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

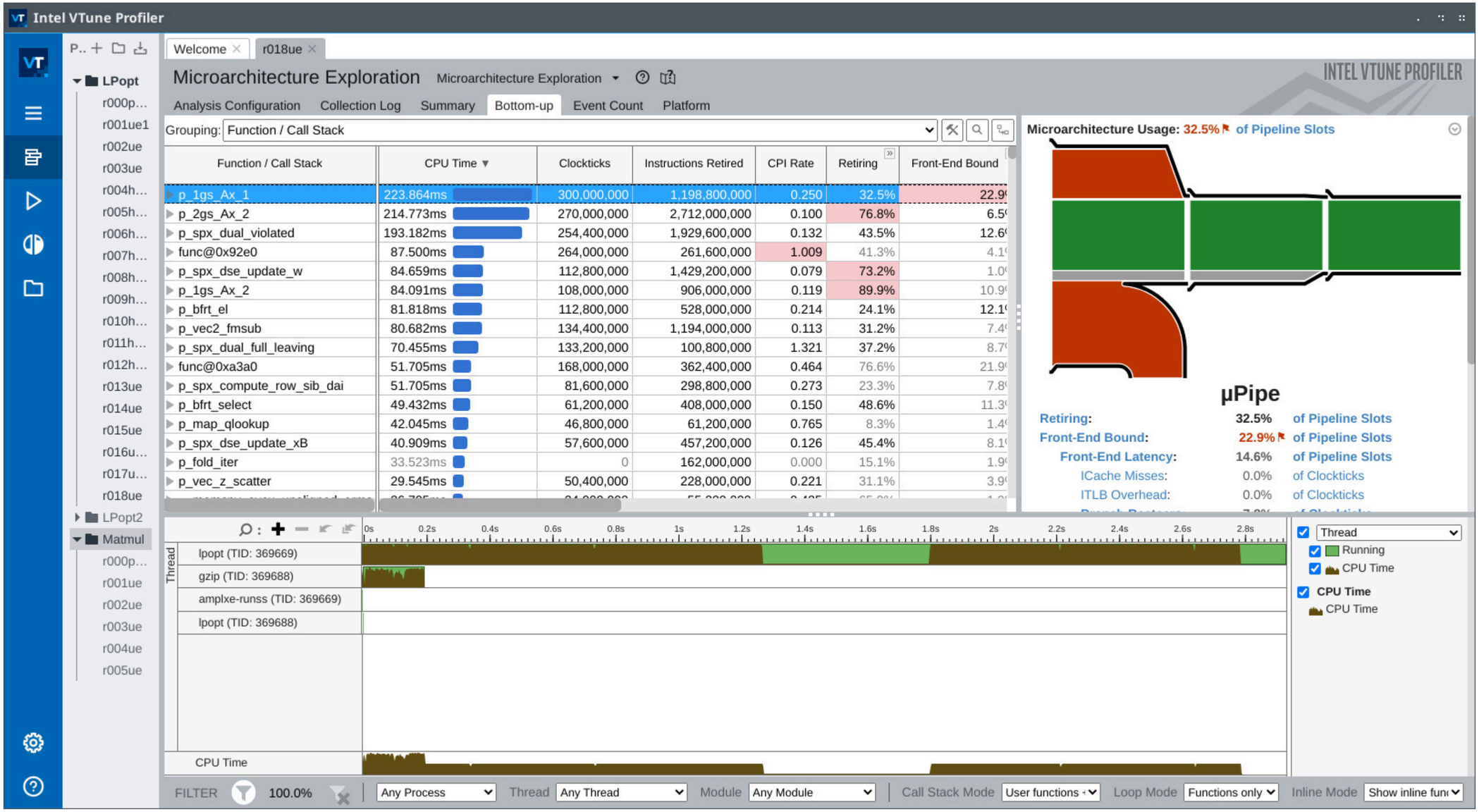
- Pros
  - no performance penalty
  - no interference with normal execution
  - accuracy naturally increases on hotspots

- Cons
  - like performance counters, needs hardware support

Analysis applications

- Linux
  - `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.

```c
#define SIZE    1024

...

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

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;
}
```