The how and why of profiling D code

Max Haughton

What is a profiler

Nullius in verba

- Produces a report about how the program spends its time for a given input. Although the dependence on the input is trivial, it is worth keeping in mind as performance is often dominated by patterns not inherent to the (machine) code that makes up the program.
- Information produced is typically fairly dumb. The profiler can't tell you to change algorithm, but can tell you how to make your algorithm faster.
- The time taken is not the only thing that can be profiled: Memory allocation may be more important, or in a multithreaded environment one may be solely interested in contention.

When to profile?

Nullius in verba

- The program is running slowly -> Profiling should hopefully reveal issues that can be resolved.
- There is an (informal) Pareto Principle involved: 20% of the work yields 80% of the speedup.
- Although profiling is typically (and mostly should be) reserved for diagnosing performance issues, it can also yield important understand of a program: A program may be bug free, but misunderstood in that the ratios of different work within the program may be different to the assumptions the program was designed around.
- Benchmarking and profiling aren't the same thing, but it's worth mentioning that doing the former regularly and the latter every now and again can do wonders in terms of keeping track of just how "fast" your code really is.

Profiler taxonomy

What kinds of profilers are available?

- After identifying what we want to measure (e.g. time or memory), how to do we go about data acquisition?
- Instrumentation: Add a hook to measure the data we're interested in, which is then stored and processed later. The naive approach (instrument everywhere), can lead to very complete but contextless data. And potentially a lot of overhead, YMMV.
- Sampling: Interrupt process, collect data, keep going, repeat. Much lower overhead.
- More feature complete sampling profilers (e.g. VTune) provide APIs for instrumentation and tracing. Other's follow a hybrid approach e.g. Tracy.
- Emulation: Run the program in an emulated environment, collect very fastidious but synthetic data. Valgrind's callgrind and cachegrind are famous examples.

Sampling what exactly

While better than nothing, -profile leaves much to be desired.

- We mentioned sampling before, but what will we sample.
- We need to measure the quantity we're interested in, obviously, but we also need to save where we got that data.
- Saving the instruction pointer is easy, but we need the full context so the callstack is superior.

Caveman profiling

Jesus' Blood Never Failed Me Yet

- Before looking at a *true* scotsman's profiler let's reject modernity and return to ~~monke~~ basics by thinking about how we could approximate a profiler with a humble debugger.
- Just use your debugger, get a few backtraces.
- A few samples and your brain can go a very long way, however the practical utility of this method is very limited. Data acquisition is annoying, data processing even more so.
- This method’s utility is much better on program with some notion of progress e.g. a simple % completion metric or even verbose output (so you know what is actually going on).
- If, however, the program is slow enough to be considered faulty in some way, i.e. blocked on some device, service, or library then this can be very useful.

Frame pointers

Something to keep in mind

- Sampling the call stack requires getting the instruction pointer, and being able to walk the call stack. First part is easy, second part not so much.
- On X86, omission of the frame pointer can let the compiler play with one more register at the expense of debugability.
- Debug info now means these frame pointers are not necessary, however a profiler might get this wrong (so worth keeping in mind)
- Always profile with debug symbols if possible.
- If needed, use "-gs" for dmd, "-fno-omit-frame-pointer" for GDC, or "--frame-pointer=all" for ldc

A simple instrumenting profiler

Profiling a contrived example using dmd's builtin profiler

```
int add(int x, int y)
{
  return x + y;
}
```

The compiler turns LHS into RHS (simplified)

```
int add(int x, int y)
{
  char[] loc = "add";
  trace_pro(loc);
  const res = x + y;
  _c_trace_epi();
  return res;
}
```

The pair of functions (prolog and epilog, to be clear) are in druntime, they collect timing information, which is then stored and printed upon program exit.

The data is outputted to a file called trace.log, or if this file already exists the new data will be merged. For this reason, *delete the log file on each run*.

See D & Digital Mars website for history of the feature.

Is that it then?

While better than nothing, -profile leaves much to be desired.

- Only instrumented functions are seen in the profile. This is potentially catastrophic for some programs, e.g. IO bound workloads, calls into non-root module functions etc
- The feature makes a valiant attempt to sample the call graph, but not the call stack. This is not ideal - more on that next.
- Data is only collected at the function-level.
- Potentially very high overhead, especially if a function.
- We can do better.

Profiling allocations with dmd

Don't fear the reaper - how to easily profile GC allocations

- Instrumentation is not great for profiling time, but for profiling allocations it's very useful.
- Overhead? Recorded data unaffected, allocation is slow anyway so time not an issue.
- Compile with "-profile=gc"
- Heap profiling does not have to be integrated with the language, but it's helpful to know the exact type of an allocation.

