Top-down Microarchitecture Analysis

Top-down Microarchitecture Analysis

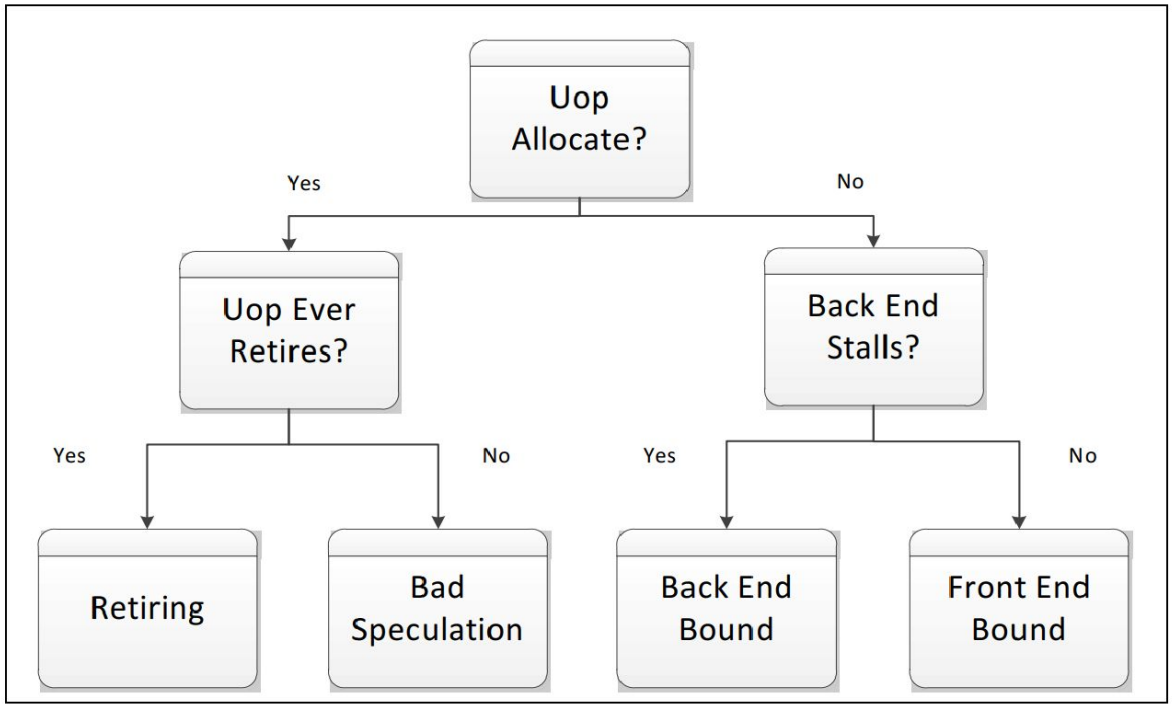
- Top-down Microarchitecture Analysis (TMA) methodology identifies ineffective usage of CPU microarchitecture by a program.

- It characterizes the **bottleneck** of a workload and allows locating the **exact place** in the source code where it occurs.

- It abstracts away the intricacies of the CPU microarchitecture and is relatively easy to use even for inexperienced developers.

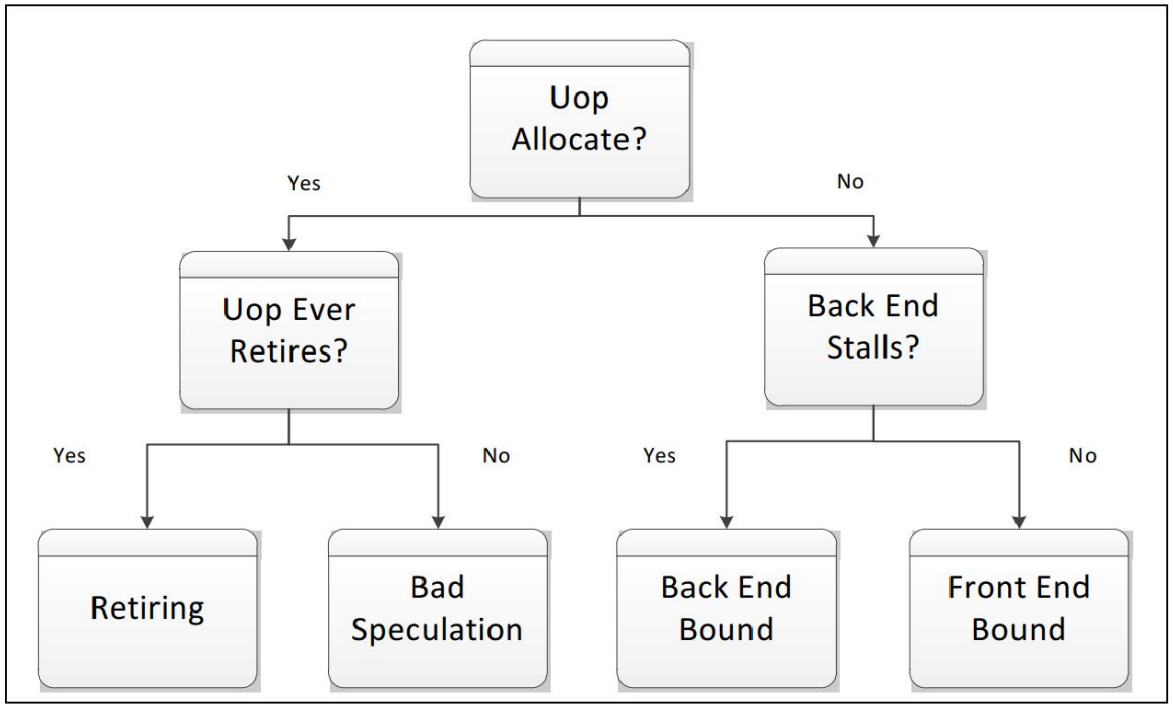
Top-down Microarchitecture Analysis

- TMA is a **hierarchical** organization of event-based metrics.
- It identifies the dominant performance bottlenecks in an application.
- It shows, on average, how well the CPU's pipelines are being utilized while running an application.



Top-down Microarchitecture Analysis

- The allocation of a *µop* can fail for one of two reasons:
  - The CPU were not able to fetch and decode it (**Frontend Bound**),
  - the **Backend** was overloaded with work, (**Backend Bound**).
- If a *µop* was allocated but never retired, this means it came from a mispredicted path (**Bad Speculation**).
- Finally, **Retiring** represents a normal execution.

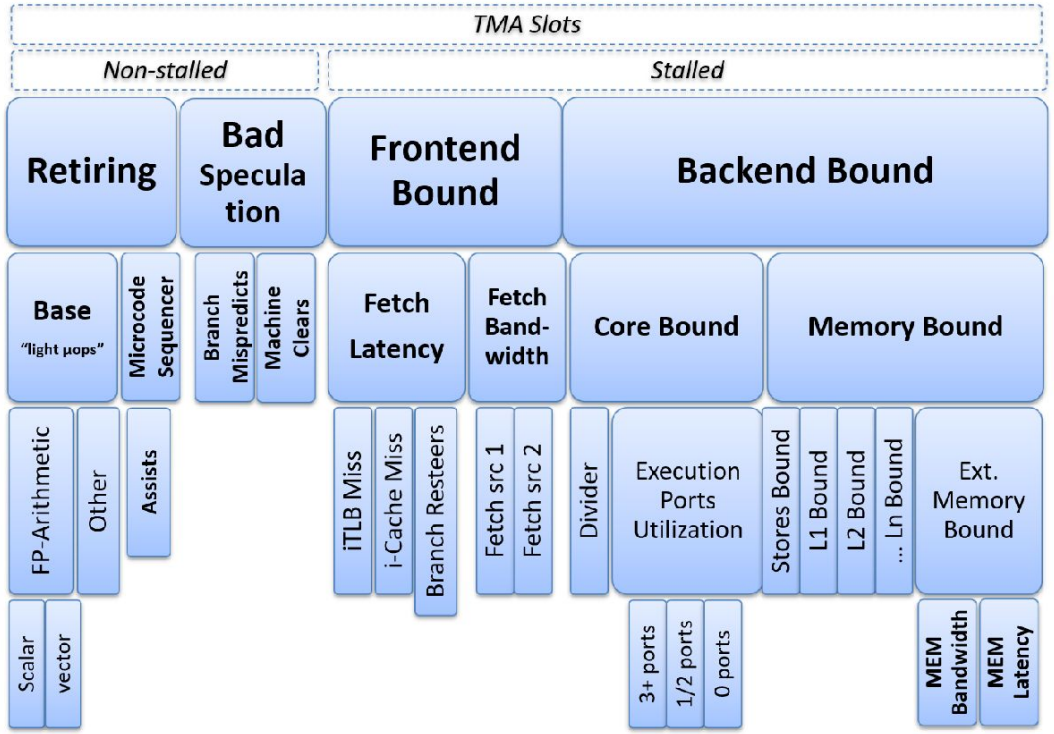


TMA hierarchy

TMA monitors a specific set of events and calculates metrics.

