L41: Lab 3 - Micro-Architectural Implications of IPC

Dr Robert N.M. Watson Dr Graeme Jenkinson

Michaelmas Term 2017

The goals of this lab are to:

- Introduce hardware performance counters (hwpmc).
- Explore micro-architectural implications of IPC.
- Gather additional data to support the writing of your first assessed lab report.

You will do this by applying PMC to analyse the behaviour of the same potted, kernel-intensive IPC benchmark used in *L41: Lab 2 - Kernel Implications of IPC*.

Background: Performance Monitoring Counters (PMC)

Hardware performance counters are a low-level processor facility that gathers statistics about *architectural* and *micro-architectural* performance properties of code execution and data access.

Architectural features are those exposed explicitly via the documented instruction set and device interfaces – e.g., the number of instructions executed, or the number of load instructions executed.

Micro-architectural features have to do with the programmer-transparent implementation details, such as pipelining, superscalar execution, caches, and so on – e.g., the number of L2 cache misses taken.

The scope for *programmer transparency* (e.g., what is included in the architecture vs. micro-architecture) varies by Instruction-Set Architecture (ISA): whereas MIPS exposes certain pipelining effects to the programmer (e.g., branch-delay slots), ARM and x86 minimise the visible exposure other than performance impact. MIPS and ARM both require explicit cache management by the operating system during I/O operations and code loading, whereas x86 also masks those behaviours.

Performance counters can be used in two ways: *counting*, in which instances of a particular architectural or micro-architectural event are counted during program execution; and *sampling*, in which $1/n$ instances of the event will trigger a hardware trap that allows, for example, a stack trace to be taken (similar to historic timer-driven profiling techniques). In this lab, we will use PMC only in counting mode.

PMC support may be integrated into the operating system in a variety of ways. Typically, this is done by an additional tracing and profiling framework: in FreeBSD, HWPMC; in Linux, OProfile and related tools. It is also possible to integrate PMC support with DTrace, as has been done in Solaris, but not yet in FreeBSD. On FreeBSD, HWPMC provides a programming API that allows applications to measure their own micro-architectural impacts. We have integrated explicit PMC support into the L41 IPC benchmark using these APIs, allowing it to count events such as memory accesses and cache misses at various points in the cache hierarchy.

The ARM Cortex A8, used in the BeagleBone Black, can track events using up to four sources at a time; we will typically track the number of cycles, the number of instructions executed architecturally (i.e., that weren't canceled in the pipeline due to, for example, a branch mispredict), and then pairs of counters tracking a particular part of the cache hierarchy. We will focus almost exclusively on memory-related counters, rather than looking at other micro-architectural performance events such as branch prediction. This is because our IPC benchmark results will be most strongly affected by memory footprint of our buffers and IPC primitives.

FreeBSD also includes tools to sample PMC behaviour by process or systemically, capturing stack traces via sampling, and mapping them back to program symbols or annotated source code. You may wish to also use these tools to help explain performance behaviour (i.e., not just that L2 cache misses were dominant at a particular buffer size, but also that the majority of cache misses were taken in a particular part of the kernel), but that is not required for this lab. If you wish to use these tools, please see the FreeBSD `pmcstat(8)` man page for details on capturing counter data for whole-program and whole-system analysis. Note that, although on the hardware side PMC may have no measurable probe effect, the software framework around PMC (i.e., to virtualise counters across multiple processes) can introduce substantial probe effect, which we must be aware of when using these counters for performance analysis.

The benchmark

The IPC benchmark has a `-P` argument that requests use of performance counters to analyse the IPC loop. Performance counters are configured in "process mode", meaning that they track user and kernel events associated with a process and its descendents, so should include events from all three of our benchmark modes including their execution in kernel, but not other system events. Where events occur asynchronously in a kernel thread not explicitly associated with the user process, those events will not be counted (e.g., kernel work performed by a timer on behalf of a user process).

Running the benchmark

Lab 3 employs the same benchmark used in Lab 2. As before, you can run the benchmark using the `ipc-static` and `ipc-dynamic` commands, specifying various benchmark parameters. When the new performance-counter argument is used, additional information will be printed about the processor-level behaviour of the IPC loop. Do ensure that, as in Lab 2, you have increased the kernel's maximum socket-buffer size.

Performance-counter arguments

Performance-counter support are enabled using the `-P` flag, which accepts one argument identifying the set of counters to track during execution. Due to the 4-counter limit in the Cortex A8, it is not possible to count all the potential events of interest at the same time. As such, some care will be required to take multiple samples and consider counter readings as members of a distribution. The following counter modes are supported:

l1d Track cache hits and misses on the L1 data cache. This counter may include the effects of speculated, but canceled, instructions.

l1i Track cache hits on the L1 instruction cache; L1 cache misses are not directly countable on this processor. This counter may include the effects of speculated, but canceled, instructions.

l2 Track cache hits on the L2 cache, which is used for both instruction and data access. L2 cache misses are not directly countable on this processor. This counter may include the effects of speculated, but canceled, instructions.

mem Count architecturally originated memory reads and writes: i.e., load and store instructions. This counter will not include the effects of speculated, but canceled, instructions.

axi Track memory accesses issued over the AXI bus: i.e., to actual DRAM or to perform I/O. Note that I/O accesses can be significant – e.g., network traffic will pass over the AXI bus – so attempt to minimise I/O during the benchmark. This counter may include the effects of speculated, but canceled, instructions.

tlb Track misses in the instruction and data Translation Lookaside Buffers (TLBs), which cache page-table entries in hardware. This counter may include the effects of speculated, but canceled, instructions.