GC profiling example

Dirty deeds done dirt cheap (But still really useful)

1. Compile with `-profile=gc`

2. Run program

3. Inspect the log (located at `profilegc.log`)

bytes allocated, allocations, type, function, file:line
18400 50 ubyte[] D main alloc.d:11 1600 50 alloc.Data D main alloc.d:12

Really useful? Allocations can (and usually are) very slow - a good malloc implementation on the hot path might still be hundreds of instructions (note: instructions, this measure doesn't even take the cache into account!)

```
class Data {
    int x;
}

void main()
{
    foreach(int elem; 0..50)
    {
        18400 bytes allocated / 50 allocations
        auto buf = new ubyte[](256);
        1600 bytes allocated / 50 allocations
        auto encapsulated = new Data;
        /* Do work */
    }
}
```

Visual Studio Code integration! (Say thanks to WebFreak)

Beyond contrived examples.

Towards "real" code

- Basics first: You can't really optimize add, and it was probably inlined anyway.
- Programming practice can lead to a relatively obscure mapping of name to task. OOP: horses vs. chickens.
- Let's look at a profile of dmd compiling hello world.

An informative iota

A small profile of non-trivial program

```
- int dmd.mars.tryMain(ulong, const(char)**, ref dmd.globals.Param)
   + 26.75% Module::importAll
   - 9.10% DsymbolSemanticVisitor::visit
      DsymbolSemanticVisitor::visit
      - DsymbolSemanticVisitor::visit
      - 3.55% DsymbolSemanticVisitor::visit
         - 2.10% DsymbolSemanticVisitor::funcDeclarationSemantic
            typeSemantic
            dmd.mtype.Type dmd.typesem.typeSemantic(dmd.mtype.Type, ref const(dmd.globals.Loc), dmd.dscope.Scope*).visitFunction(dmd.mtype.TypeFunction)
            + dmd.mtype.Type dmd.typesem.merge(dmd.mtype.Type)
```

- Collected using call stack sampling.
- Due to a quirk of c++ demangling and the visitor pattern, this information is basically lost without CSS.
- Alternative is just a list of functions that appeared in samples, which isn't great.

So, what are we looking for?

We know what we want, we don’t know (yet) how to get it.

- Low overhead - zero is impossible, but we can get close.
- Call stack sampling is a must-have.
- Source level profiling is very nice to have (but requires debug info)
- Full complement of information from the hardware (more on that later).
- Cross-platform?

Profiling with perf

The Second Best Secret Agent In The Whole Wide World

- The Linux Kernel exposes a subsystem called perf_event to read performance counters in a mostly platform agnostic way, perf is the canonical frontend to it.
- `perf list` reveals more than just measuring time. A long list of hardware and software events are presented. Page faults, for example, are a very handy thing to keep an eye on.
- perf has a lot of functionality out of the box, also serves as the basis for several other tools (profilers, optimizers etc.). A jack of all trades, master of some.
- Although the tool aims to be platform agnostic (and for the first 80% of performance problems it is), some architectures are more equal than others.
- perf is also part of (and can act as a frontend for) a rich set of tracing utilities covering both userspace and the kernel itself. These are a talk all by themselves (more for profiling *systems* than code), so I won't cover them. See Brendan Gregg's excellent website to learn more about it than you'll ever get to use in anger unless you work for Netflix.

Perf workflow

A basic recipe for using perf

1. Start with "`perf record -g`" to collect data and sample callstacks - use "`-e ...`" to enable specific performance counters (as elaborated on later)

2. "`perf report`" will open a fairly nice TUI for you to navigate the data collected

3. "`perf annotate`" annotates the assembly and source code (albeit not very ichthyomorphically) with collected data.

GUI profilers (and windows...)

The Second Best Secret Agent In The Whole Wide World

- perf is very good at getting data onto your screen, however the interface is not the best for exploring the data.
- There is a pretty good GUI for perf called hotspot (See slide about flame graphs), but doesn't quite compare to the following tools.
- CPU designers provide tooling for getting the most out of their processors when profiling: Intel has vTune (amongst many other tools), AMD has uProf.
- Perf's source annotation tool is functional and useful but a bit 1980s. The aforementioned tools are quite a bit better.

