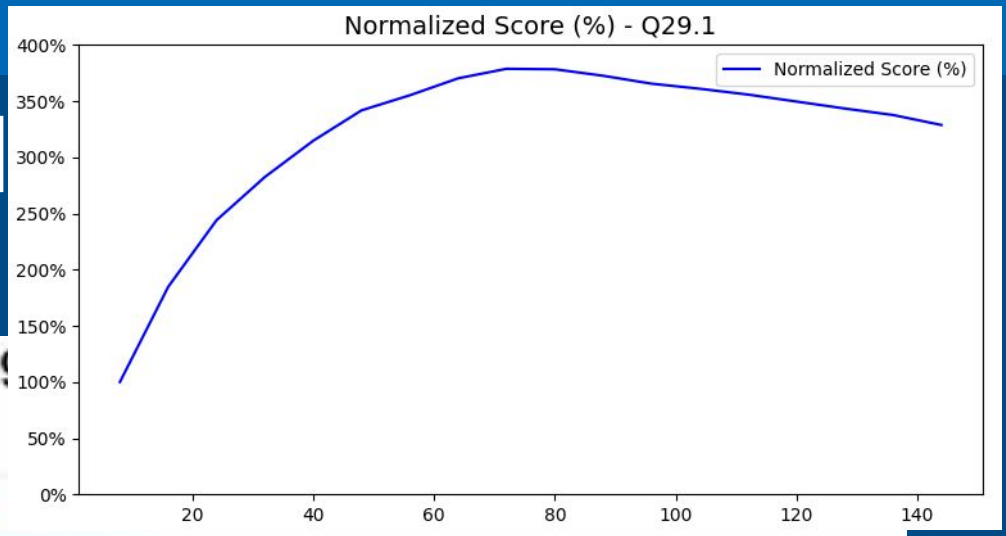
- Clickbench Q29:

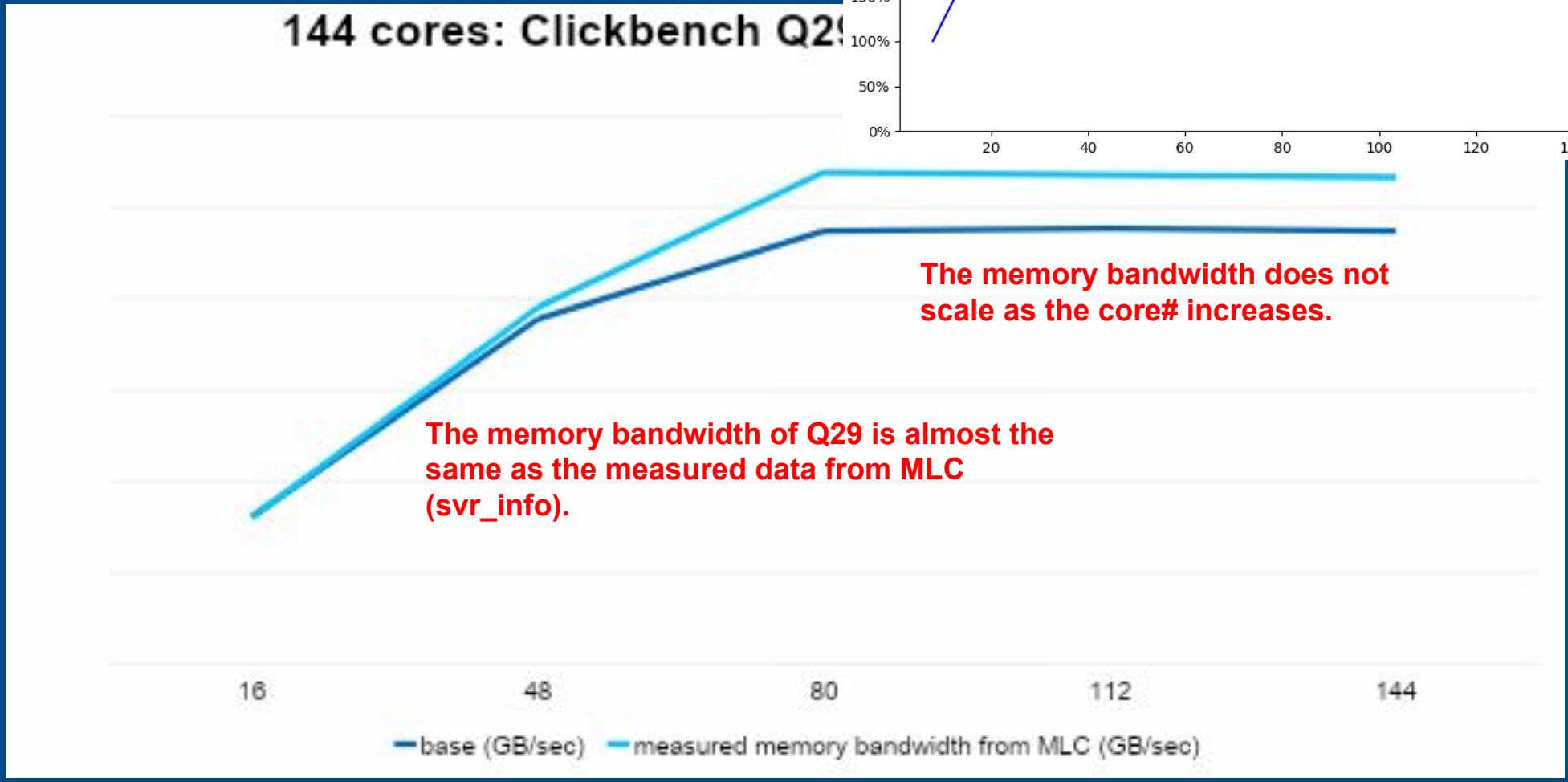
```
SELECT SUM(ResolutionWidth),
SUM(ResolutionWidth + 1),
SUM(ResolutionWidth + 2), …,
SUM(ResolutionWidth + 89) FROM hits;
```

