

# Stochastic instrumentation

# Stochastic instrumentation

# Previous limitations

- Static instrumentation is expensive (and affects accuracy)
- With performance counters:
  - How could we find **hot spots**?  
(small groups of instructions that the application spends a lot of time running)
  - What about performance counts (cache misses, mispredicted branches,...)  
at those **hot spots**?

# Solution

## Stochastic instrumentation:

- every N cycles (e.g. every 1,000,000th cycle / every 0.1ms), a **sample** is taken
- the **sample** records:
  - which instruction is currently being executed
  - optionally, what it is waiting for (instr. decoding, pipeline bubble, memory access, ...)
  - optionally, instruction addresses of the last few branches
  - optionally, whether those branches were successfully predicted

# Stochastic instrumentation

- Pros
  - no performance penalty
  - no interference with normal execution
  - accuracy naturally increases on hotspots
- Cons
  - like performance counters, needs hardware support

# Analysis applications

- Linux
  - Linux **perf**: perf record / perf report
  - KDAB hotspot
- MacOS: **Apple XCode Instruments**
- Windows: **Visual Studio** (“dynamic instrumentation” / “collection via sampling”)
- Intel-specific: **vTune**
- AMD-specific: **uProf**

# Bottom-up analysis

**Microarchitecture Exploration** | Microarchitecture Exploration | Bottom-up

Analysis Configuration | Collection Log | Summary | **Bottom-up** | Event Count | Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound
p_1gs_Ax_1	223.864ms	300,000,000	1,198,800,000	0.250	32.5%	22.9%
p_2gs_Ax_2	214.773ms	270,000,000	2,712,000,000	0.100	76.8%	6.5%
p_spx_dual_violated	193.182ms	254,400,000	1,929,600,000	0.132	43.5%	12.6%
func@0x92e0	87.500ms	264,000,000	261,600,000	1.009	41.3%	4.1%
p_spx_dse_update_w	84.659ms	112,800,000	1,429,200,000	0.079	73.2%	1.0%
p_1gs_Ax_2	84.091ms	108,000,000	906,000,000	0.119	89.9%	10.9%
p_bfrt_el	81.818ms	112,800,000	528,000,000	0.214	24.1%	12.1%
p_vec2_fmsub	80.682ms	134,400,000	1,194,000,000	0.113	31.2%	7.4%
p_spx_dual_full_leaving	70.455ms	133,200,000	100,800,000	1.321	37.2%	8.7%
func@0xa3a0	51.705ms	168,000,000	362,400,000	0.464	76.6%	21.9%
p_spx_compute_row_sib_dai	51.705ms	81,600,000	298,800,000	0.273	23.3%	7.8%
p_bfrt_select	49.432ms	61,200,000	408,000,000	0.150	48.6%	11.3%
p_map_qlookup	42.045ms	46,800,000	61,200,000	0.765	8.3%	1.4%
p_spx_dse_update_xB	40.909ms	57,600,000	457,200,000	0.126	45.4%	8.1%
p_fold_iter	33.523ms	0	162,000,000	0.000	15.1%	1.9%
p_vec_z_scatter	29.545ms	50,400,000	228,000,000	0.221	31.1%	3.9%

**Microarchitecture Usage: 32.5% of Pipeline Slots**

**μPipe**

- Retiring: 32.5% of Pipeline Slots
- Front-End Bound: 22.9% of Pipeline Slots
- Front-End Latency: 14.6% of Pipeline Slots
- ICache Misses: 0.0% of Clockticks
- ITLB Overhead: 0.0% of Clockticks

Thread Timeline:

- lpopt (TID: 369669)
- gzip (TID: 369688)
- amplx-runss (TID: 369669)
- lpopt (TID: 369688)

CPU Time

FILTER 100.0% | Any Process | Thread Any Thread | Module Any Module | Call Stack Mode User functions + | Loop Mode Functions only | Inline Mode Show inline funt

# Flame graphs





# Tutorial

# Matrix multiplication

We want to implement a fast matrix multiply code for medium-sized matrices (e.g.  $1024 \times 1024$ ).

Download [matmul\\_0.c](#) and implement the code of the function `matrix_multiply()`.

Then, find performance issues and, if possible, improve the implementation.

```
#define SIZE    1024

...

void matrix_multiply(float *x, const float *a, const float *b)
{
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            x[i * SIZE + j] = ...;
        }
    }
}
```

# 1024x1024 fp32

version	description	seconds	ratio	improvement
matmul_0	placeholder code, sets X to 0	0.008		
matmul_1	straightforward implementation	3.565	<b>1x</b>	
matmul_2	transpose B matrix	0.585	6x	6x
matmul_3	block multiply	0.170	21x	3x
matmul_4	AVX512	0.034	105x	5x
matmul_5	OpenBLAS	0.029	122x	1.2x
matmul_6	OpenBLAS, 24 threads	0.023	155x	1.3x

# 8192x8192 fp32

version	description	seconds	ratio	improvement
matmul_0	placeholder code, sets X to 0	0.400		
matmul_1	straightforward implementation	2794.068	<b>1x</b>	
matmul_2	transpose B matrix	338.163	8x	8x
matmul_3	block multiply	79.346	35x	5x
matmul_4	AVX512	14.121	198x	6x
matmul_5	OpenBLAS	7.462	374x	2x
matmul_6	OpenBLAS, 24 threads	1.114	2508x	7x

# CPU vs. GPU

32768x32768 (multiplication only)

version	description	seconds	ratio
matmul_6	OpenBLAS, 24 threads, 7900x3d	44.365	1x
matmul_7	cuBLAS, nVidia H100	4.498	10x

65536x65536 (multiplication only)

version	description	seconds	ratio
matmul_6	OpenBLAS, 24 threads, 7900x3d	344.961	1x
matmul_7	cuBLAS, nVidia H100	28.657	12x

Approx. total speedup

naive CPU impl. → GPU impl.: 30000x

# Byte stream filtering

We read (from standard input) a stream of bytes as unsigned 8-bit integers.

We want to filter those integers, and write (to standard output) only some of them. Specifically, we write those who are divisors of 873248763249102240.

Download [filter\\_0.c](#), and implement the code of the function `filter()`.

Then, find performance issues and, if possible, improve the implementation.

```
size_t filter(unsigned char *out, unsigned char *in, size_t n)
{
    size_t s = 0;

    for (size_t j = 0; j < n; j++) {
        unsigned char c = in[j];

        // TODO: keep only divisors of 873248763249102240

        out[s] = c;
        s++;
    }

    return s;
}
```