Flame Graph

I can see clearly now

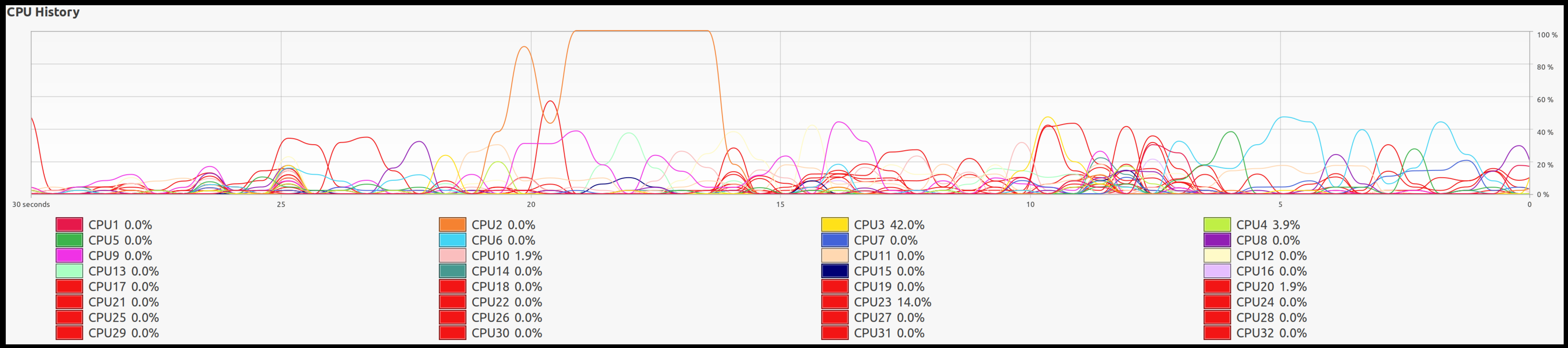
- A really handy way of looking at a profile's callstack data without going insane. A flame graph is a big stack of boxes: the x-axis indicates frequency, the y-axis is stack depth.
- The x-axis is ordered alphabetically, NOT by time. This is so identical frames can be merged.
- Simple code will likely have a very simple flame graph, the utility of the technique comes in larger projects.
- How to generate one? Available from the Profilers Hotspot, vTune, uProf, a few others. To generate one standalone, Brendan Gregg has a popular tool.

Flame Graph

Function HotSpots
Metrics
Flame Graph
Call Graph
Counters:
CYCLES_NOT_IN_HALT
Process IDs:
[3168393] dmd
Zoom Entire Graph
Search function name...
Clear
Click on any block in Flame Graph to focus on it's children.
_D3dmd4mars7tryMainFmPPxaKSQz7globals5ParamZi
_Dmain
_d_run_main2
_d_run_main
main
__libc_start_main
[ROOT]

Reliable data

You can tame the chaos



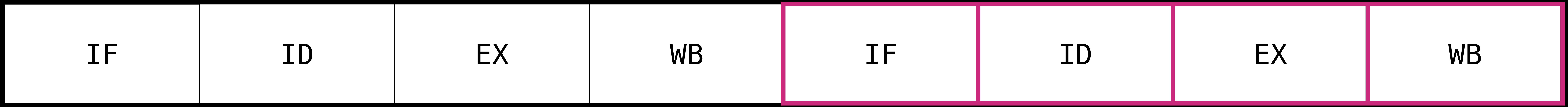
Random snapshot of CPU Utilization while writing this slide

- If you are going to go to war based on a profile, you may want to make sure your data is consistent.
- This is mostly something to keep in mind when benchmarking rather than profiling, but the techniques can nonetheless be useful.
- Some metrics are reliable (i.e. an instruction is always an instruction regardless of how long it takes), but other measurements can be dependant on transient (extrinsic) properties of a particular run.
- Power management (laptops *especially)* is something processor companies beat each other to death over, so the processor will often be extremely aggressively turned down / turned-off to save power).
- Example:If the CPU Freq is turned down relative to the (say) speed of memory, memory latency now looks better than previously.
- Should you bother? Depends on who owns the computers your code is going to run on.