Using these metrics, TMA characterizes the program by assigning it to one of the four high-level buckets.

Each of the four high-level categories has several nested levels, which *CPU vendors* may choose to implement differently.



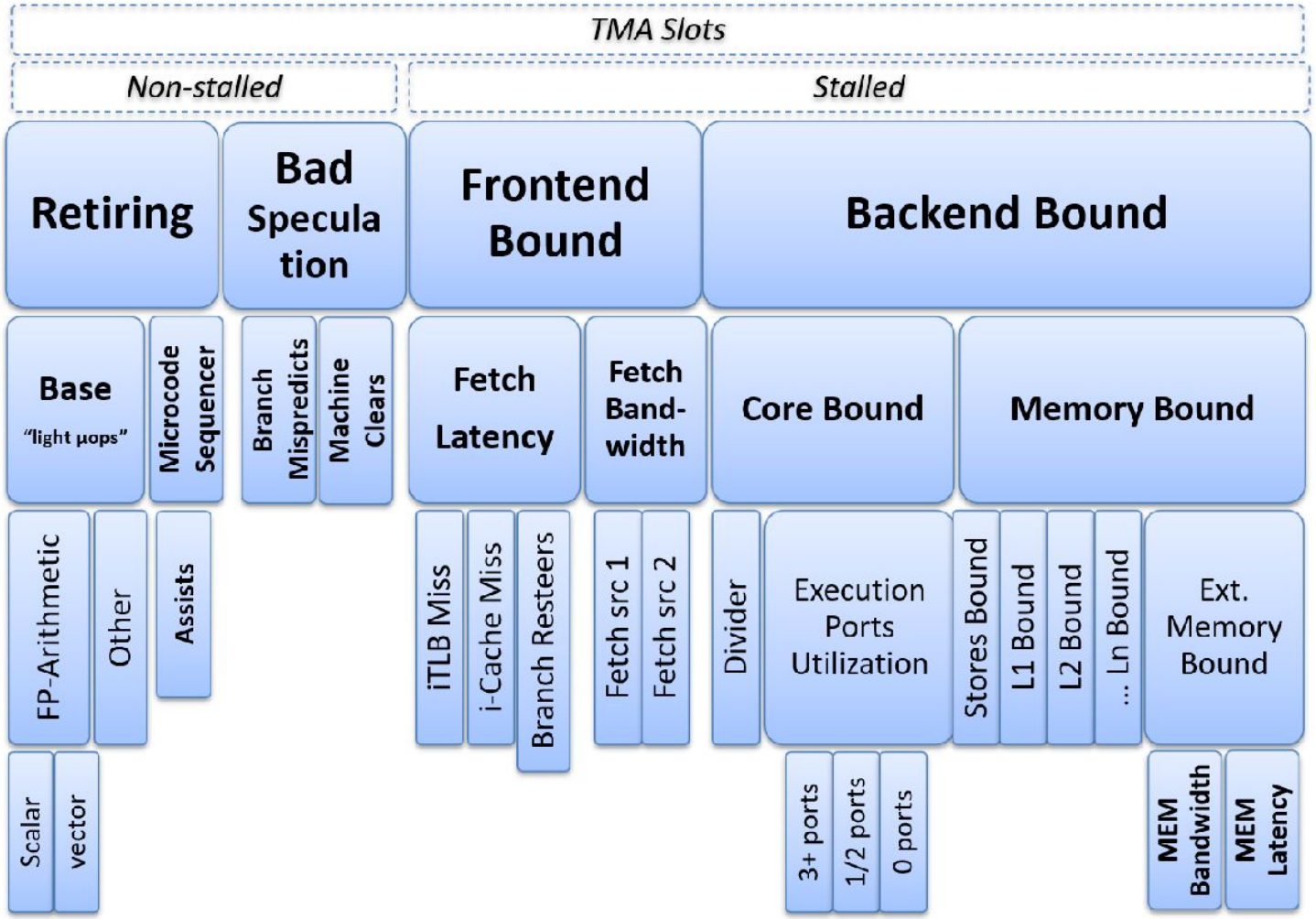
TMA hierarchy

Front End Bound

- Code Duplication
- Code Layout (Locality)
- Frequent Branching
- Unnecessary Work

Back End Bound

- *Core Bound*
  - Data Dependencies
  - Divisions and Special Functions
- *Memory Bound*
  - False Sharing
  - Scattered Memory Accesses



TMA workflow

- The workflow is designed to “drill down” to lower levels of the TMA hierarchy until we get to the very specific classification of the bottleneck.
    - For example, if a big portion of the program execution was stalled by memory accesses (**Backend Bound**) we can collect metrics specific to the Memory Bound bucket only, identifying an L3 Bound.
- The top two levels of TMA metrics are expressed in the percentage of all pipeline slots. It allows TMA to give an accurate representation of CPU microarchitecture utilization.
    - starting from Level 3, buckets may be expressed in clocks and stalls cycles (we don’t need to compare with other TMA buckets)

TMA workflow

TMA workflow is split in three stages:

1. **Bottleneck identification**: we collect events to classify the bottleneck.
2. **Bottleneck localization**: we sample on the event that characterize the bottleneck to identify the line in the source code that contributes to the bottleneck.
3. **Bottleneck removal:** we change the code to remove the bottleneck (if possible)
4. We go back to 1.

Case Study: Reduce The Number of Cache Misses with TMA

- We used a simple benchmark to show the TMA workflow. Since TMA is very effective even if you see the application for the first time we don't show the source code(1).

- Here is a short description: the benchmark allocates a 200 MB array on the heap, then enters a loop of 100M iterations. On every iteration of the loop, it generates a random index into the allocated array, performs some dummy work, and then reads the value from that index.

- To obtain TMA metrics Level 2, 3, and further, we will use the toplev tool that is a part of pmu-tools(2)

(1) Code is here: https://github.com/dendibakh/dendibakh.github.io/tree/master/_posts/code/TMAM

(2) PMU tools - https://github.com/andikleen/pmu-tools

Step 1: Identify the Bottleneck

As a first step, we run our microbenchmark and collect a limited set of events that will help us calculate Level 1 metrics.

```
$ perf stat -- ./benchmark.exe
...
TopdownL1 (cpu_core) # 53.4 % tma_backend_bound <==
                     # 0.2 % tma_bad_speculation
                     # 13.8 % tma_frontend_bound
                     # 32.5 % tma_retiring
...
```

Thus, the application is bound by the **CPU backend**

Step 1: Identify the Bottleneck

Now we collect L2 metrics.

```
$~/pmu-tools/toplev.py ....
...
# Level 1
S0-C0 Frontend_Bound:              13.92 % Slots
S0-C0 Bad_Speculation:              0.23 % Slots
S0-C0 Backend_Bound:               53.39 % Slots
S0-C0 Retiring:                  32.49 % Slots
# Level 2
S0-C0 Frontend_Bound.FE_Latency:       12.11 % Slots
S0-C0 Frontend_Bound.FE_Bandwidth:      1.84 % Slots
S0-C0 Bad_Speculation.Branch_Mispred:   0.22 % Slots
S0-C0 Bad_Speculation.Machine_Clears:   0.01 % Slots
S0-C0 Backend_Bound.Memory_Bound:      44.59 % Slots <==
S0-C0 Backend_Bound.Core_Bound:      8.80 % Slots
...
```

Step 1: Identify the Bottleneck

Now we collect L3 metrics.

```
$~/pmu-tools/toplev.py ....
...
# Level 3
S0-C0-T0 BE_Bound.Mem_Bound.L1_Bound:    4.39 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.L2_Bound:    2.42 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.L3_Bound:    5.75 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.DRAM_Bound:   47.11 % Stalls <==
S0-C0-T0 BE_Bound.Mem_Bound.Store_Bound:    0.69 % Stalls
S0-C0-T0 BE_Bound.Core_Bound.Divider:    8.56 % Clocks
S0-C0-T0 BE_Bound.Core_Bound.Ports_Util:  11.31 % Clocks...
```

Step 1: Identify the Bottleneck

- The bottleneck is in **DRAM_Bound:** many memory accesses miss in all levels of caches and go all the way down to the main memory.

