

uarch-bench

A collection of low level, fine-grained benchmarks intended to investigate micro-architectural details of a target CPU, or to precisely benchmark small functions in a repeatable manner.

Disclaimer

This project is very much a work-in-progress, and is currently in a very early state with limited documentation and testing. Pull requests and issues welcome.

Purpose

The uarch-bench project is a collection of micro-benchmarks that try to stress certain micro-architectural features of modern CPUs and a framework for writing such benchmarks. Using libpfpc you can accurately track the value of Intel performance counters across the benchmarked region - often with precision of a single cycle.

At the moment it supports only x86, using mostly assembly and a few C++ benchmarks. In the future, I'd like to have more C or C++ benchmarks, allowing coverage (in principle) of more platforms (non-x86 assembly level benchmarks are also welcome). Of course, for any non-asm benchmark, it is possible that the compiler makes a transformation that invalidates the intent of the benchmark. You could detect this as a large difference between the C/C++ and assembly scores.

Of course, these have all the pitfalls of any microbenchmark and are not really intended to be a simple measure of the overall performance of any CPU architecture. Rather they are mostly useful to:

1. Suss out changes between architectures. Often there are changes to particular micro-architectural feature that can be exposed via benchmarks of specific features. For example, you might be able to understand something about the behavior of the store buffer based on tests that exercise store-to-load forwarding.
2. Understand low-level performance of various approaches to guide implementation of highly-tuned algorithms. For the vast majority of typical development tasks, the very low level information provided by these benches is essentially useless in providing any guidance about performance. For some very specific tasks, such as highly-tuned C or C++ methods or hand-written assembly, it might be useful to characterize the performance of, for example, the relative costs of aligned and unaligned accesses, or whatever.

-
3. Satisfy curiosity for those who care about this stuff and to collect the results from various architectures.
 4. Provide a simple, standard way to quickly do one-off tests of some small assembly or C/C++ level idioms. Often the test itself is a few lines of code, but the cost is in all the infrastructure: implementing the timing code, converting measurements to cycles, removing outliers, running the tests for various parameters, reporting the results, whatever. This project aims to implement that infrastructure and make it easy to add your own tests (not complete!).

Platform support

Currently only supports x86 Linux, but Windows should arrive at some point, and one could even imagine a world with OSX support.

Prerequisites

You need some C++ compiler like `g++` or `clang++`, but if you are interested in this project, you probably already have that. Beyond that, you need `nasm` and perhaps `msr-tools` on Intel platforms (used to as a backup method to disable turbo-boost if you aren't using `intel_pstate` driver). On Debian-like systems, this should do it:

```
1 sudo apt-get install nasm
2 sudo apt-get install msr-tools
```

NASM

The minimum required version of `nasm` is 2.12, for AVX-512 support (strictly speaking, some later version of `nasm` 2.11, e.g., `nasm-2.11.08` also work). If you don't have `nasm` installed, a suitable version (on Linux) is used automatically from the included `/nasm-binaries` directory.

Building

This project has submodules, so it is best cloned with the `--recursive` flag to pull all the submodules as well:

```
1 git clone --recursive https://github.com/travisdowns/uarch-bench
```

If you've already cloned it without `--recursive`, this should pull in the submodules:

```
1 git submodule update --init
```

Then just run `make` in the project directory. If you want to modify any of the make settings, you can do it directly in `config.mk` or in a newly created local file `local.mk` (the latter having the advantage that this file is ignored by git so you won't have any merge conflicts on later pulls and won't automatically commit your local build settings).

For more about building, see BUILDING.md.

Running

Ideally, you run `./uarch-bench.sh` as root, since this allows the permissions needed to disable frequency scaling, as well as making it possible use `USE_LIBPFC=1` mode. If you don't have root or don't want to run a random project as root, you can also run it as non-root as `uarch-bench` (i.e., without the wrapper shell script), which will still work with some limitations. There is currently an open issue for making non-root use a bit smoother.