- Cost lots of memory to load and get the similar function `sum(column + literal)`;
- Emon data:

Over 90% memory bound

Rewrite -- The Problem: H
Bandwidth

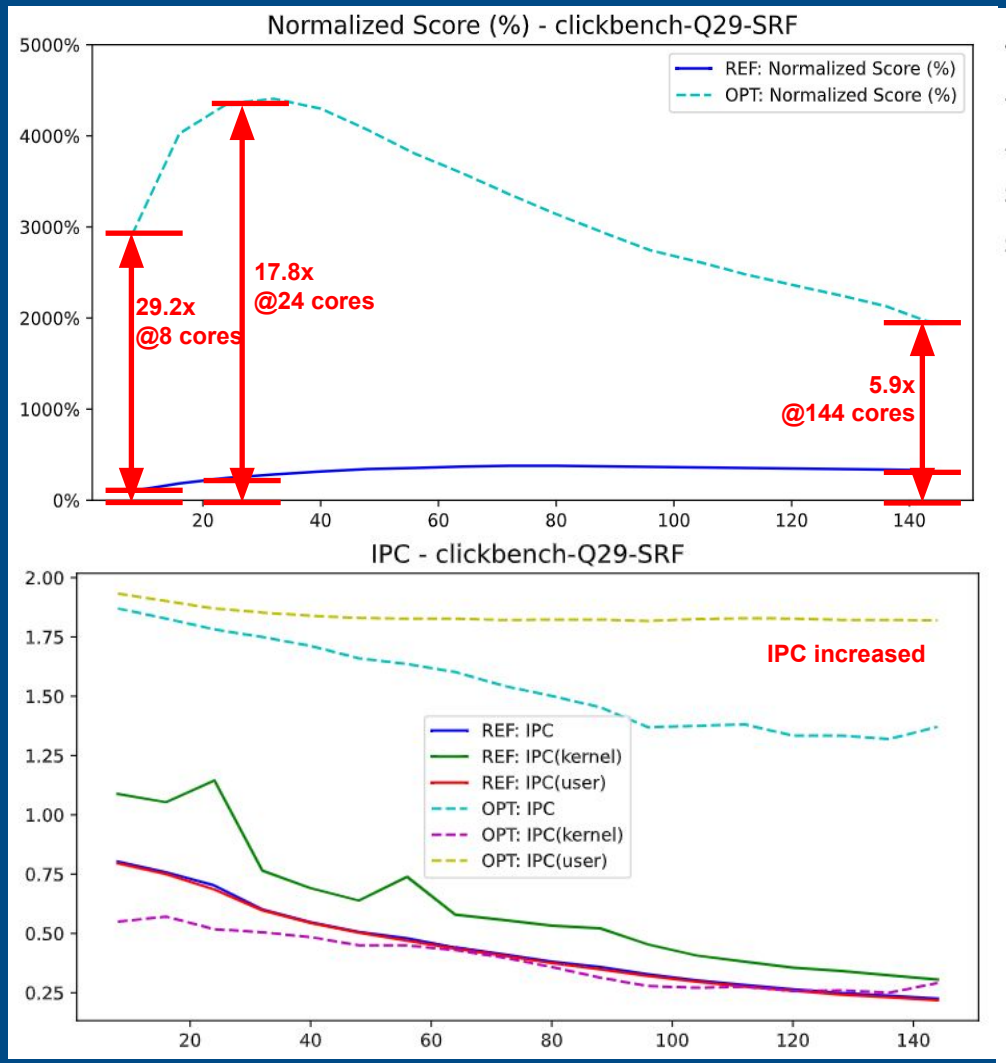


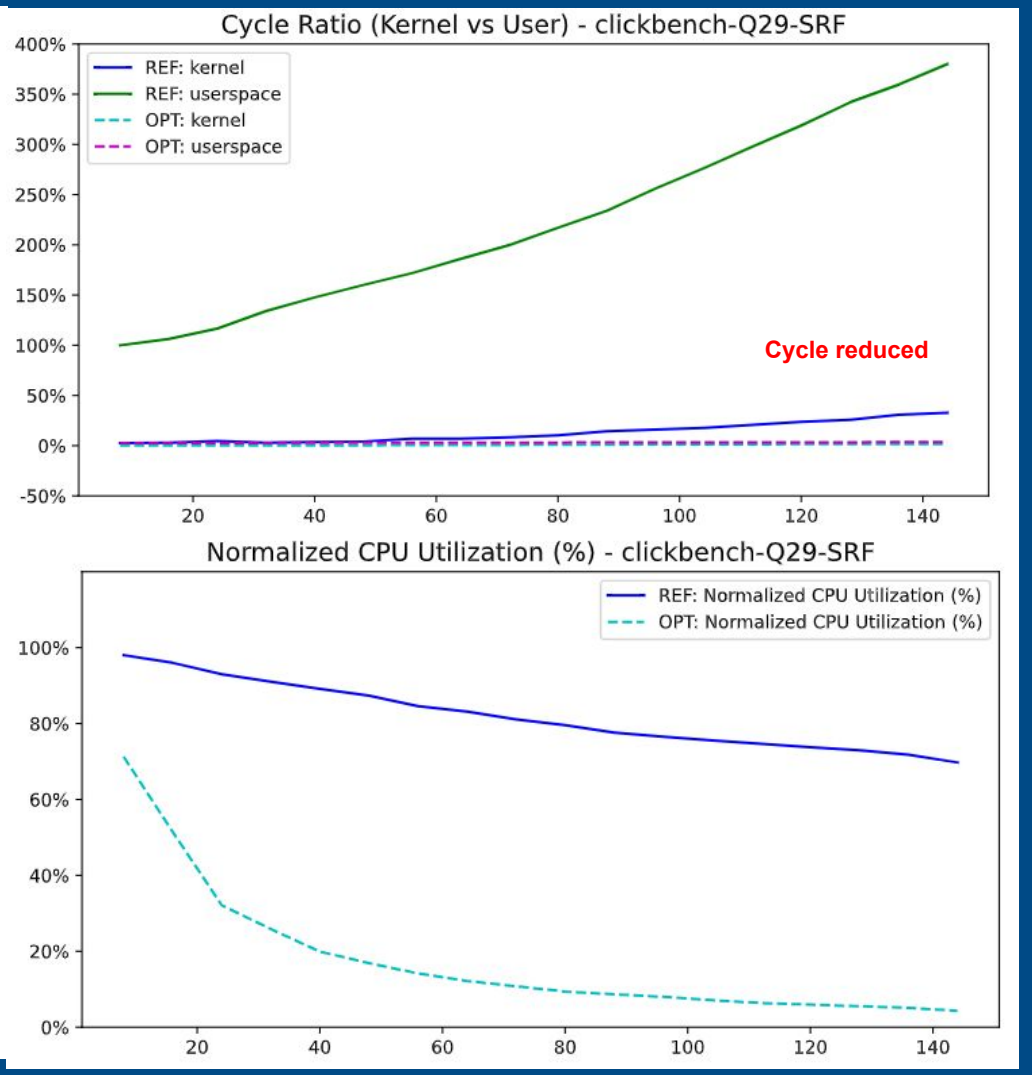


Rewrite -- Solution: Rewrite the sum function