- We can confirm this collecting the absolute number of L3 cache misses for the program.

Step 1: Identify the Bottleneck

- The bottleneck is in **DRAM_Bound:** many memory accesses miss in all levels of caches and go all the way down to the main memory.

- We can confirm this collecting the absolute number of L3 cache misses for the program.

```
$ perf stat -e cycles,cycle_activity.stalls_l3_miss -- ./benchmark.exe

32226253316 cycles
19764641315 cycle_activity.stalls_l3_miss
```

Step 1: Identify the Bottleneck

- The bottleneck is in **DRAM_Bound:** many memory accesses miss in all levels of caches and go all the way down to the main memory.

- We can confirm this collecting
for the program.

The `CYCLE_ACTIVITY.STALLS_L3_MISS` event counts cycles when execution stalls, while the L3 cache miss demand load is outstanding

```
$ perf stat -e cycles,cycle_activity.stalls_l3_miss -- ./benchmark.exe

32226253316 cycles
19764641315 cycle_activity.stalls_l3_miss
```

Step 1: Identify the Bottleneck

- The bottleneck is in **DRAM_Bound:** many memory accesses miss in all levels of caches and go all the way down to the main memory.

- We can confirm this collecting
for the program.

CPU is in stall for 60% of clock cycles

```
$ perf stat -e cycles,cycle a      ty.stalls_l3_miss -- ./benchmark.exe

32226253316 cycles
19764641315 cycle_activity.stalls_l3_miss
```

Step 2: Locate the Place in the Code

- The second step in the TMA process is to locate the place in the code sampling using the event related to the bottleneck.

- The recommended way to find such an event is to run `toplev` with the `--show-sample` option that suggests the perf record command line

- Correspondence between performance bottlenecks and performance events can be found with the TMA metrics table[1].

(1) TMA metrics - https://github.com/intel/perfmon/blob/main/TMA_Metrics.xlsx

Step 2: Locate the Place in the Code

In our case, we sample on `MEM_LOAD_RETIRED.L3_MISS_PS` precise event.

```
$ perf record -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRED.L3_MISS/ppp --
./benchmark.exe
$ perf report -n --stdio
...
# Samples: 33K of event 'MEM_LOAD_RETIRED.'L3_MISS
# Event count (approx.): 71363893
# Overhead   Samples  Shared Object Symbol
# ........ ......... ............... ..................
#
   99.95% 33811    benchmark.exe [.] foo
    0.03%    52    [kernel]      [k] get_page_from_freelist
    0.01%     3    [kernel]      [k] free_pages_prepare
    0.00%     1    [kernel]      [k] free_pcppages_bulk
```

Step 2: Locate the Place in the Code

In our case, we sample on `MEM_LOAD_RETIRED.L3_MISS_PS` precise event.

```
$ perf record -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRED.L3_MISS/ppp --
./benchmark.exe
$ perf report -n --stdio
...
# Samples: 33K of event 'MEM_LOAD_RE
# Event count (approx.): 71363893
# Overhead   Samples  Shared Object D
# ........ ......... ............. ..................
#
   99.95% 33811    benchmark.exe [.] foo
    0.03%    52     [kernel]      [k] get_page_from_freelist
    0.01%     3     [kernel]      [k] free_pages_prepare
    0.00%     1     [kernel]      [k] free_pcppages_bulk
```

Almost all L3 misses are caused by memory accesses in function foo inside executable benchmark.exe.

Step 2: Locate the Place in the Code

In our case, we sample on `MEM_LOAD_RETIRED.L3_MISS_PS` precise event.

```
$ perf record -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRED.L3_MISS/ppp --
./benchmark.exe
$ perf report -n --std
...
# Samples: 33K of event 'MEM_LOAD_RETIRED.L3_MISS
# Event count (approx.): 71363893
# Overhead   Samples  Shared Object Symbol
# ........ ......... ............... ..................
#
   99.95% 33811    benchmark.exe [.] foo
    0.03%    52    [kernel]      [k] get_page_from_freelist
    0.01%     3    [kernel]      [k] free_pages_prepare
    0.00%     1    [kernel]      [k] free_pcppages_bulk
```

What is a precise event?

Precise Events

- Interrupt-based sampling is based on counting a specific performance event and waiting until it overflows. When an overflow happens, it takes a processor some time to stop the execution and tag the instruction that caused the overflow.

- The distance between the IP (instruction address) that caused the event to the IP where the event is tagged can be hundreds of processor instructions, causing confusion in performance analysis.

- With Intel PEBS (*Precise Event-Based Sampling*), the EventingIP field in the PEBS record indicates the instruction that caused the event.

-

- This is typically available only for a subset of supported events, called “Precise Events”.

Step 2: Locate the Place in the Code

`Perf annotate` locates the assembly line at which the bottleneck occurs

```
$ perf annotate --stdio -M intel foo
Percent | Disassembly of benchmark.exe for MEM_LOAD_RETIRED.L3_MISS
------------------------------------------------------------------
     : Disassembly of section .text:
     :
     : 0000000000400a00 <foo>:
     : foo():

   0.00 : 400a00: nop DWORD PTR [rax+rax*1+0x0]
   0.00 : 400a08: nop DWORD PTR [rax+rax*1+0x0]
      ... # more NOPs
 100.00 : 400e07: mov rax,QWORD PTR [rdi+rsi*1] <==
      ...
   0.00 : 400e13: xor rax,rax
   0.00 : 400e16: ret
```

Step 2: Locate the Place in the Code

The source code of the `main()` function:

- allocate an array bigger than the L3 cache.
- Generates a random index into array a and passes this index to the `foo()` function (in the `rdi` and `rsi` registers)

```
extern "C" { void foo(char* a, int n); }
const int _200MB = 1024*1024*200;
int main() {
    char* a = new char[_200MB]; // 200 MB buffer
    ...
    for (int i = 0; i < 100000000; i++) {
       int random_int = distribution(generator);
       foo(a, random_int);
    }
...
}
```

Step 2: Locate the Place in the Code

`Perf annotate` locates the assembly line at which the bottleneck occurs

```
$ perf annotate --stdio -M intel foo
Percent | Disassembly of benchmark.exe for MEM_LOAD_RETIRED.L3_MISS
------------------------------------------------------------------
     : Disassembly of section .text:
     :
     : 0000000000400a00 <foo>:
     : foo():

   0.00 : 400a00: nop DWORD PTR [rax+rax*1+0x0]
   0.00 : 400a08: nop DWORD PTR [rax+rax*1+0x0]
        ... # more NOPs
 100.00 : 400e07: mov rax,QWORD PTR [rdi+rsi*1] <==
        ...
   0.00 : 400e13: xor rax,rax
   0.00 : 400e16: ret
```

a+random_int

Step 3: Fix the Issue

The `foo()` function executes some dummy work emulated with NOPs.

- This creates a time window between the computing of `random_int` and the moment when we load `*(a+random_int)`
- The presence of the time window allows us to start prefetching the memory location in parallel with the dummy work.

Step 3: Fix the Issue

The `foo()` function executes some dummy work emulated with NOPs.