With Root

Just run `./uarch-bench.sh` after building. The script will generally invoke `sudo` to prompt you for root credentials in order to disable frequency scaling (either using the `no_turbo` flag if `intel_pstate` governor is used, or `rdmsr` and `wrmsr` otherwise).

Without Root

You can also run the binary as `./uarch-bench` directly, which doesn't require `sudo`, but frequency scaling won't be automatically disabled in this case (you can still separately disable it prior to running `uarch-bench`).

Command Line Arguments

Run `uarch-bench --help` to see a list and brief description of command line arguments.

Frequency Scaling

One key to more reliable measurements (especially with the timing-based counters) is to ensure that there is no frequency scaling going on.

Generally this involves disabling turbo mode (to avoid scaling above nominal) and setting the power saving mode to performance (to avoid scaling below nominal). The `uarch-bench.sh` script tries to do this, while restoring your previous setting after it completes.

Example Output

```
1 $ sudo ./uarch-bench.sh
2 Driver: intel_pstate, governor: performance
3 Vendor ID: GenuineIntel
4 Model name: Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz
5 Successfully disabled turbo boost using intel_pstate/no_turbo
6 Using timer: clock
7 Welcome to uarch-bench (caa208f-dirty)
8 Supported CPU features: SSE3 PCLMULQDQ VMX SMX EST TM2 SSSE3 FMA CX16
   SSE4_1 SSE4_2 MOVBE POPCNT AES AVX RDRND TSC_ADJ BMI1 HLE AVX2 BMI2
   ERMS RTM RDSEED ADX INTEL_PT
9 Pinned to CPU 0
10 Median CPU speed: 2.193 GHz
11 Running benchmarks groups using timer clock
12
13 ** Running group basic : Basic Benchmarks **
14           Benchmark      Cycles      Nanos
15           Dependent add chain      1.00      0.46
16           Independent add chain      0.26      0.12
17           Dependent imul 64->128      3.00      1.37
18           Dependent imul 64->64      3.00      1.37
19           Independent imul 64->128      1.01      0.46
20           Same location stores      1.00      0.46
21           Disjoint location stores      1.00      0.46
22           Dependent push/pop chain      5.00      2.28
23           Independent push/pop chain      1.00      0.46
24           Simple addressing pointer chase      4.00      1.83
25           Complex addressing pointer chase      5.01      2.28
26 Finished in 556 ms (basic)
27
28 ** Running group memory/load-parallel : Parallel loads from fixed-size
   regions **
29           Benchmark      Cycles      Nanos
30           16-KiB parallel load      0.53      0.24
31           24-KiB parallel load      0.52      0.24
32           30-KiB parallel load      0.53      0.24
33           31-KiB parallel load      0.53      0.24
34           32-KiB parallel load      0.52      0.24
35           33-KiB parallel load      0.54      0.24
36           34-KiB parallel load      0.55      0.25
37           35-KiB parallel load      0.56      0.26
38           40-KiB parallel load      1.34      0.61