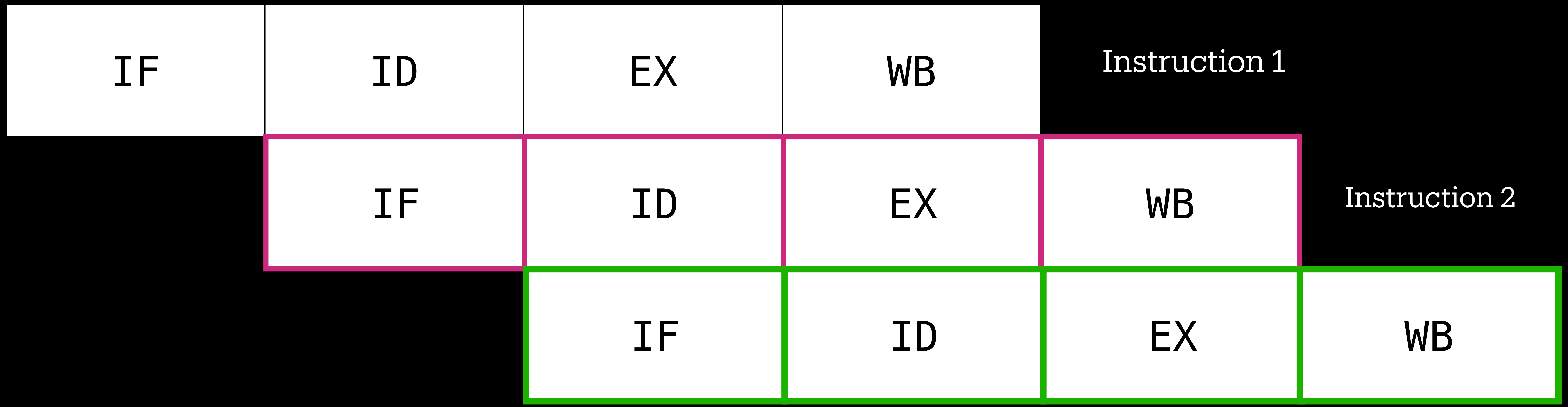
A little microarchitecture

Nowhere near enough time to go into detail, but enough time to build intuition (hopefully)

Stages of an instruction's execution: Fetch, Decode, Execute, Writeback (ignore memory for now)



Can we overlap them? Yes, in many cases we can overlap their executions



Instruction 1

Instruction 2

Instruction 3

We have just decreased our CPI from 4 cycles per instructions to 1!

When can't we? Hazards. If we have a write-after-read dependency, then we will have to induce a stall - i.e. wait for the result of instruction 1 so instruction 2 can use it.

A little microarchitecture

Nowhere near enough time to go into detail, but enough time to build intuition (hopefully)

- We have CPI >= 1, i.e. a *scalar* processor.
- If we have CPI < 1, then we have a *superscalar* processor. A modern processor is *very* superscalar.
- At the expense of complexity and power usage, we can have a processor be out-of-order: The processor can do independent work independently (ideally in parallel by using a superscalar backend).
- Speculation: In a modern OOO superscalar processor, speculation (doing work based on a guess rather than a guarantee) is the default state of being. Branch Prediction is very successful, making deep speculation possible.
- See Tomasulo's algorithm for how this actually works, in any computer architecture book (all 2 of them)
- To learn how these are techniques are actually implemented, Agner Fog produces a detailed monograph on processor architecture.

The memory hierarchy

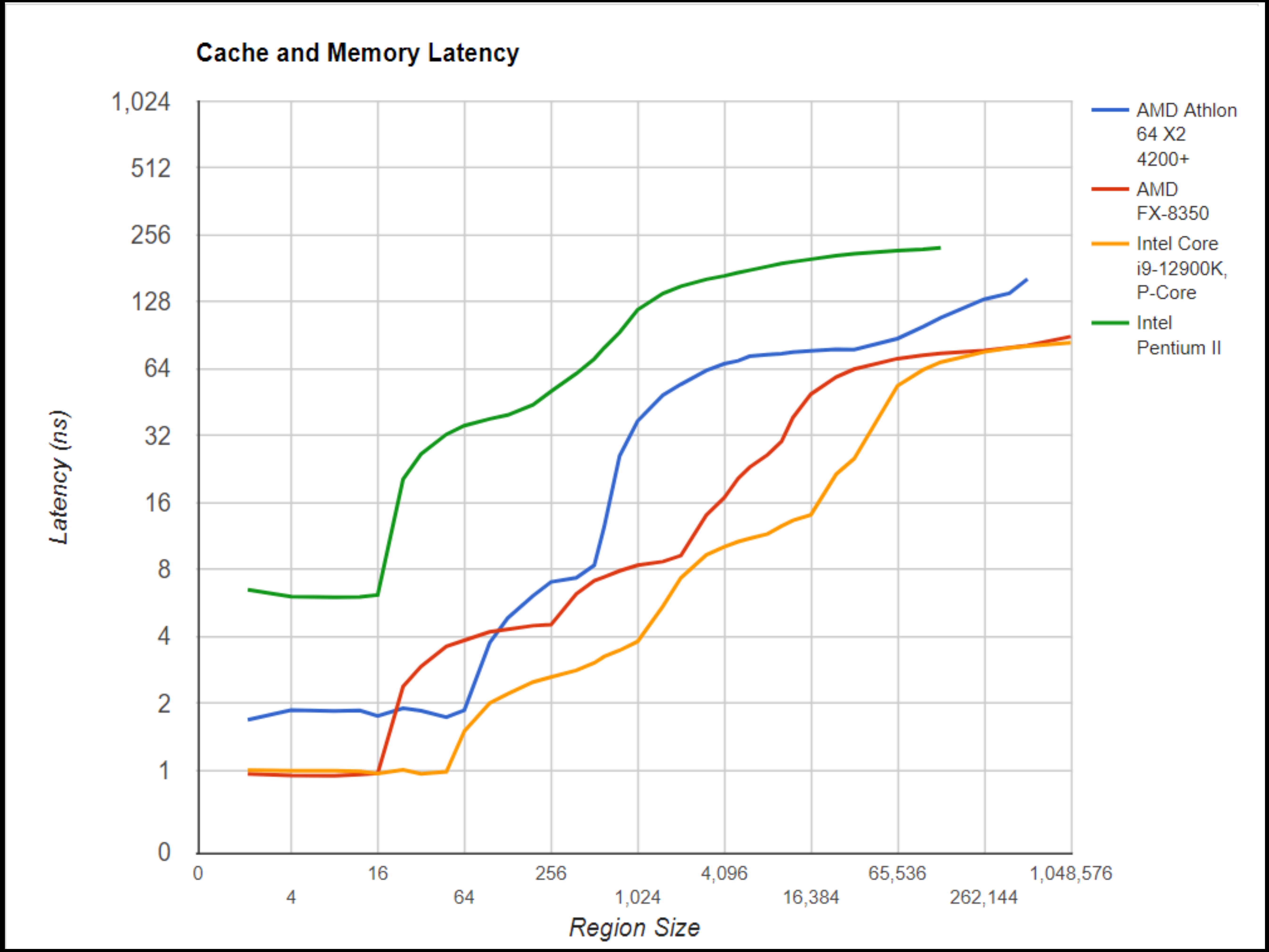
It's the memory stupid!