- This creates a time window between the computing of `random_int` and the moment when we load `*(a+random_int)`
- The presence of the time window allows us to start prefetching the memory location in parallel with the dummy work.

```
for (int i = 0; i < 100000000; i++) {
    int random_int = distribution(generator);
+   __builtin_prefetch(a+random_int,0,1);
    foo(a, random_int);
}
```

Step 3: Fix the Issue

The `foo()` function executes some dummy work emulated with NOPs.

- This creates a time window between the computing of `random_int` and the moment when we load `*(a+random_int)`
- The presence of the time window allows us to start prefetching the memory location in parallel with the dummy work.

```
for (int i = 0; i < 100000000; i++) {
    int random_int = distribution(generator);
+   __builtin_prefetch(a+random_int,0,1);
    foo(a, random_int);
}
```

- Memory prefetching decreases execution time from 8.5 seconds to 6.5 seconds.
- The number of `CYCLE_ACTIVITY.STALLS_L3_MISS` events becomes goes from 19 billion down to 2 billion.

TMA Remarks

Back-End Bound can be divided in

1. Memory bound:
    - Performance limited by memory subsystem latency/bandwidth.

2. Core Bound:
    - Performance limited by the CPU execution ports

TMA Remarks

Memory bound:

- The majority of un-tuned applications will be Memory bound.

- Most of the metrics identify which level of the memory hierarchy is the bottleneck. Optimizations should focus on moving data closer to the core.

- Store Bound can indicate dependencies - such as when loads in the pipeline depend on prior stores.

TMA Remarks

Core bound:

- Core Bound stalls are typically less common than memory bound.

- They occur when available computing resources are not sufficient. For example, a tight loop doing Floating Point (FP) arithmetic calculations on data that fits within cache. Some metrics helps to detect behaviors in this category:
  - the Divider metric identifies cycles when divider hardware is heavily used
  - the Port Utilization metric identifies competition for discrete execution units.

TMA Remarks

Front-End Bound can be reduced by:

- Changing code layout (co-locating hot code)
- Reducing branchy code
- code size optimization and compiler profile-guided optimization (PGO) are likely to reduce stalls in many cases.

Bad Speculation can be reduced through compiler techniques such as:

- Profile-Guided Optimization (PGO),
- avoiding indirect branches,
- eliminating error conditions that cause machine clears.

Correcting Bad Speculation issues may also help decrease the number of Front-End Bound stalls.

TMA Remarks

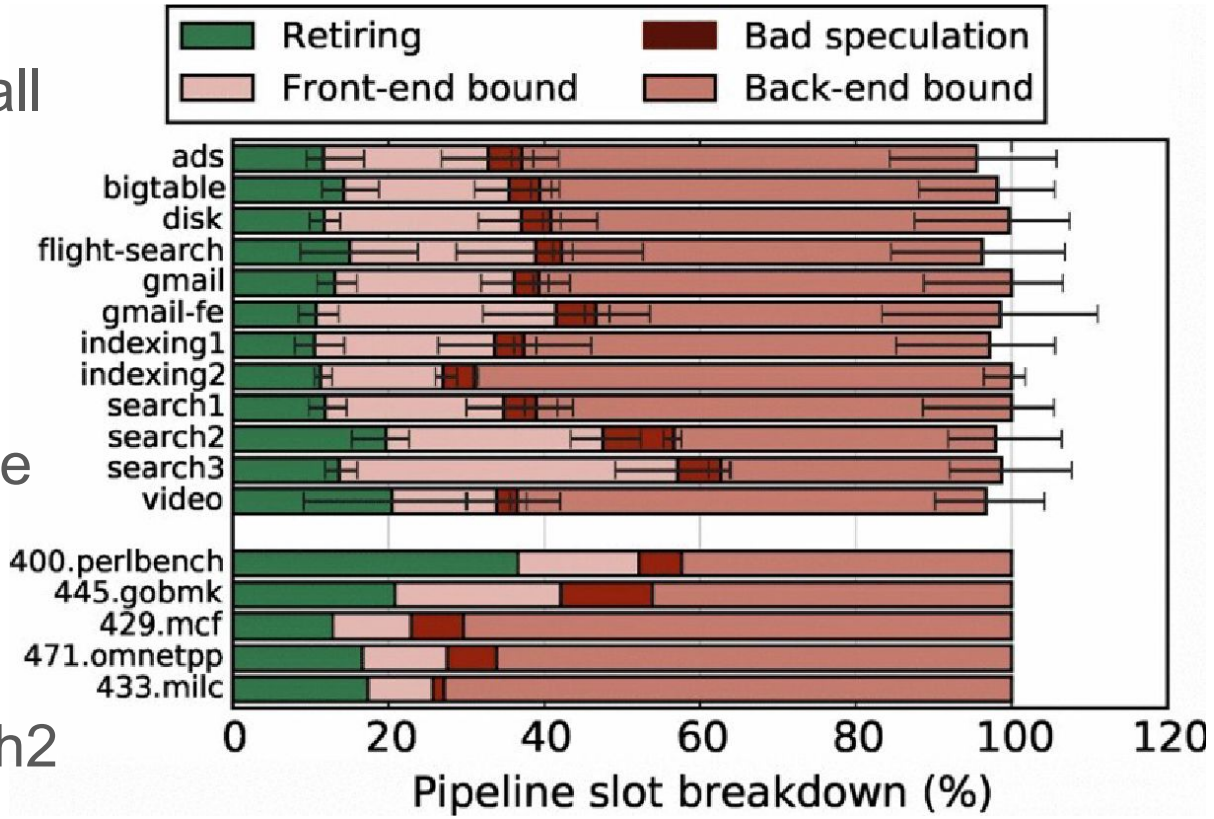
Ideally, after several iterations we would like to see the **Retiring** metric at 100%. Having 100% Retiring means a CPU is fully saturated and it crunches instructions at full speed.

- But it doesn’t say anything about the quality of those instructions. A program can spin in a tight loop waiting for a lock; that would show a high Retiring metric, but it doesn’t do any useful work.
- If you see a high Retiring metric but slow overall performance probably the program has a hotspot that was not vectorized.
- Another performance issue that will fall under the retiring category is heavy use of the micro-sequencer (e.g. to manage subnormal FP)

TMA Remarks

Achieving 100% Retiring in real-world applications is rare.

- Most data center workloads have a very small fraction in the Retiring bucket.
- BackendBound is the primary source of performance issues.
- FrontendBound is a bigger problem for data center workloads than in SPEC2006 because those applications typically have large codebases with poor locality.
- Finally, some workloads suffer from branch mispredictions more than others, e.g., search2 and 445.gobmk.



TMA breakdown of Google's datacenter workloads along with several SPEC CPU2006 benchmarks,

