Quantifying the OS Kernel-induced Memory System Interference in Multi-threaded Server Workloads

Nayan Deshmukh
Indian Institute of Technology Kanpur
nayan26deshmukh@gmail.com

Mainak Chaudhuri
Indian Institute of Technology Kanpur
mainakc@cse.iitk.ac.in

ABSTRACT

Typical microarchitecture simulators and binary instrumentation tools ignore the OS kernel code when executing a benchmark application. While this is acceptable for compute-intensive applications, this analysis methodology fails for a number of multi-threaded server workloads with deep back-end software stack. In this project, we build an infrastructure which allows tracing user as well as kernel instructions in legacy x86 binaries. The tracing infrastructure is built on top of Qsim which is integrated with QEMU. The infrastructure can be used to collect traces that are rich in kernel code information. So far we have traced MySQL server responding to TPC-C and TPC-H queries coming from eight client connections, and Apache server responding to SPECWeb queries coming from eight simultaneous HTTP connections. We replay these traces through the Multi2Sim microarchitecture simulator modeling an eight-core chip-multiprocessor with three levels of caches, two levels of TLBs per core, and all traditional details of a contemporary server processor. The simulator uses a detailed multi-channel multi-ranked multi-banked DRAM model developed using DRAMSim2.

We use the simulation results to understand the effect of OS kernel activities on the memory system performance such as mid-level and last-level cache misses, DRAM row buffer hit rates, DRAM bandwidth demand, etc.. We study the detailed characteristics of the kernel and user codes in MySQL and Apache and their interactions during the runtime using our infrastructure.

1. INTRODUCTION

Software simulation is the de-facto way to verify the evaluating new ideas in computer architecture. With the increasing complexity of the computer system, the new ideas are first tested, evaluated, prototyped and verified in software simulator before proceeding with the actual hardware implementation. This puts microarchitectural simulators at the forefront of the innovation in computer architecture. Simulating the entire system in software can be slow can sometimes lead to large simulation times. Therefore a lot of simulators compromise on the accuracy at the cost of making the simulator. There is always a tradeoff between accuracy of the simulation and the simulation time. One common compromise made by a lot of simulators is omitting the OS kernel instruction from the simulation. While this might be justified for compute-intensive workloads which spent a minor fraction of their time in the kernel, but for multithreaded server workloads, this could lead to wrong conclusions. We show that ignoring OS kernel could lead to deviations in performance stats which could lead to errors as large as 105%.

2. BACKGROUND

2.1 QSim

Most architectural simulators can be divided into two parts, the frontend and the backend. The frontend is responsible for emulating ISA and maintaining the register file state. The backend is where the simulation happens and is the component that distinguishes the simulator. The backend and frontend are tightly coupled in these simulators. Some of these simulators like gem5 [2], have multiple frontends to allow simulation of multiple ISAs like x86, ARM etc but has a single backend for all the frontends. The frontend is generally a significant portion of source code of simulators and represents works that need not be replicated for each new simulator. QSim [3] tries to avoid this redundant work by providing a generic frontend which exposes a standard API which can then be used with a multitude of backends. We will discuss the QSim API in section 2.1.1.

QSim is implemented as a library with C++ API (called QSim API) that uses Qemu [1] as an emulator. The Qemu translation cache is modified to triggers from the helper functions that emulate the guest ISA.

2.1.1 QSim API

The frontend needs to send execution information to the backend. The exact information and frequency of sending the information depend on the backend e.g. A backend simulating the memory hierarchy only requires the information about memory accesses, whereas an instruction trace generation backend needs information about all the instructions. QSim uses callbacks to send information to the backend. Figure 1 shows a simplified view of the QSim API. Table 1 gives information about the functions exposed by QSim API.

function	Description
run (i)	Advance guest by i instructions
set_∗_callbacks(x)	Set callbacks
unset_∗_callback(h)	Unset callbacks

Table 1: Description of the QSim API

The QSim API allows the backend to set callbacks which are triggered by specific events. The QSim allow callbacks to be set for the following events:

- Every instruction
- Atomic memory operations
- Memory reads/writes
- Register reads/writes

