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

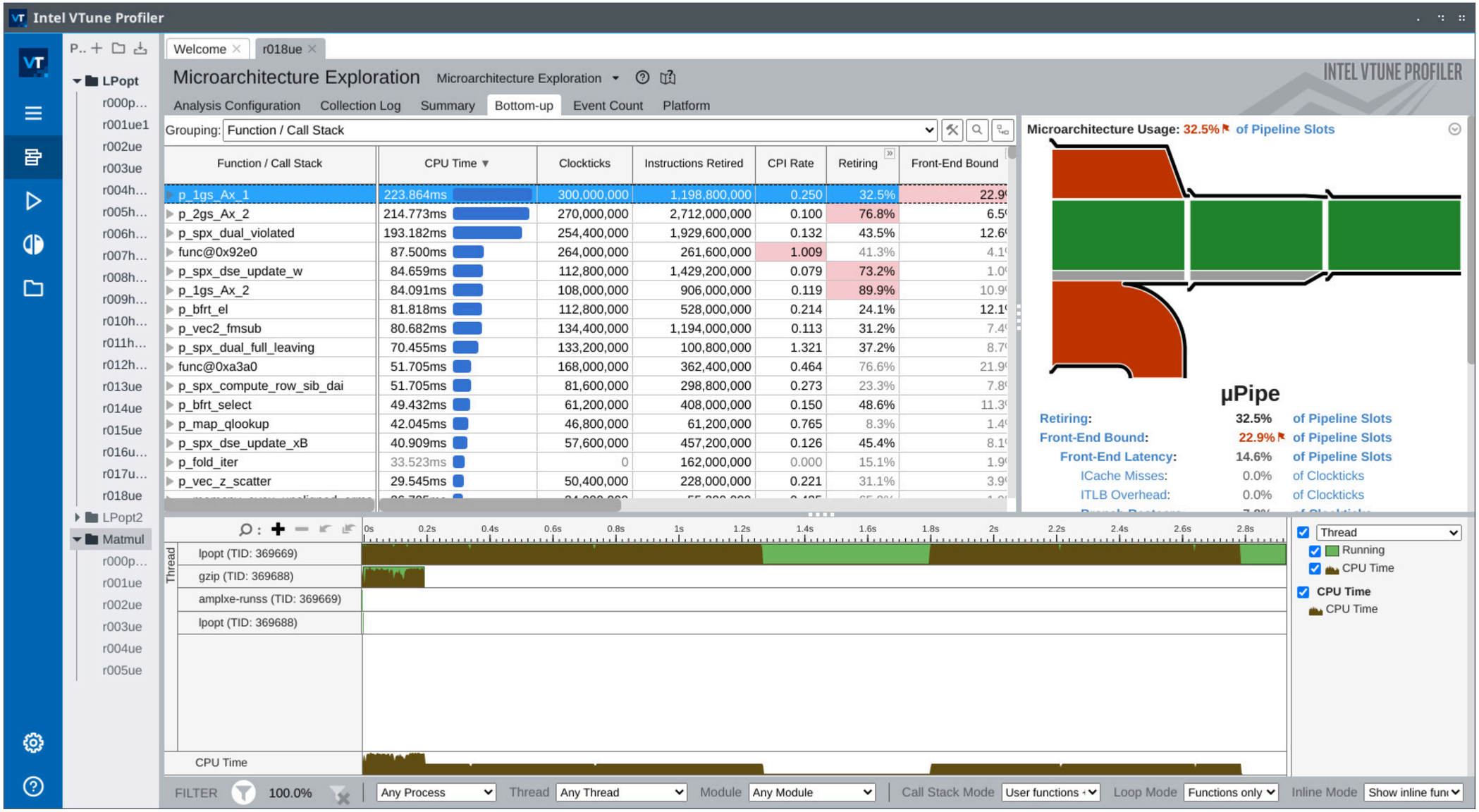
- Pros
  - no performance penalty
  - no interference with normal execution
  - accuracy naturally increases on hotspots

- 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



Flame graphs

Tutorial

Matrix multiplication

We want to implement a fast matrix multiply code for medium-sized matrices (e.g. 1024 × 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	**1x**	
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	**1x**	
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 87324876324910240.

Download `filter_0.c`, and implement the code of the function `filter()`.

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

```c
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;
}
```