TMA with Intel Vtune profiler

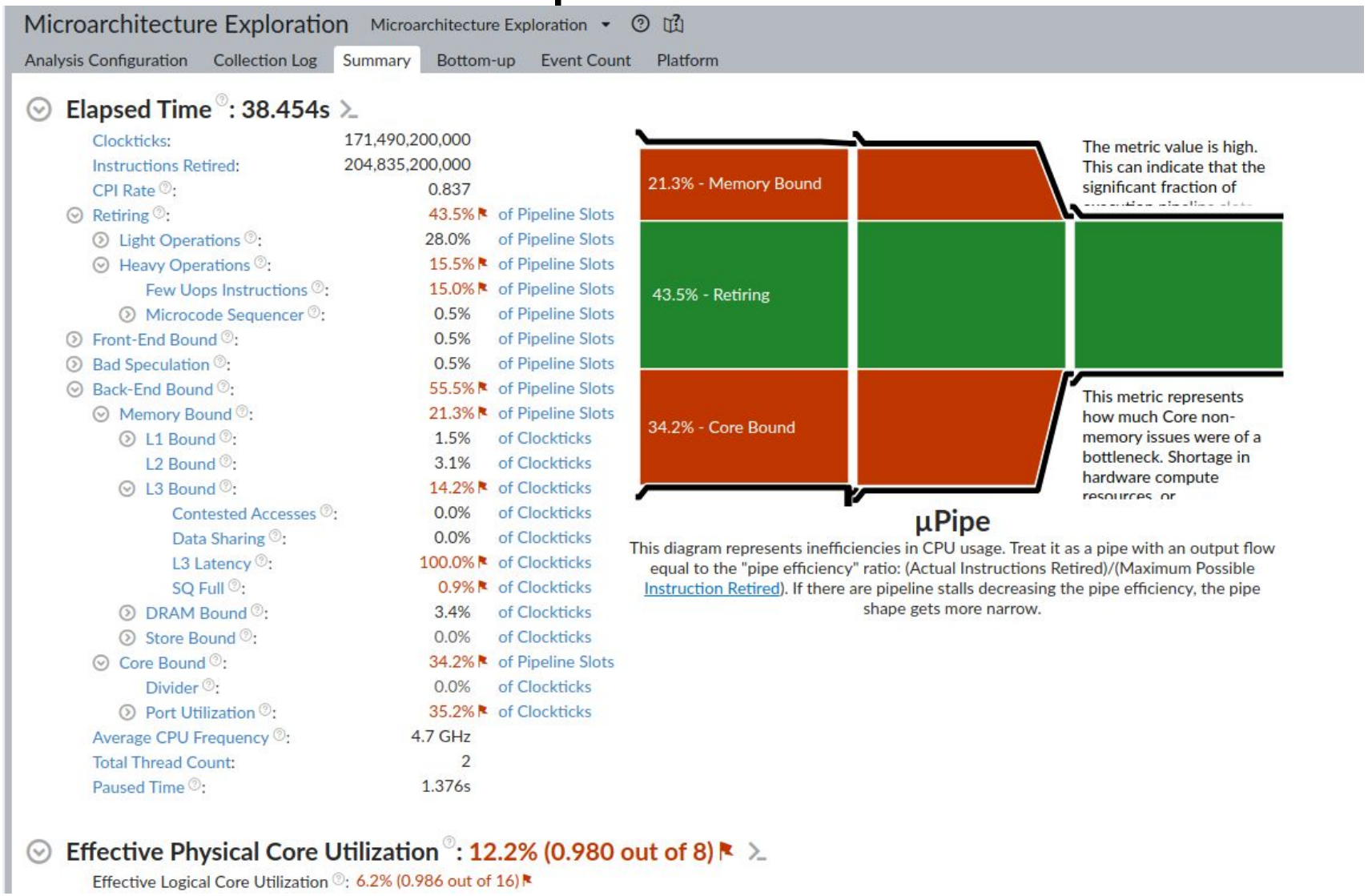
Linux `perf` is a lightweight, command-line profiling tool:

- available on all Linux systems
- Low overhead,
- easy to integrate in scripts
- limited visualization (requires external tools)
- No deep microarchitecture insight (limited to events counters)

Intel Vtune:

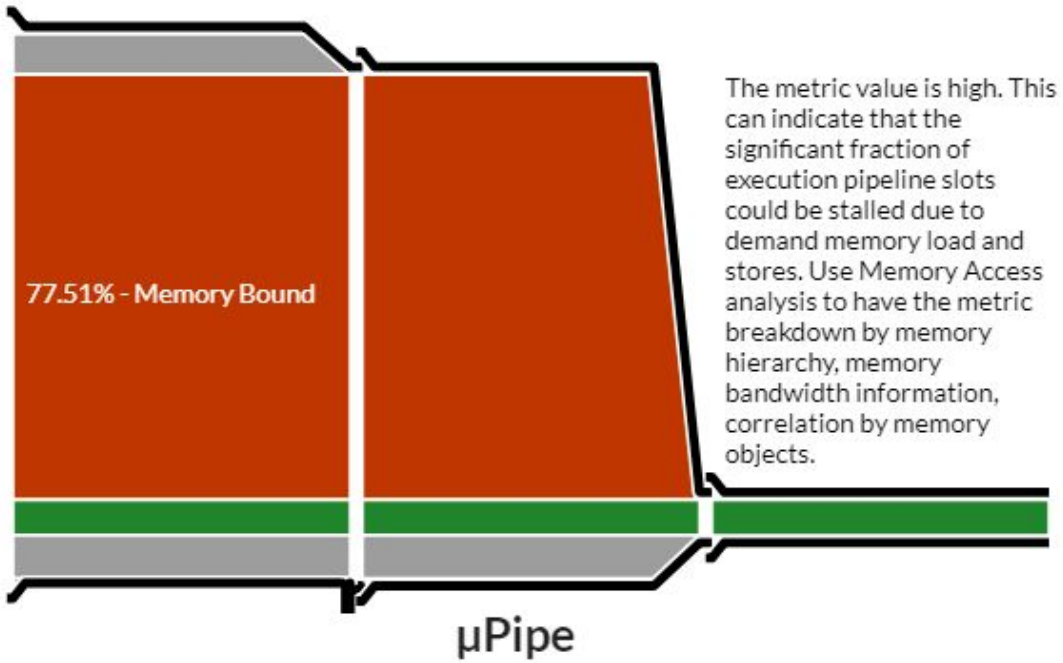
- Mode detailed microarchitecture-level metrics
- Detailed visualization
- Provide optimization hints

TMA with Intel Vtune profiler



Microarchitecture Pipe

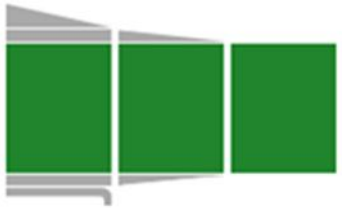
The Microarchitecture Pipe graphically identifies inefficiencies in the CPU utilization.



The output flow is the ratio: **Actual Instructions Retired/Possible Maximum Instruction Retired** (pipe efficiency). If there are pipeline stalls decreasing retiring, the pipe shape gets narrow.

Microarchitecture Pipe examples

A. 100% retiring



B. Memory bound execution

C. Core bound execution

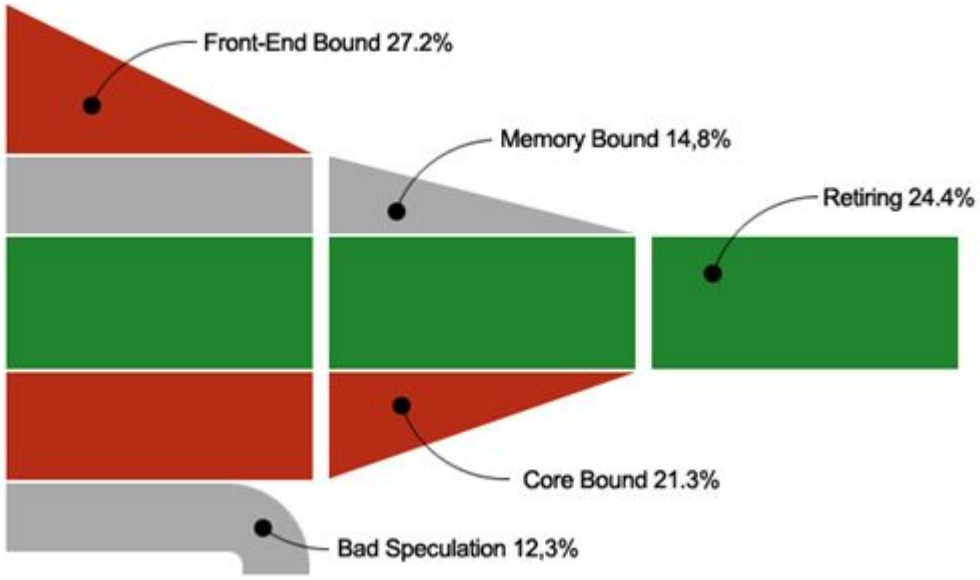
D. Front End bound execution

E. Bad speculation

F. Memory and Bad speculation

Microarchitecture Pipe Example

This an example of a pipe representing significant **Front-End Bound** and **Core Bound** issues limiting the whole efficiency to 24.4%:



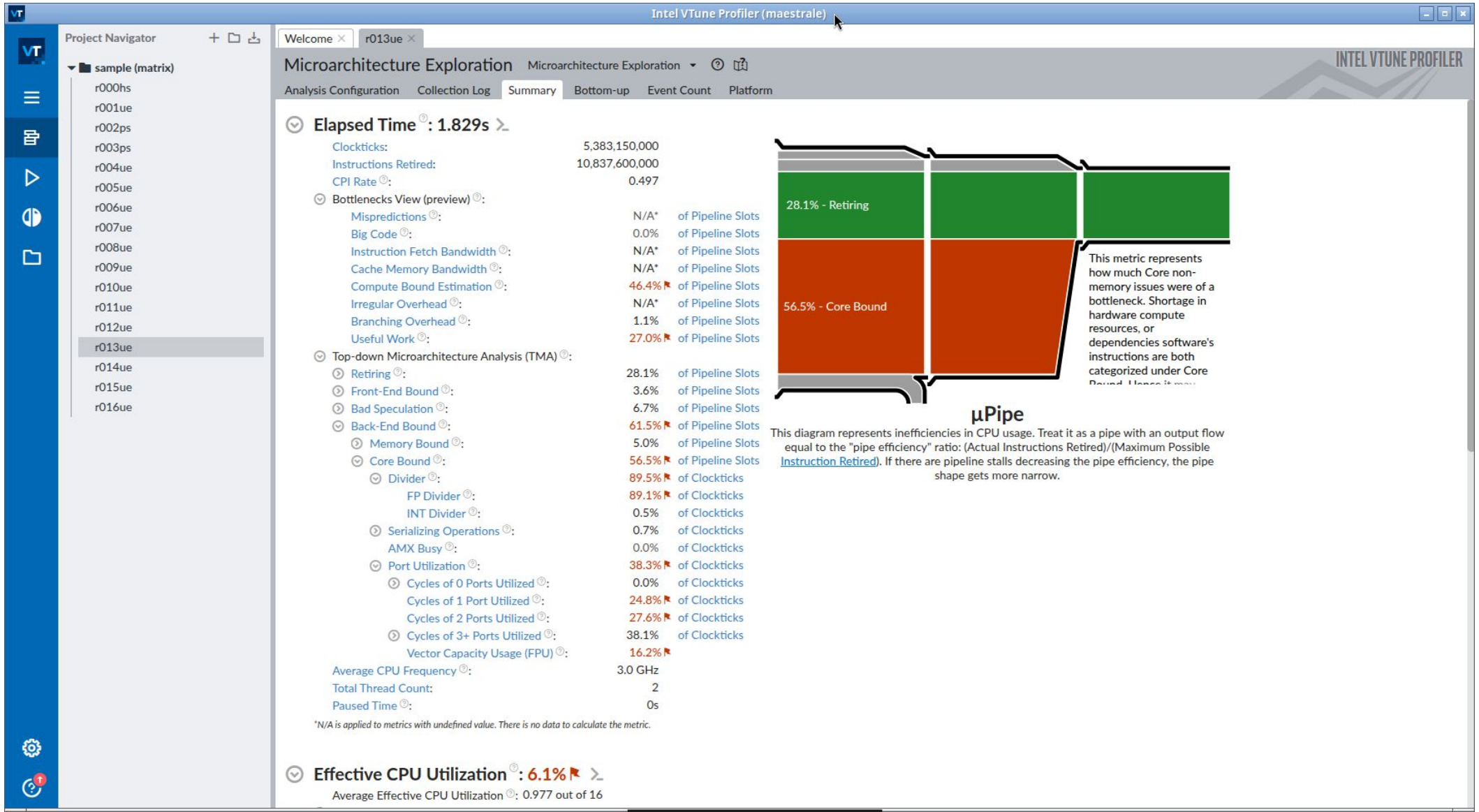
Case Study: n-body problem

We already optimized the n-body using vectorization

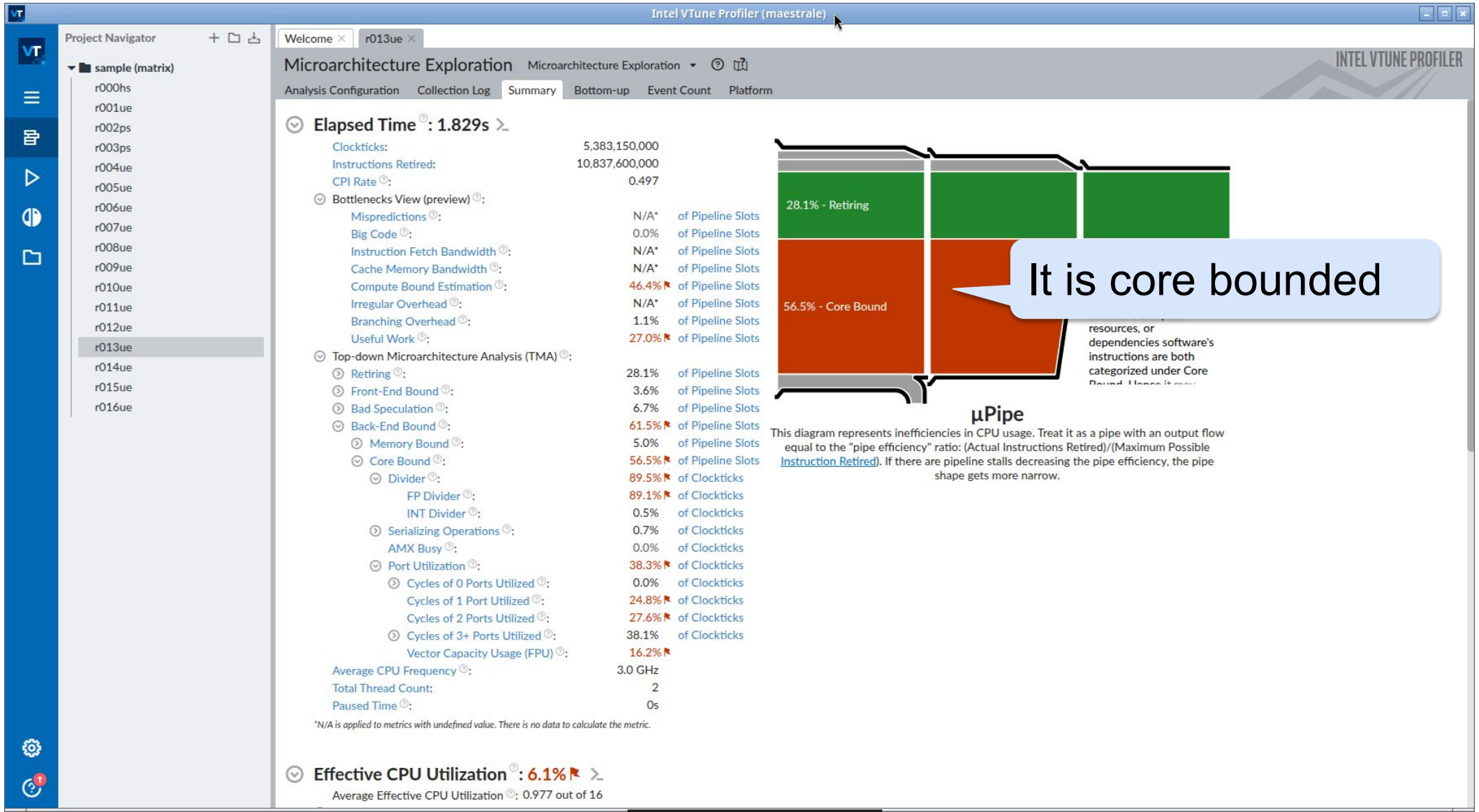
```
// Compute pairwise forces
for (int i = 0; i < Np; i++) {
        for (int j = i+1; j < Np; j++) {
      double dx = p[j].x - p[i].x;
double dy = p[j].y - p[i].y;
      double dz = p[j].z - p[i].z;
      […]
      double inv_r3 = 1.0/r2 *sqrt(r2);
      }
}
```