39           48-KiB parallel load      2.01      0.92
40           56-KiB parallel load      2.01      0.92
41           64-KiB parallel load      2.01      0.92
42           80-KiB parallel load      2.06      0.94
43           96-KiB parallel load      2.18      0.99
44           112-KiB parallel load      2.27      1.03
45           128-KiB parallel load      2.24      1.02
46           196-KiB parallel load      2.72      1.24
```

```

47         252-KiB parallel load      3.75      1.71
48         256-KiB parallel load      3.68      1.68
49         260-KiB parallel load      0.53      0.24
50         384-KiB parallel load      4.34      1.98
51         512-KiB parallel load      5.19      2.36
52         1024-KiB parallel load     5.64      2.57
53         2048-KiB parallel load     6.16      2.81
54 Finished in 7050 ms (memory/load-parallel)
55
56 ** Running group memory/store-parallel : Parallel stores to fixed-size
    regions **
57         Benchmark      Cycles      Nanos
58         16-KiB parallel store      1.00      0.46
59         24-KiB parallel store      1.00      0.46
60         30-KiB parallel store      1.17      0.53
61         31-KiB parallel store      1.00      0.46
62         32-KiB parallel store      1.00      0.46
63         33-KiB parallel store      1.15      0.52
64         34-KiB parallel store      1.32      0.60
65         35-KiB parallel store      1.29      0.59
66         40-KiB parallel store      4.32      1.97
67         48-KiB parallel store      6.20      2.83
68         56-KiB parallel store      6.23      2.84
69         64-KiB parallel store      6.10      2.78
70         80-KiB parallel store      6.25      2.85
71         96-KiB parallel store      6.24      2.84
72         112-KiB parallel store     6.26      2.85
73         128-KiB parallel store     6.26      2.86
74         196-KiB parallel store     6.36      2.90
75         252-KiB parallel store     6.71      3.06
76         256-KiB parallel store     6.75      3.08
77         260-KiB parallel store      1.01      0.46
78         384-KiB parallel store      7.78      3.55
79         512-KiB parallel store      8.67      3.95
80         1024-KiB parallel store     9.59      4.37
81         2048-KiB parallel store    9.97      4.55
82 Finished in 14892 ms (memory/store-parallel)
83
84 ** Running group memory/prefetch-parallel : Parallel prefetches from
    fixed-size regions **
85         Benchmark      Cycles      Nanos
86         16-KiB parallel prefetcht0  0.50      0.23
87         16-KiB parallel prefetcht1  0.50      0.23
88         16-KiB parallel prefetcht2  0.50      0.23
89         16-KiB parallel prefetchnta 0.50      0.23
90         32-KiB parallel prefetcht0  0.50      0.23
91         32-KiB parallel prefetcht1  1.98      0.90
92         32-KiB parallel prefetcht2  1.99      0.91
93         32-KiB parallel prefetchnta 0.50      0.23
94         64-KiB parallel prefetcht0  2.00      0.91
95         64-KiB parallel prefetcht1  1.90      0.86