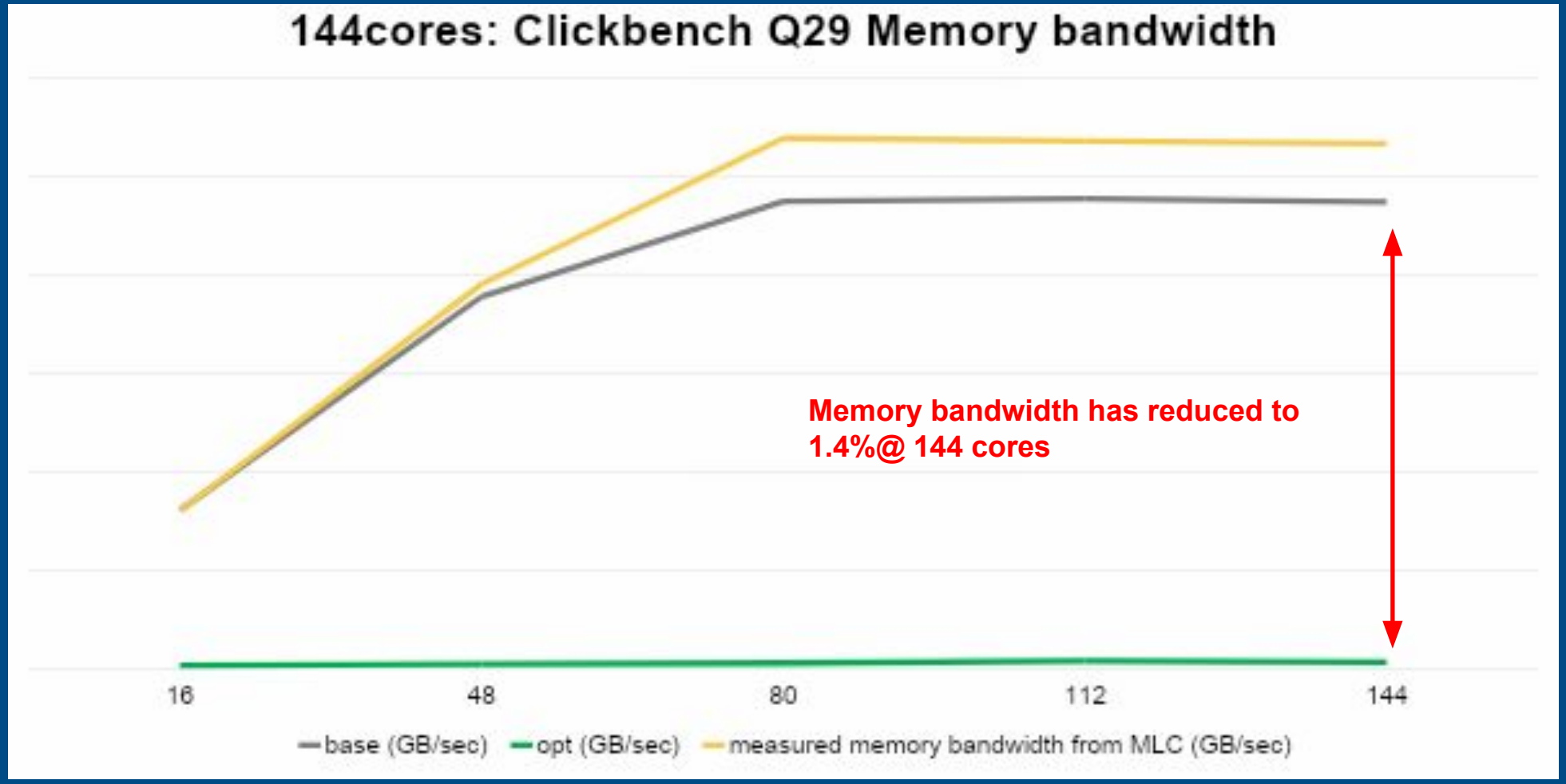
- Solve the problem at the sql frontend.
- Reduce the arithmetic in the aggregate sum function. Rewrite `sum(column + literal)` into two individual functions.
  - E.g. sum(column + literal) -> sum(column) + literal * count(column)