Now we check perform TMA on the nbody program

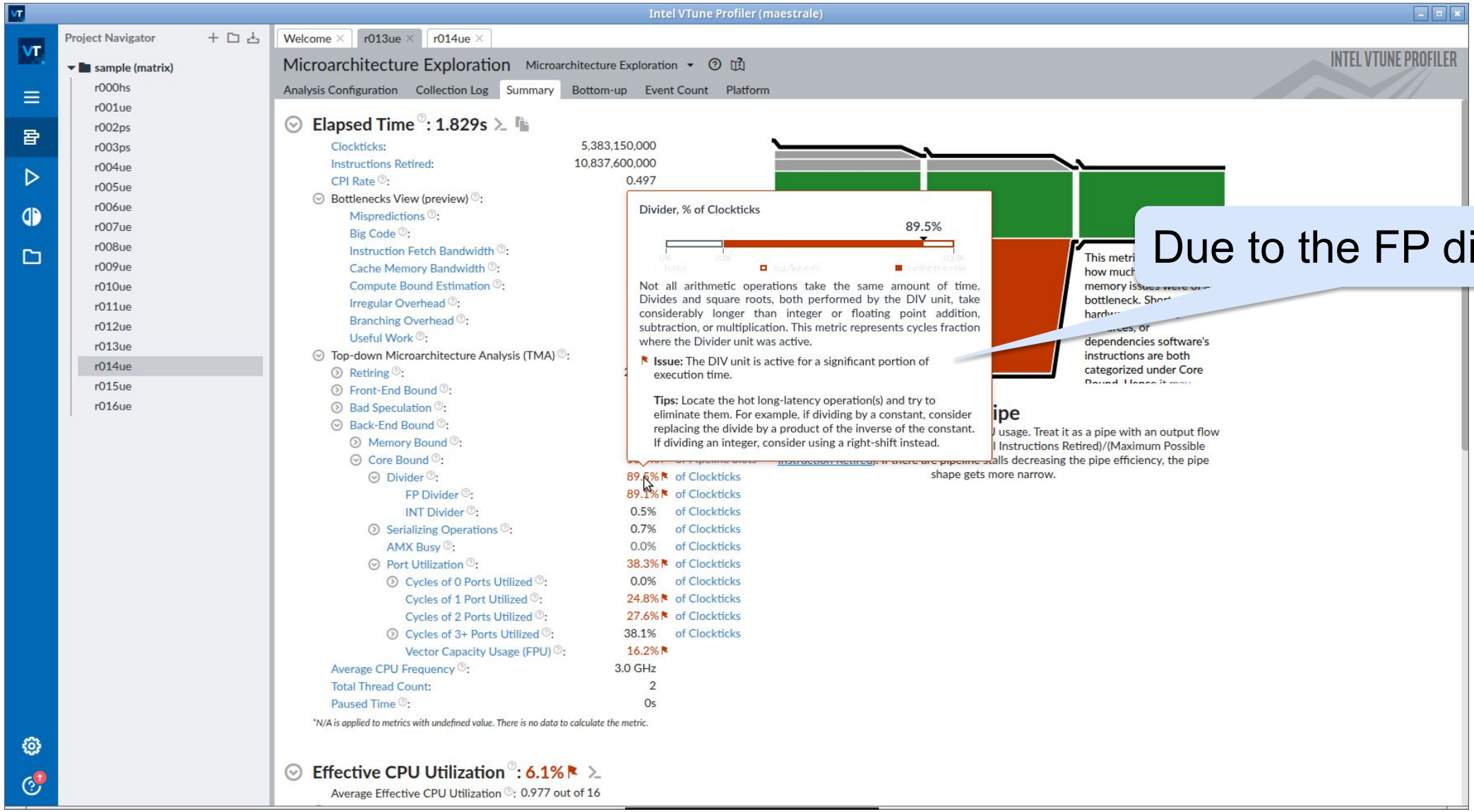
TMA on the nbody code



TMA on the nbody code



TMA on the nbody code



RSQRT — Reciprocal Square Root Approximation

- The `rsqrt` computes an **approximate value of 1 / sqrt(x)**. It has 14-bit precision

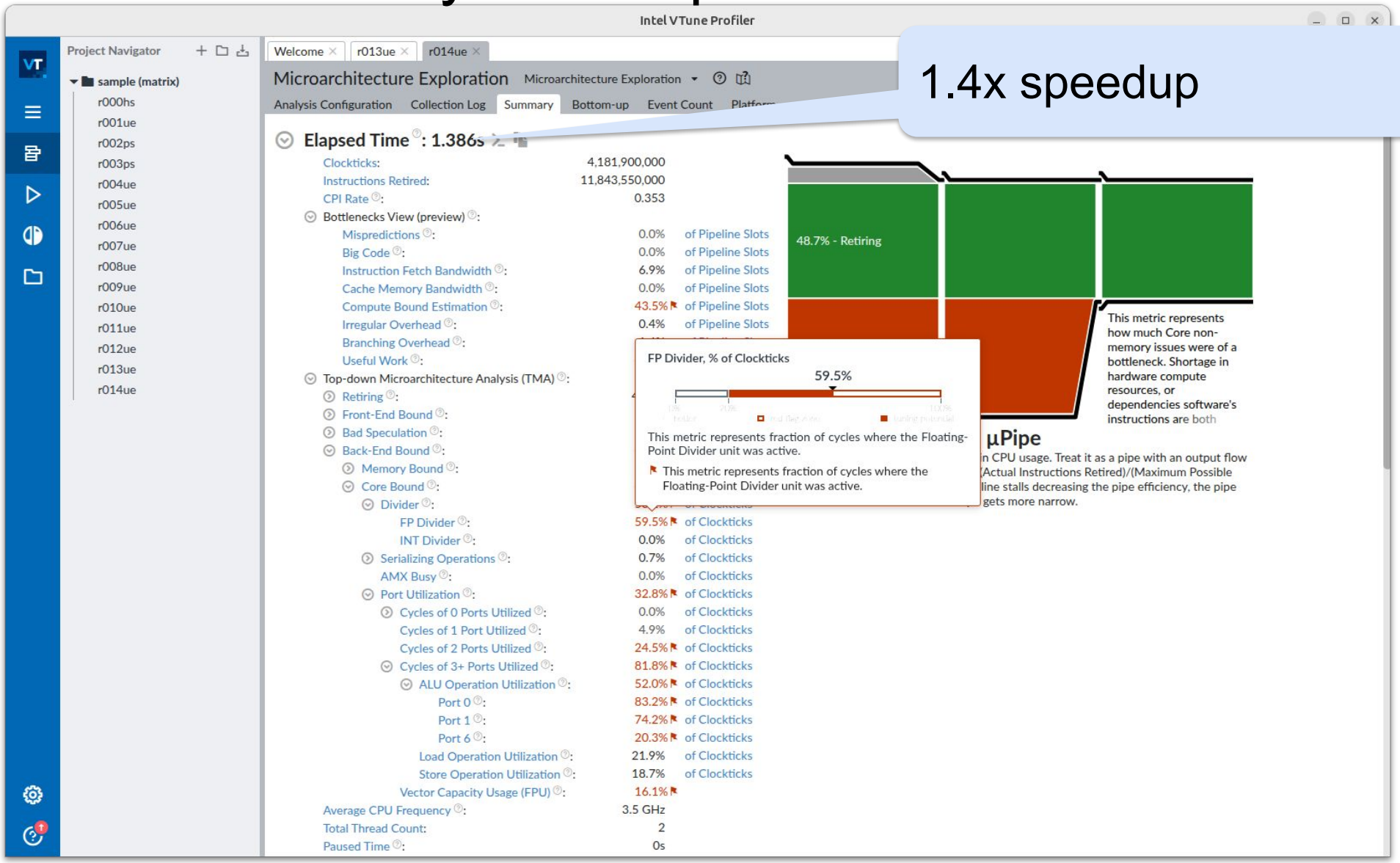
```
RSQRTSS xmm1, xmm2 ; xmm1[i] = 1 / sqrt(xmm2[i])  (approx)
```

- It exist a vectorized AVX-512 instruction

```
vrsqrt14ps zmm1, zmm2 ; zmm1[i] = 1 / sqrt(zmm2[i])  (approx)
```