Example benchmark commands

This command instructs the IPC benchmark to capture information on memory instructions issued when operating on a socket with a 512-byte buffer from a single thread:

```
# ipc/ipc-static -i local -b 512 -P mem 1thread
```

This command performs the same benchmark while tracking L1 data-cache hits and misses:

```
# ipc/ipc-static -i local -b 512 -P l1d 1thread
```

This command performs the same benchmark while tracking L2 cache hits:

```
# ipc/ipc-static -i local -b 512 -P l2 1thread
```

And this command performs the same benchmark while tracking memory operations that make it out the bus to DRAM (or I/O devices):

```
# ipc/ipc-static -i local -b 512 -P axi 1thread
```

Cortex A8 caches

The ARM Cortex A8 has independent level-1 instruction and data caches (each 32k) and a shared instruction/data level-2 cache (256k). The cache line size is 64 bytes, and most counters will refer to cache lines rather than bytes of memory. For example, the rough utilised memory bandwidth of the system might be estimated as the sum of AXI reads and writes multiplied by 64, although the actual data used will depend on how effectively software has been able to pack data into cache lines. As we are working with virtually contiguous buffers and most access is via memory copies, this is a reasonable approximation in our environment.

Performance counters

The following performance counters are exposed by the IPC benchmark via its various PMC modes:

AXI_READ The number of AXI-bus read transactions.

AXI_WRITE The number of AXI-bus write transactions.

CLOCK_CYCLES The number of clock cycles.

DTLB_REFILL The number of data-TLB misses.

INSTR_EXECUTED The number of instructions executed architecturally.

ITLB_REFILL The number of instruction-TLB misses.

L1_DCACHE_ACCESS The number of L1 data-cache hits.

L1_DCACHE_REFILL The number of L1 data-cache misses.

L1_ICACHE_REFILL The number of L1 instruction-cache misses.

L2_ACCESS The number of L2 cache hits.

MEM_READ The number of memory read instructions that executed architecturally.

MEM_WRITE The number of memory write instructions that executed architecturally.

Note on graphs in this lab report

Because of the large amounts of data (and number of data sets) explored in this lab, you will need to pay significant attention in writing the lab report to how you present data visually. Graphs should make visual arguments, and how a set of graphs are plotted can support (or confuse) that argument. Make sure all graphs are clearly presented with labels and textual descriptions helping the reader identify the points you think are important.

When two graphs have the same independent variable (e.g., buffer size), it is important that they use the same X axis in terms of labelling and scale. Graphs with the same X axis will often benefit from being arranged so that they align horizontally on the page, such that inflection points can be visually compared.

Where an X axis is identical, and dependent variables have the same Y axis (e.g., both measure bandwidth and have the same scale), placing them on the same graph is frequently useful, as visual artefacts (such as intersecting lines, differing slopes) have specific meaning and will pop out at readers. Be careful to clearly label different lines, and ideally use shading, point symbol, and/or colour to make the visual distinction clear. If you have having trouble distinguishing the different data sets, then there are too many data sets on the graph.

Where an X axis is identical, but dependent variables differ on their scales (e.g., one measures bandwidth, and a second cache miss rate), placing them on the same graph could lead to confusion as, for example, line intersections may not actually have meaning. You can, however, vertically stack multiple graphs on the same X axis, allowing inflection points and changes in slopes to be visually compared. Do this by aligning the X axes of the two graphs, and then 'squinching' (a technical term) the two close together; as the X axes will have identical units and values, you can have the graphing package include labels only for the bottom graph. This will allow comparison of linked data – e.g., a larger graph showing bandwidth, and a set of smaller graphs showing micro-architectural effects such as TLB and cache miss rates, to be visually compared to make it easy to assess possible correlation.

Experimental questions (part 2)

These questions supplement the experimental questions in the Lab 2 handout. As with the configuration described in the prior handout, they are with respect to a fixed total IPC size, statically linked version of the benchmark, and refer only to IPC-loop and not whole-program analysis. Consider all three IPC types (pipe, socket, socket with `-s`) and both `2thread` and `2proc` models:

- How does changing the IPC buffer size affect the architectural and micro-architectural aspects of cache and memory behaviour – and why?
- Can we reach causal conclusions about the scalability of the kernel's pipes and local socket implementations given additional evidence from processor performance counters?

Your lab report must address the experimental questions in both Lab 2 and this lab.