Rewrite -- Performance Impact

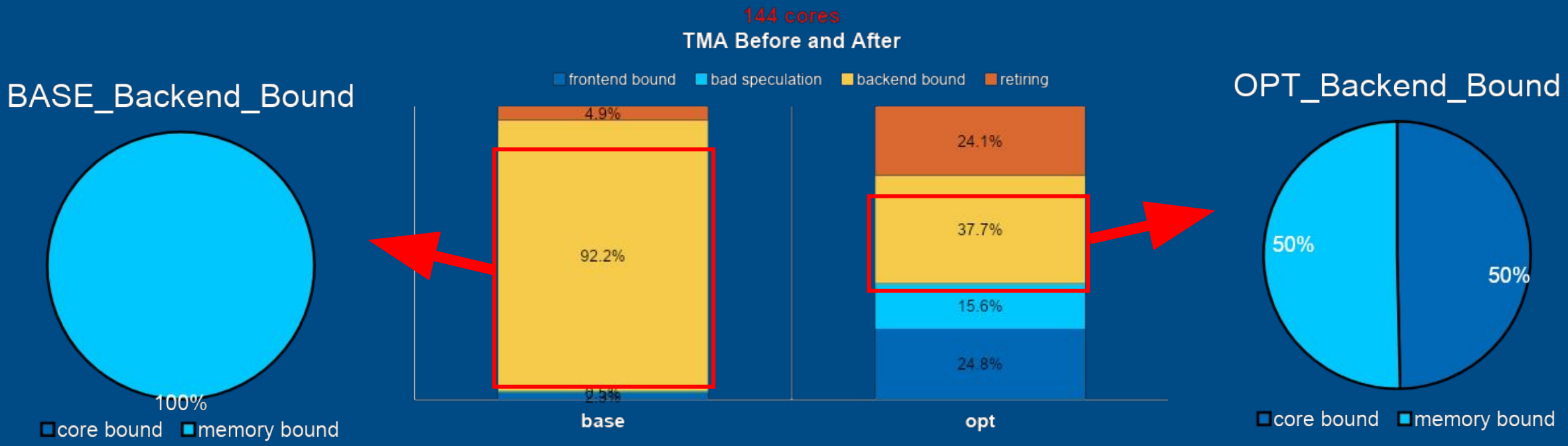




Rewrite -- Memory Bandwidth Before and After

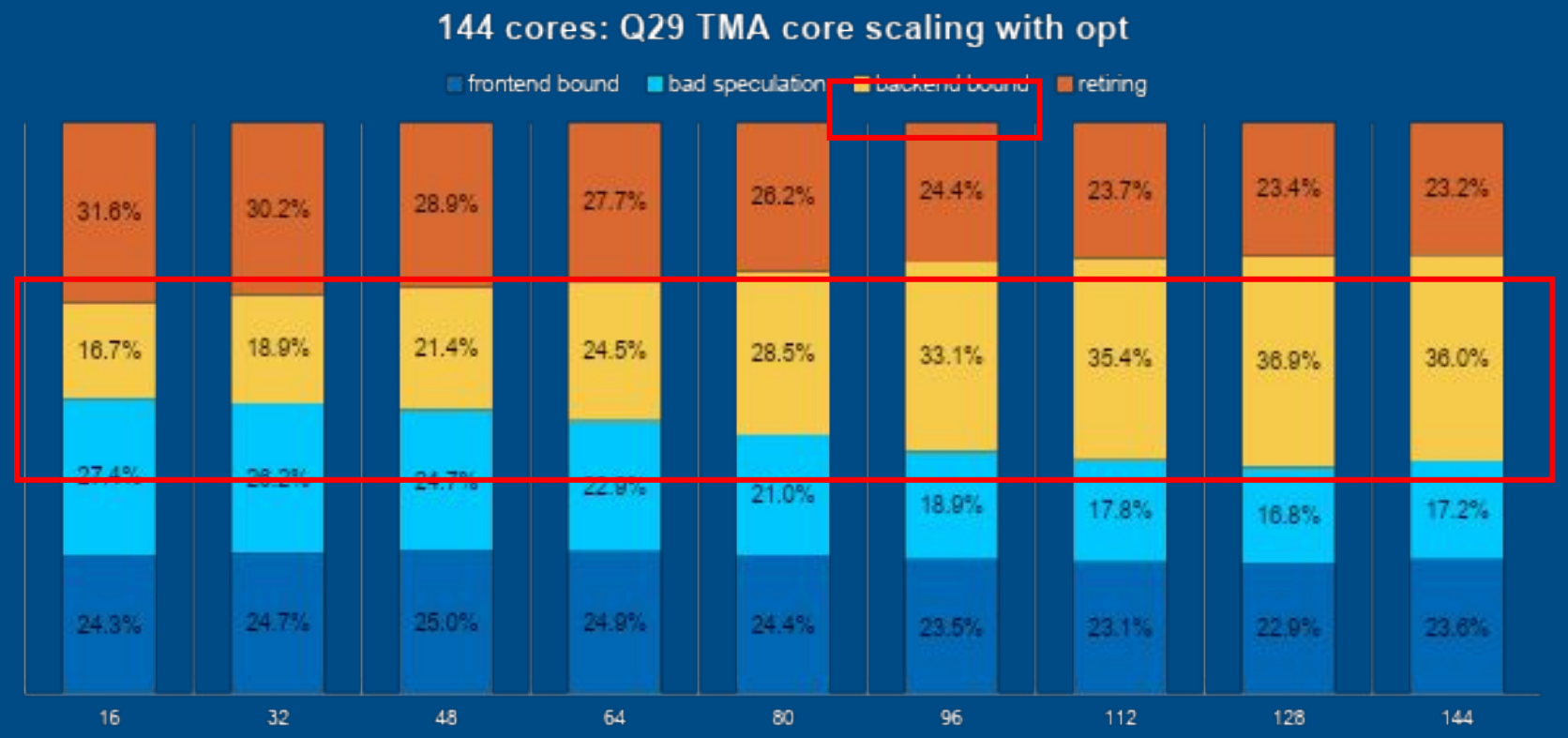


Rewrite -- TMA Analysis Before and After



- Clickbench Q29: SELECT SUM(ResolutionWidth), SUM(ResolutionWidth + 1), SUM(ResolutionWidth + 2), …, SUM(ResolutionWidth + 89) FROM hits;
- Optimization: sum(column + literal) -> sum(column) + literal * count(column)