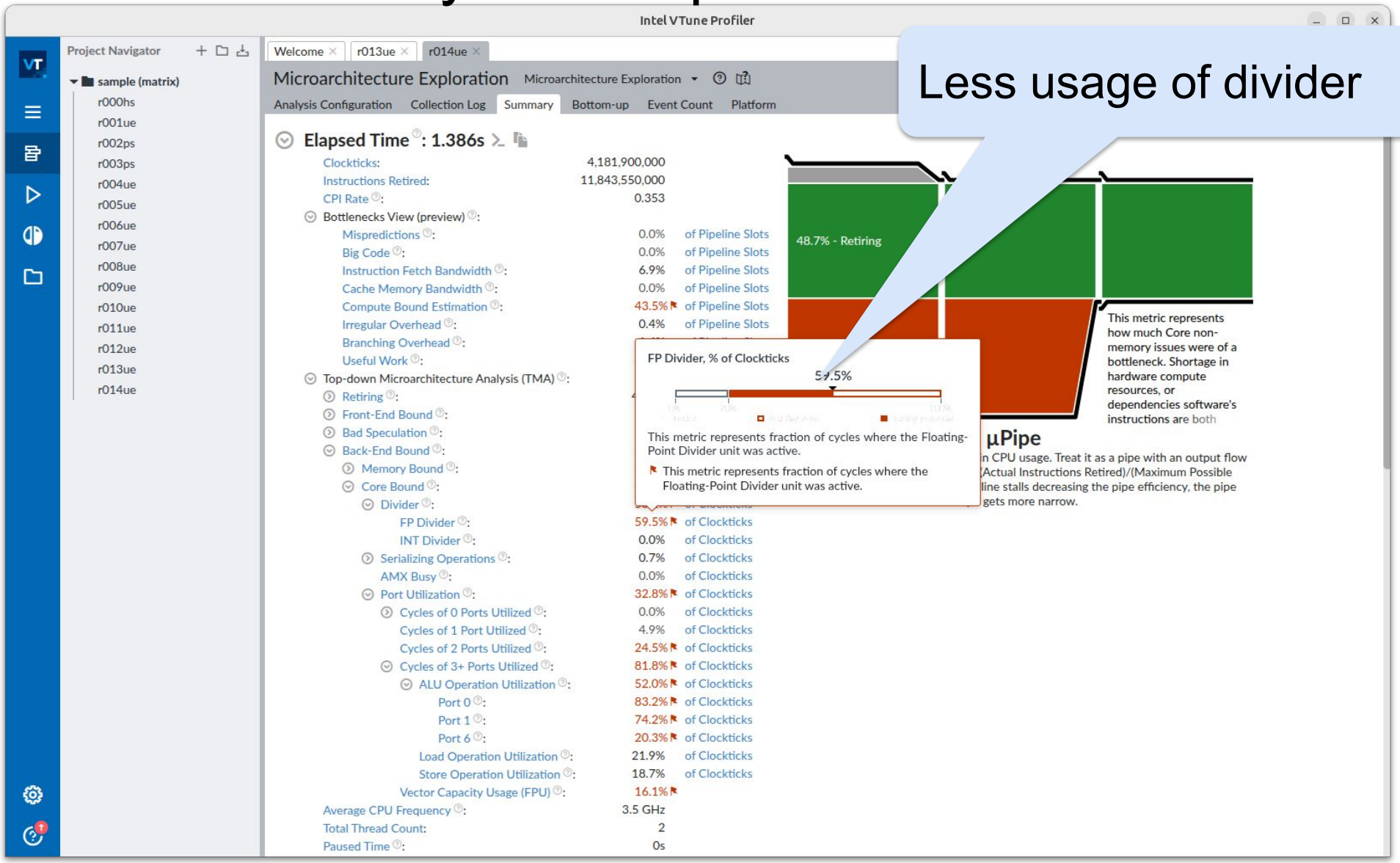
- Typically followed by a Newton–Raphson refinement for full precision if needed.

TMA on the nbody with rsqrt

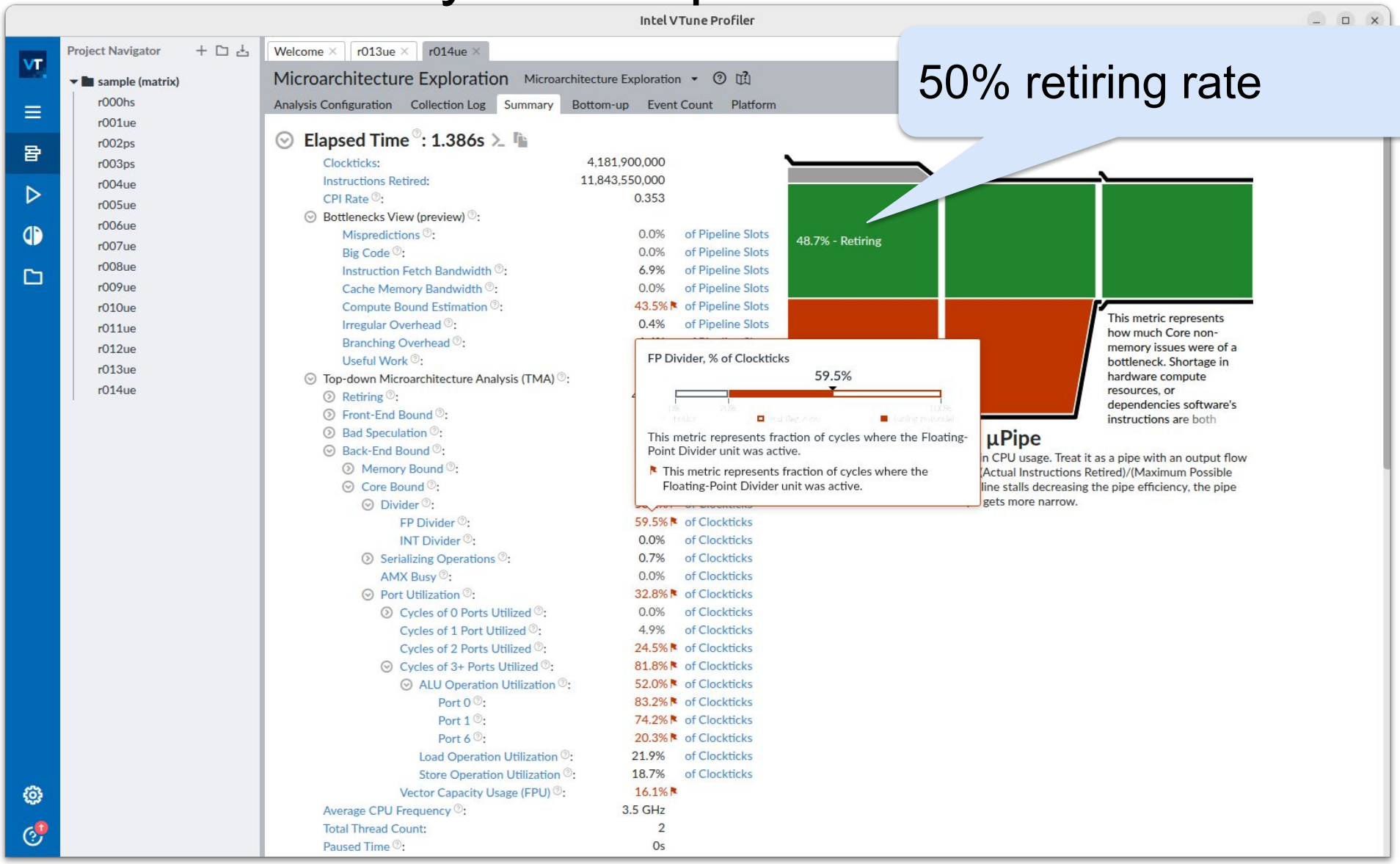


1.4x speedup

TMA on the nbody with rsqrt



TMA on the nbody with rsqrt



Bibliography

Denis, Bakhvalov, Performance Analysis And Tuning On Modern CPUs: Second Edition

Intel VTune Profiler Performance Analysis Cookbook