- Processors have become much faster than their memory.
- The techniques mentioned previously allow the processor to alleviate some of that, e.g. by doing other work while waiting for memory.
- Despite ever-increasing amounts of memory allocated to programs, memory access remains predictable and local - spatial locality, temporal locality.
- The chip designer has a choice between big and slow, or small and fast. Rather than choosing one, your processor has a memory *hierarchy* - multiple levels of tradeoffs between latency, bandwidth, and size (and power usage).
- You can do some serious work in the time taken by missing a level of this memory hierarchy, so memory is practically the number one thing to keep an eye on.

Historical memory latencies

A classic latency test

- Random selection of old and new processors
- 12900K is the brand new Intel chip.
- Not much has changed. Things have been getting faster, of course, but not quite
- Notice the straight line, then a bump then a (slightly mangled) straight line, these are the gradations between different levels of the (data) caches.



chipsandcheese.com

There's always leaks - Spectre and Meltdown

Famous proof that speculation is not all good.

- Speculation: Great when it's right, what happens when the processor guessed wrong?
- Processor guesses wrong, bails out, end of story? Not quite.
- If we can find a side channel and make the CPU touch it in a speculative/transient operation (nomenclature varies in lit.), we can extract sensitive information.
- We can! Trick the processor into doing an operation speculatively, use that result as an index into an array.
- Time the accesses to this array, do some basic arithmetic, you know which one was transiently accessed, that's the result of the work you made the processor do.
- You can now access *any* virtual *memory*. Meltdown (same rough idea) let's you access any *physical* memory!

Performance counters

Smarter profiling.

- As the processor goes about running your code, it keeps track of the statistics of execution types, stalls, etc.
- Using the techniques mentioned previously, we can relate code to how long it took and why it took so long.
- On Linux, using these is easy (you may need to set your perf_event_paranoid setting) - just use perf with the "-e" flag as mentioned previously.
- On Windows, you need a profiler like vTune or uProf and a special driver which will be installed with that profiler.
- Example: Port utilization - the CPU dispatches work to execution ports, you can use performance counters to track the frequencies of how many your code was able to utilize (a low number is an indication you can't pull in data fast enough).