```

96	64-KiB parallel prefetcht2	2.00	0.91
97	64-KiB parallel prefetchnta	5.90	2.69
98	128-KiB parallel prefetcht0	2.26	1.03
99	128-KiB parallel prefetcht1	2.04	0.93
100	128-KiB parallel prefetcht2	2.04	0.93
101	128-KiB parallel prefetchnta	5.91	2.69
102	256-KiB parallel prefetcht0	3.66	1.67
103	256-KiB parallel prefetcht1	3.49	1.59
104	256-KiB parallel prefetcht2	3.49	1.59
105	256-KiB parallel prefetchnta	5.85	2.67
106	512-KiB parallel prefetcht0	5.25	2.39
107	512-KiB parallel prefetcht1	4.90	2.23
108	512-KiB parallel prefetcht2	4.90	2.24
109	512-KiB parallel prefetchnta	5.77	2.63
110	2048-KiB parallel prefetcht0	6.22	2.84
111	2048-KiB parallel prefetcht1	5.84	2.66
112	2048-KiB parallel prefetcht2	5.84	2.66
113	2048-KiB parallel prefetchnta	9.43	4.30
114	4096-KiB parallel prefetcht0	10.96	5.00
115	4096-KiB parallel prefetcht1	10.69	4.87
116	4096-KiB parallel prefetcht2	10.74	4.90
117	4096-KiB parallel prefetchnta	9.58	4.37
118	8192-KiB parallel prefetcht0	16.96	7.73
119	8192-KiB parallel prefetcht1	16.44	7.50
120	8192-KiB parallel prefetcht2	16.77	7.64
121	8192-KiB parallel prefetchnta	12.27	5.59
122	32768-KiB parallel prefetcht0	20.60	9.39
123	32768-KiB parallel prefetcht1	20.23	9.22
124	32768-KiB parallel prefetcht2	20.09	9.16
125	32768-KiB parallel prefetchnta	20.22	9.22
126	Finished in 4492 ms (memory/prefetch-parallel)		
127			
128	** Running group memory/pointer-chase : Pointer-chasing **		
129	Benchmark	Cycles	Nanos
130	Simple addressing chase, half diffpage	6.51	2.97
131	Simple addressing chase, different pages	8.49	3.87
132	Simple addressing chase with ALU op	6.01	2.74
133	load5 -> load4 -> alu	10.02	4.57
134	load4 -> load5 -> alu	11.03	5.03
135	8 parallel simple pointer chases	4.00	1.82
136	10 parallel complex pointer chases	5.16	2.35
137	10 parallel mixed pointer chases	5.19	2.37
138	Finished in 916 ms (memory/pointer-chase)		
139			
140	** Running group memory/load-serial : Serial loads from fixed-size regions **		
141	Benchmark	Cycles	Nanos
142	16-KiB serial loads	4.00	1.82
143	24-KiB serial loads	4.00	1.82
144	30-KiB serial loads	4.00	1.82
145	31-KiB serial loads	4.00	1.82

```

146          32-KiB serial loads      4.01      1.83
147          33-KiB serial loads      6.02      2.74
148          34-KiB serial loads      8.01      3.65
149          35-KiB serial loads      9.81      4.47
150          40-KiB serial loads     11.93      5.44
151          48-KiB serial loads     11.96      5.45
152          56-KiB serial loads     11.95      5.45
153          64-KiB serial loads     12.12      5.52
154          80-KiB serial loads     11.98      5.46
155          96-KiB serial loads     11.98      5.46
156         112-KiB serial loads     12.01      5.48
157         128-KiB serial loads     12.00      5.47
158         196-KiB serial loads     15.10      6.88
159         252-KiB serial loads     21.27      9.70
160         256-KiB serial loads     21.11      9.63
161         260-KiB serial loads     20.99      9.57
162         384-KiB serial loads     28.64     13.06
163         512-KiB serial loads     31.71     14.46
164        1024-KiB serial loads     38.92     17.74
165        2048-KiB serial loads     47.21     21.52
166 Finished in 683 ms (memory/load-serial)
167
168 ** Running group bmi : BMI false-dependency tests **
169           Benchmark      Cycles      Nanos
170       dest-dependent tzcnt      3.00      1.37
171       dest-dependent lzcnt      3.00      1.37
172       dest-dependent popcnt      3.00      1.37
173 Finished in 190 ms (bmi)
174
175 ** Running group studies/vzeroall : VZEROALL weirdness **
176           Benchmark      Cycles      Nanos
177       vpaddq zmm0, zmm0, zmm0 Skipped because hardware doesn
178                               't support required features: [AVX512F]
179       vpaddq zmm0, zmm1, zmm0 Skipped because hardware doesn
180                               't support required features: [AVX512F]
181       vpaddq zmm0, zmm16, zmm0 Skipped because hardware doesn
182                               't support required features: [AVX512F]
183       vpxor zmm16; vpaddq zmm0, zmm16, zmm0 Skipped because hardware doesn
184                               't support required features: [AVX512F]
185       vpaddq ymm0, ymm0, ymm0      1.00      0.46
186       vpaddq ymm0, ymm1, ymm0      1.00      0.46
187       vpaddq xmm0, xmm0, xmm0      1.00      0.46
188       vpaddq xmm0, xmm1, xmm0      1.00      0.46
189       paddq  xmm0, xmm0      1.00      0.46
190       paddq  xmm0, xmm1      1.00      0.46
191 Finished in 97 ms (studies/vzeroall)
192 Reverting no_turbo to 0
193 Succesfully restored no_turbo state: 0

```