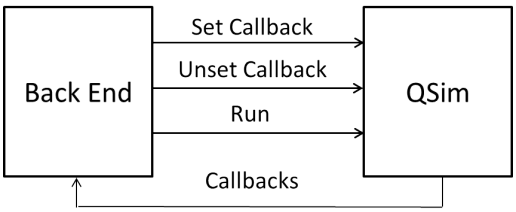


Figure 1: A simplified view of the QSim API

- Interrupts

A backend simulating a memory hierarchy only requires getting the information related to memory accesses. So it can set callbacks for the memory reads/writes. For a backend collecting instruction traces, we need to set the callback for each instruction. Figure 2 shows the architecture of QSim. QSim allows two modes of execution: (i) non-interactive, (ii) interactive. The non-interactive mode only executes the required benchmark on a kernel with an initial RAM disk which is built using *busybox*. The benchmarks need to be statically compiled for them to work in the non-interactive mode, this might not be easy for certain benchmarks like MySQL. The interactive mode has a root filesystem and runs a full-fledged OS e.g. Debian. This allows us to trace applications without the need to statically link them. In the interactive mode, QSim allows us to trace a particular process as there are a lot of other process running the system. To trace a particular process we need to modify its code and insert a specific instruction (called the qsim start marker) at the start of the application, similarly we need to add a instruction at the end of the application to end the tracing. This markers also allow us to trace a specific region of interest instead of the entire process. However this requires us to modify the code of the benchmark and compile it, this might be difficult for certain applications with large code-bases and will not be even possible for closed source benchmarks. We address this limitation in section 3. We use QSim with a trace-generating backend and simulate the traces using a trace-driven simulation.

2.2 Multi2Sim

Multi2Sim [5] is an execution based simulator, which means that benchmark is directly executed. The simulation is done simultaneously with the program execution. However, we use a modified version of Multi2Sim which is driven by the trace. We feed Multi2Sim with the traces that we collect from QSim. Trace-driven Multi2Sim allows us to pin a trace file to a core and then that core executes the instructions from that file.

Component	Parameters
Processor	8 cores, 1 thread per core
iL1 and dL1 caches	32 KB, 8-way, LRU, latency 1 cycle
L2 cache	128 KB, 8-way, LRU, latency 3 cycles
Shared LLC	4 MB, 16-way, 8 banks, LRU, latency 5 cycles
Memory controllers	two single-channel DDR3-1066 controllers
DRAM modules	modeled using DRAMSim2, 14-14-14, BL=8, 64-bit channels, one rank/channel, 8 banks/rank, 1 KB row/bank/device

Table 2: Configuration of the simulated system

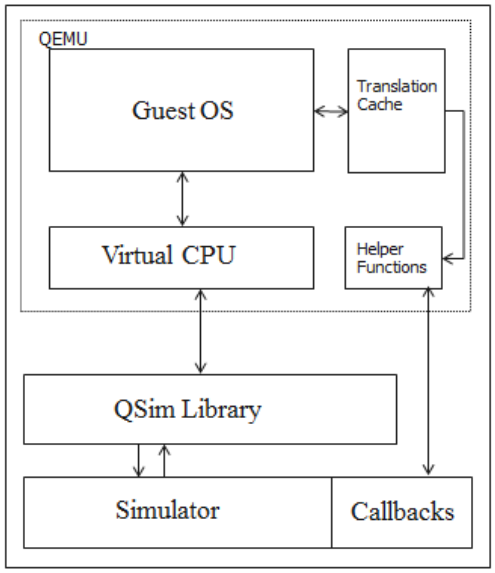


Figure 2: Block diagram of Qsim. Source: [4]

3. OUR WORK

QSim was lacking certain features that were required for trace generation. We discus some of the significant features that we implemented on top Qsim.

3.1 Tracing unmodified binaries

In the interactive mode, QSim requires the application code to be modified to include the start and end markers which signal the QSim fronted to start/stop triggering the callbacks. But modifying the source code of the benchmarks might not feasible for cases where the finding the region of interest is difficult to the large size of the codebase. For closed source benchmarks modifying the source might not even be an option like Adobe Photoshop.