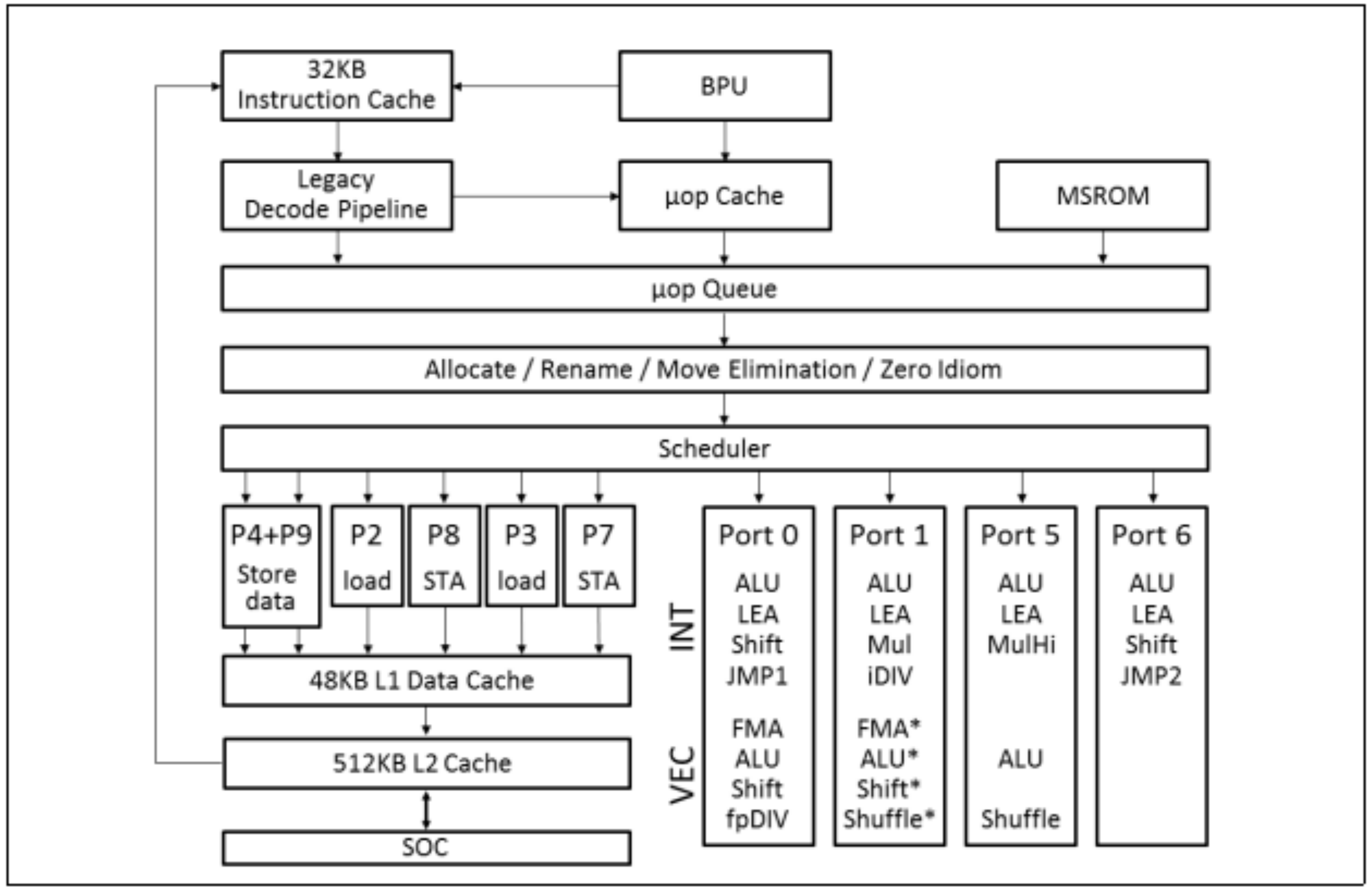


Figure 2-1. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

vTune

Smarter profiling.

- Thanks to TMAM (Top-down Microarchitecture Analysis Method) we can synthesize all these counters into a cohesive view like shown on the RHS