Rewrite -- TMA Core Scaling with Optimization



Rewrite -- Code Path Reduced

					opt/base	
	Cycles(%)	Instructions(%)	Cycles(%)	Instructions(%)	cycles	instructions
void DB::impl_::BinaryOperation<short, char8_t, DB::PlusImpl<short, char8_t>, int>::process<(DB::impl_::OpCase)2>(short const*, char8_t const*, int*, unsigned long, DB::PODArray<char8_t, 4096ul, Allocator<false, false>, 63ul, 64ul> const*)	40.0%	39.7%	0.0%	0.0%	0.0%	0.0%
DB::AggregateFunctionSumData<long>::addManyImplAVX2	27.6%	45.9%	2.3%	4.1%	0.1%	1.1%
DB::AggregateFunctionCount::addBatchSinglePlace	0.0%	0.0%	0.01%	0.0%		0
Others	32.4%	14.5%	97.7%	95.9%	10.6%	21.9%
Total	100.0%	100.0%	100.0%	100.0%	1.6%	7.4%

Total cycles 1.6% as before
Total instructions 7.4% as before

sum(column + literal) -> sum(column) + literal * count(column)

`Plus` cycles eliminated

`Sum` cycles 0.1% as before
Instructions 1.1% as before

`count` cycles only occupied 0.01%
In the new total cycles

Q29: SELECT SUM(ResolutionWidth), SUM(ResolutionWidth + 1), SUM(ResolutionWidth + 2), …, SUM(ResolutionWidth + 89) FROM hits;

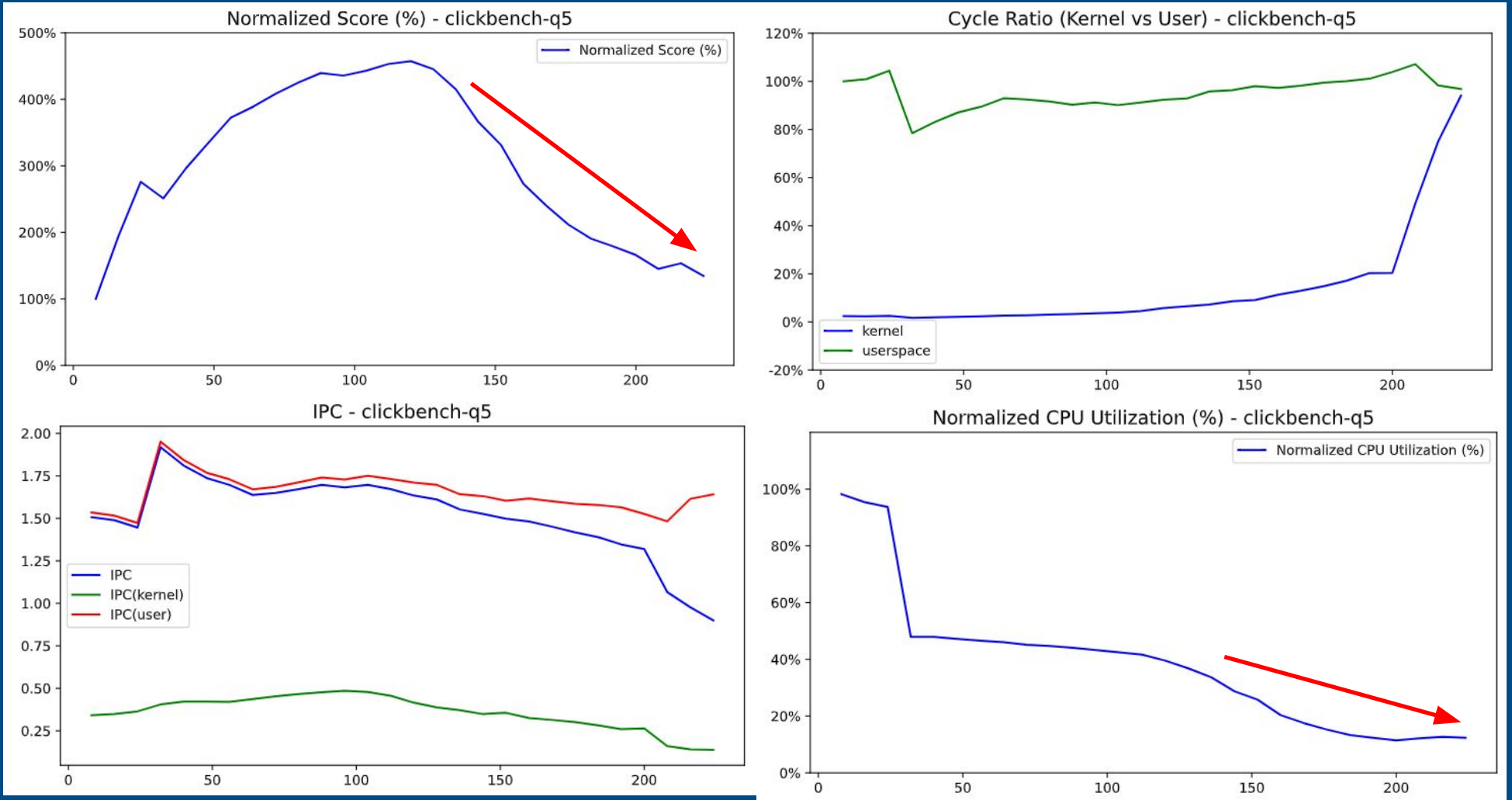
Increase the thread level parallelism

- Examples:
  - Convert hashsets in parallel before merge. Q5: 2.64x @ 2 x 112 vCPUs.
  - Optimize the merge of singleLevelHash. 2.35x @ 2x 80 vCPUs.
  - Support parallel merge with key. Q8: 10.3%, Q9: 7.6% @ 2 x 80 vCPUs.
  - Release more num_streams if data is small. Q39: 3.3x, Q36: 2.6x @ 2x80 vCPUs.
  - Change the default threshold to enable hyper threading. overall geomean: 13.2% @32 vCPUs, 7.6% @48 vCPUs

Parallel merge -- The Problem

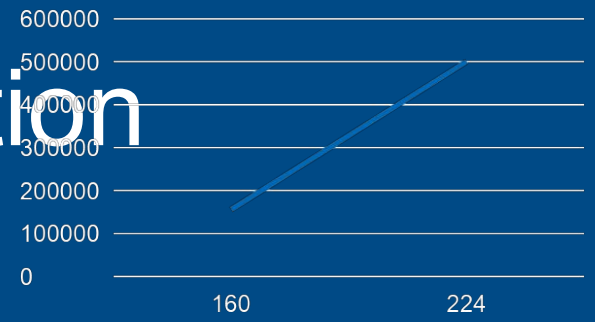
Score and CPU utilization is not scalable. (fixed working amount)

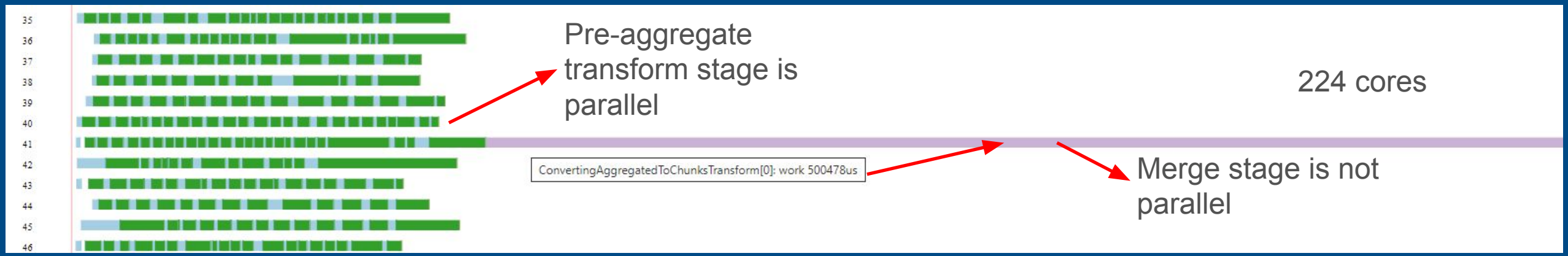
Q5: <SELECT COUNT(DISTINCT SearchPhrase) FROM hits;>



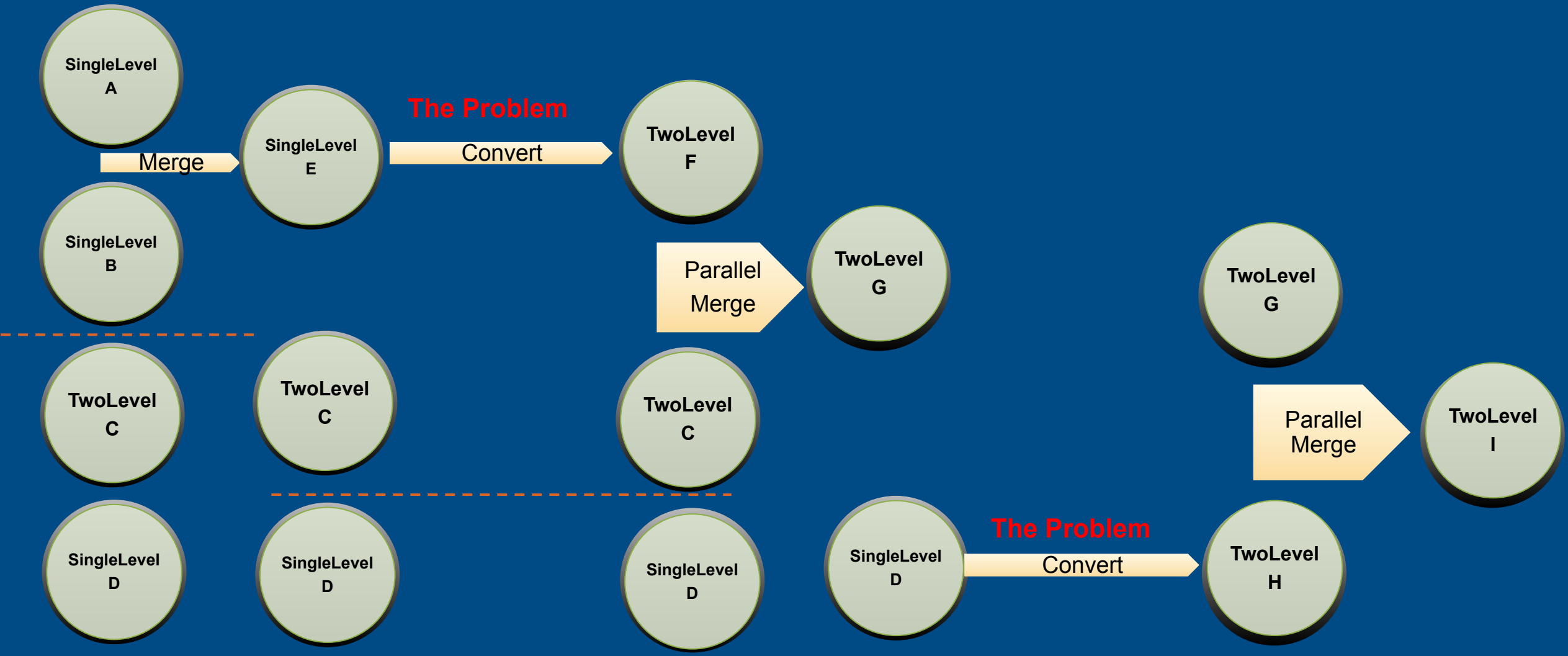
Parallel merge -- Pipeline visualization

pipeline for Q5 (fixed working amount)