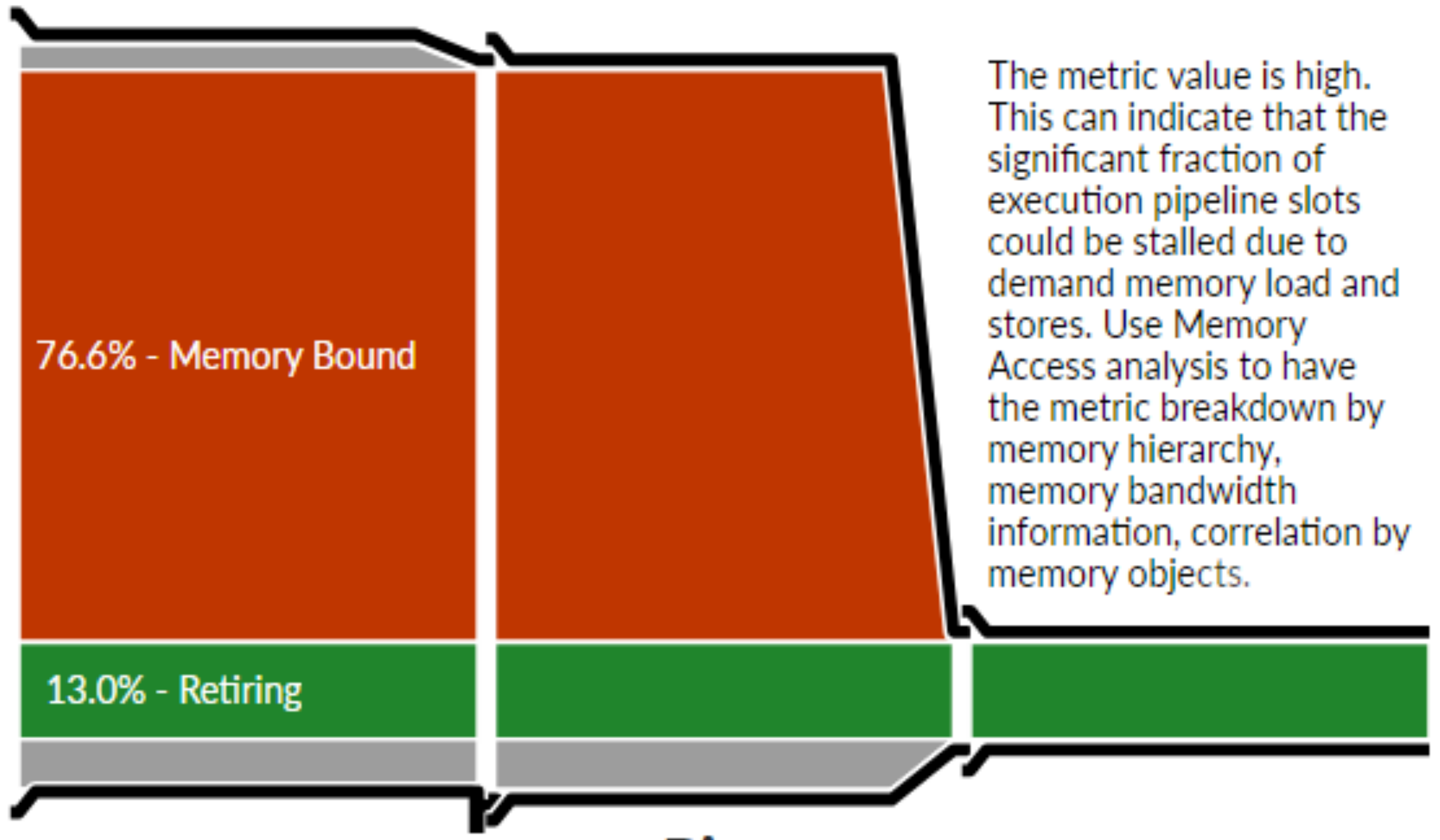
- This is an example of memory bound code.

- Only vTune is able to do this well.

- Note that 65% of memory accesses are missing the cache entirely, so we have 0-ports exercised most of the time.

Elapsed Time: 43.609s

Clockticks:	1,209,171,600,000	
Instructions Retired:	321,081,600,000	
CPI Rate:	3.766	
MUX Reliability:	1.000	
Retiring:	13.0%	of Pipeline Slots
Front-End Bound:	3.4%	of Pipeline Slots
Bad Speculation:	0.3%	of Pipeline Slots
Back-End Bound:	83.3%	of Pipeline Slots
Memory Bound:	76.6%	of Pipeline Slots
L1 Bound:	4.7%	of Clockticks
L2 Bound:	1.0%	of Clockticks
L3 Bound:	13.1%	of Clockticks
Contested Accesses:	0.3%	of Clockticks
Data Sharing:	1.6%	of Clockticks
L3 Latency:	8.1%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	65.3%	of Clockticks
Memory Bandwidth:	32.0%	of Clockticks
Memory Latency:	58.7%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	6.7%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	7.3%	of Clockticks
Cycles of 0 Ports Utilized:	71.4%	of Clockticks
Cycles of 1 Port Utilized:	11.2%	of Clockticks
Cycles of 2 Ports Utilized:	8.2%	of Clockticks
Cycles of 3+ Ports Utilized:	9.0%	of Clockticks
Vector Capacity Usage (FPU):	25.0%	
Average CPU Frequency:	4.4 GHz	
Total Thread Count:	11	
Paused Time:	0s	