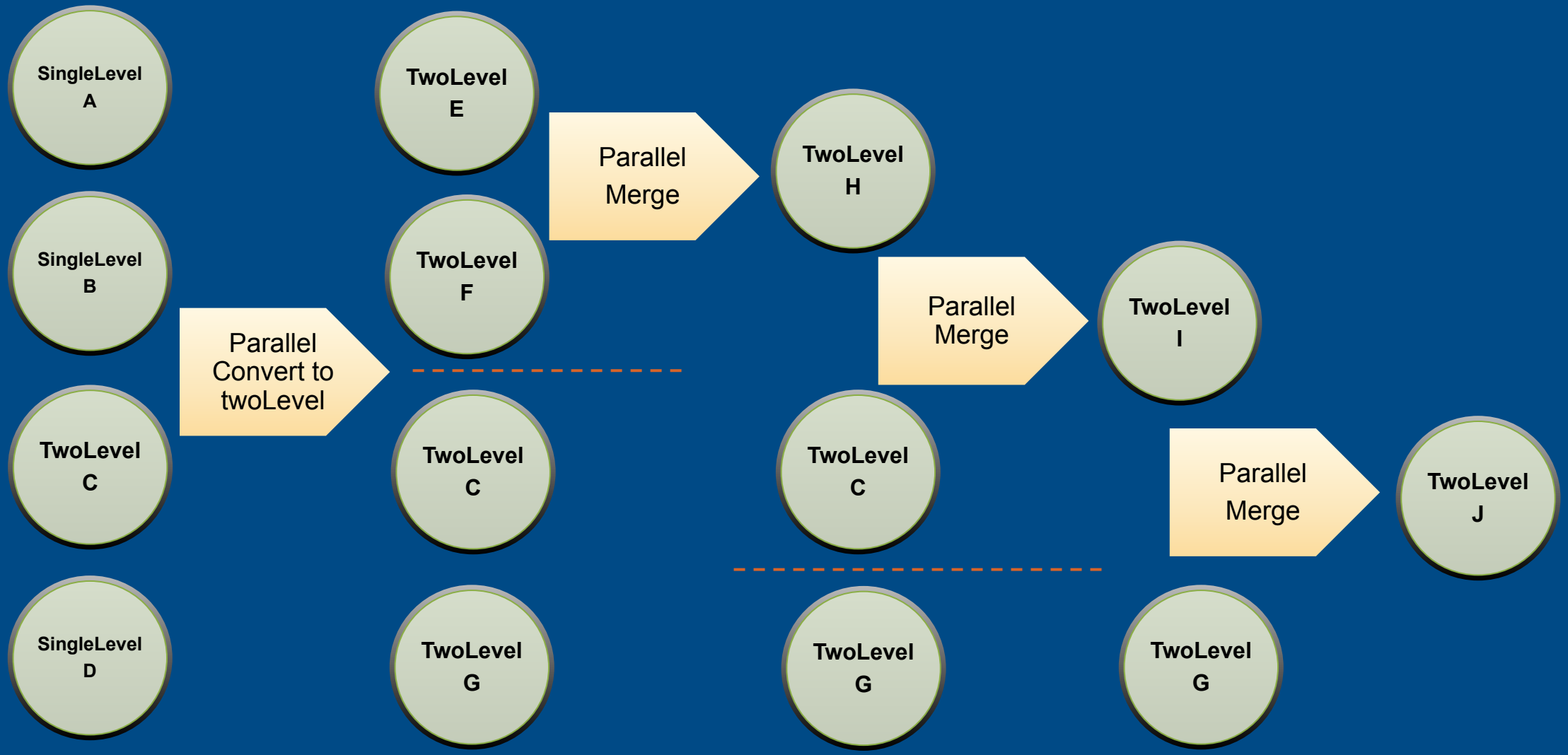




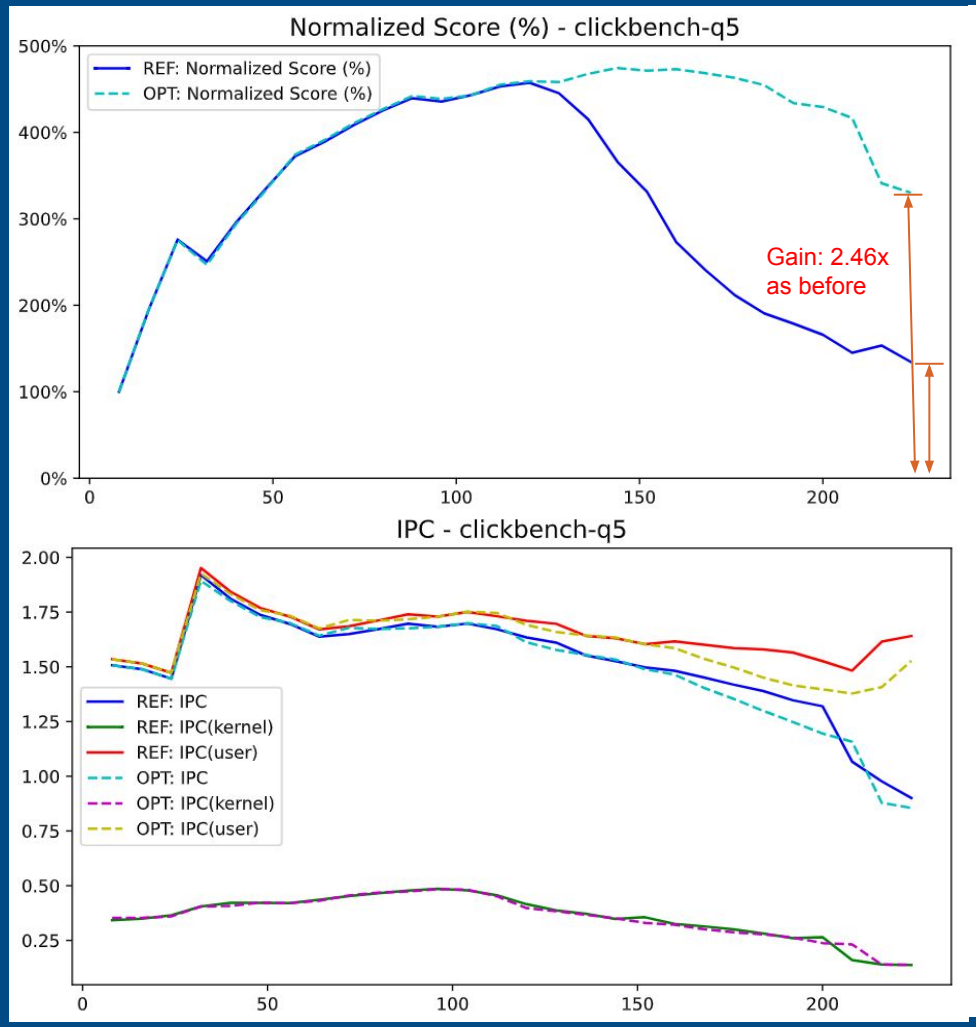
Parallel merge -- Merge in ClickHouse

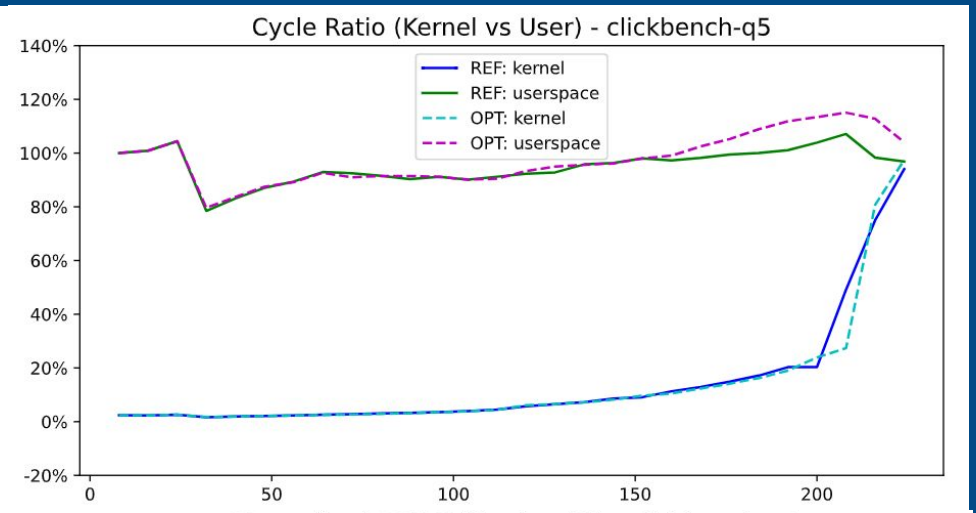


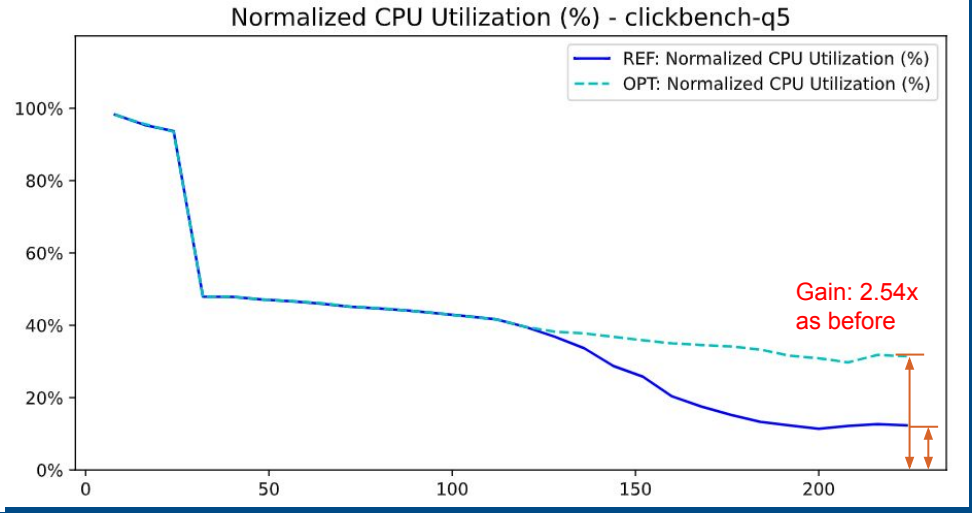
Parallel merge -- Merge optimization



Parallel merge -- The results







Parallel merge -- Question

- Can we optimize further when all the hashtables are singleLevel?

Search algorithm

- Example
  - Optimize the StringSearcher with SIMD by searching first two chars. Q20: 35%, other StringSearcher related Q: 10%, overall geomean: 4.1% @ 2 x 80 vCPUs

```
        while (haystack < haystack_end && haystack_end - haystack >= needle_size)
        {
#ifdef __SSE4_1__
            /// Compare the [0:15] bytes from haystack and broadcasted 16 bytes vector from first character of needle.
            /// Compare the [1:16] bytes from haystack and broadcasted 16 bytes vector from second character of needle.
            /// Bit AND the results of above two comparisons and get the mask.
            if ((haystack + 1 + n) <= haystack_end && pageSafe(haystack + 1))
            {
                const __m128i haystack_characters_from_1st = _mm_loadu_si128(reinterpret_cast<const __m128i *>(haystack));
                const __m128i haystack_characters_from_2nd = _mm_loadu_si128(reinterpret_cast<const __m128i *>(haystack + 1));
                const __m128i comparison_result_1st = _mm_cmpeq_epi8(haystack_characters_from_1st, first_needle_character_vec);
                const __m128i comparison_result_2nd = _mm_cmpeq_epi8(haystack_characters_from_2nd, second_needle_character_vec);
                const __m128i comparison_result_combined = _mm_and_si128(comparison_result_1st, comparison_result_2nd);
                const uint16_t comparison_result_mask = _mm_movemask_epi8(comparison_result_combined);
                /// If the mask = 0, then first two characters [0:1] from needle are not in the [0:17] bytes of haystack.
                if (comparison_result_mask == 0)
```

Q & A

Thank you!