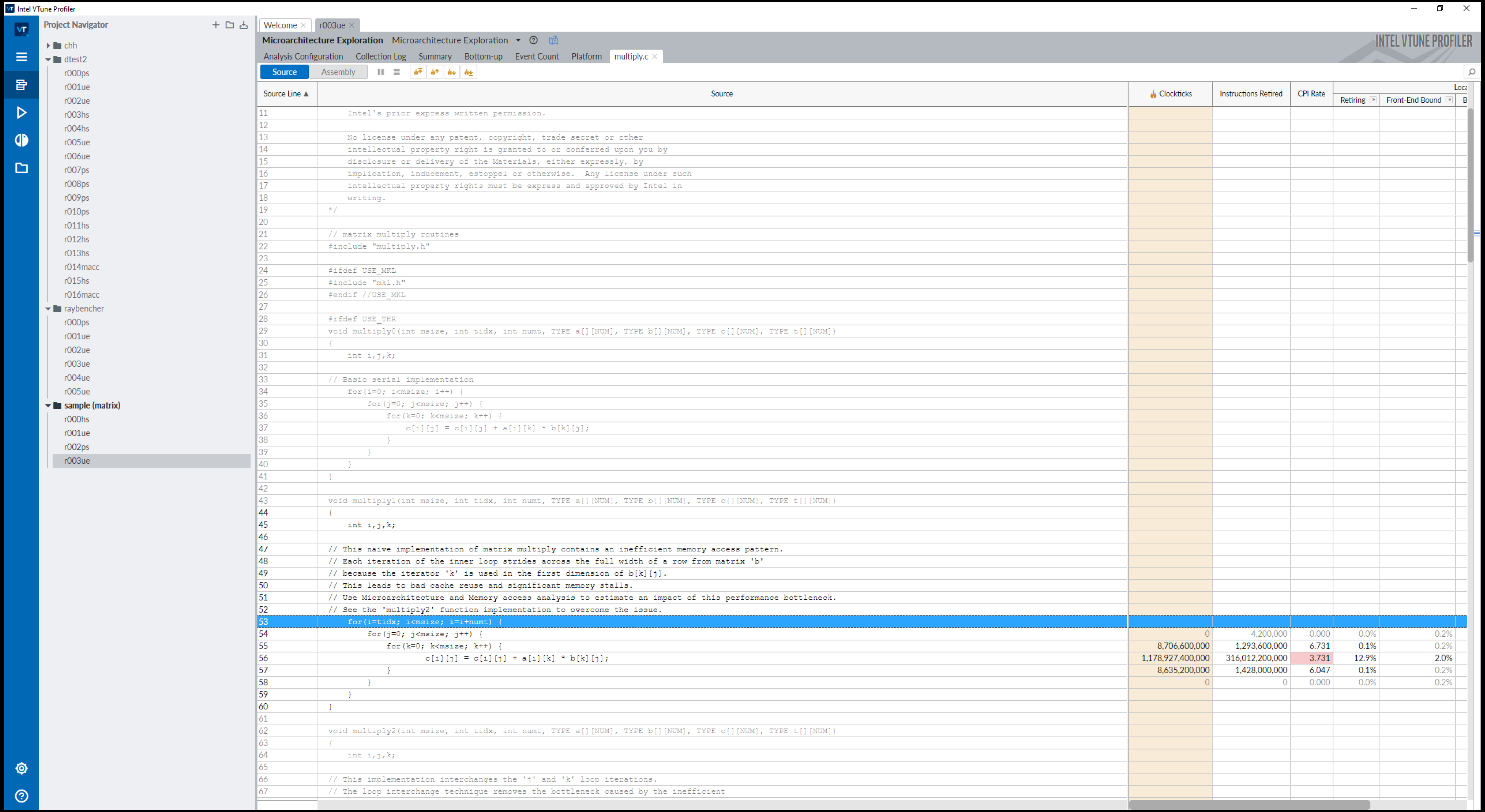


μPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

vTune

Pause the video and take a look



This example is C, works absolutely fine with D bar demangling.

vTune

Threads

vTune

Latency

Coz - A causal profiler

A very different way of doing things

- Previously we emphasized sampling profilers as the way to go.
- Coz is a bit different. It performs performance experiments. We try to measure how much a given line contributes to the speed of a program by slowing the rest of the program down.
- Mainly intended for profiling multithreaded code.

Tracy - A frame profiler

- Tracy is a profiler intended for profiling video games
- It's linked with the program being profiled, and is activated on a per-frame basis
- Data is collected externally via a socket
- Cross-platform.
- Rapidly gaining features.
- Game development exposes it to new ideas in concurrency and parallelism

The end

- Questions?