We modify QSim to allow tracing an already running process by specifying its PID. We overload the cpuid instruction to implement this by sending the PID in the *ebx* register from the simulation environment. This allows tracing unmodified binaries.

3.2 Exposing thread ID details

QSim allows tracing applications at the process granularity by add adding the start and end markers or with the feature developed in section 3.1. It triggers the callbacks for all the threads that are created by the process. But it does not provide us the information about the thread which triggered the callback. This information will be useful for multithreaded applications like MySQL which spawns a number of threads to deal with specific actions like flushing dirty pages [6]. Since we might need to analyze the performance of individual threads and study their behavior.

3.3 Fixing bugs

While working with QSim we also encountered a few bugs. Some of the significant bugs that we faced were:-

- **Certain memory accesses do not trigger callbacks:** e.g. the memory accesses by the cmpxchg instruction did not lead to memory callbacks. The helper code responsible for emulating the cmpxchg was not being trapped for memory access and hence it did not lead to memory access callbacks.

Suite	Applications	Input/Configuration	Simulation length
TPC	MySQL TPCC	10 GB database, 1 GB buffer pool, 100 warehouses, 8 clients	285 transactions
	MySQL TPC-H	2 GB database, 1 GB buffer pool, 8 clients, zero think time, even distribution of Q6, Q8, Q11, Q13, Q16, Q20 across client threads	Five billion instructions
SPEC Web	Apache HTTP server v2.4	Banking (SPEC Web-B), Ecommerce (SPEC Web-E), Support (SPEC Web-S); Worker thread model, 8 simultaneous sessions, mod_php module	Five billion instructions

Table 3: Simulated Applications

- **The size of the memory accesses do not correspond to their respective instructions:** e.g. The size of memory access by a 4-byte load is reported as 2 bytes. The problem was due to a swap of arguments in the memory access callback which lead to wrong memory size being reported.

  The fixes for this bugs were patched upstream.

4. SIMULATION INFRASTRUCTURE

In our infrastructure, we have used QSim with a tracing generating backend, called *Tracegen*. Tracegen uses instruction and memory access callbacks to collect the traces of the instruction along with the address that they access. Tracegen uses the feature developed in 3.2, to collect thread wise trace information. Tracegen also collects the CPU core number on which the instruction was executed, this information is used to replay the effects of kernel scheduling. Tracegen generates instruction file corresponding to each thread of the benchmark which was executed in the region of interest. These trace files are then processed to create corewise trace files which separate the instruction based on the CPU they were executed on. We also process the file to add dependency information i.e. we need to maintain the global order between loads and stores touching the same 64-byte block. Now we use this trace files with multi2sim, we pin the core file to a core in the multi2sim. This captures the scheduling information along with the trace corresponding to kernel scheduling which is part of the corewise traces. Generally, simulators pin the trace corresponding to a particular thread, this does capture the effects of the scheduling a process which is seen in real systems like TLB flushes register state restoration etc.

The details of the system configuration are given in Table 2. We have collected the stats in two categories for all the benchmarks. The first category (represented by USR) consists of instruction traces which only contain the user space instructions. This is representative of a trace collected from a normal trace generator like intel pintool. We generate this traces by omitting the kernel instruction from the trace files generated by Tracegen. The second category contains instruction traces which include userspace as well the OS kernel instructions, this is the unmodified trace files generated by Tracegen.

5. PERFORMANCE EVALUATION

In this section, we quantitatively evaluate the interference caused by OS kernel instruction with the userspace instructions in the Memory subsystem. We first study the impact of ignoring the OS kernel on the common memory subsystems stats like cache MPKI, DRAM row hits etc. We then analyze the characteristics of the Apache and MySQL servers separately and their interaction with the OS kernel code.

5.1 Impact of ignoring OS kernel code

Figure 3a shows that ignoring kernel code while tracing could lead to significant difference in performance for workloads that spent a significant amount of time in the kernel code. These applications include web and data serving workloads like Apache and MySQL. Stats show that could be a discrepancy of as much as 105 percent for certain workloads. Some stats even show a difference in the trend like SpecWeb support has a higher IPC as compared to SpecWeb e-commerce for the user category but the trend is reversed for the user + OS kernel category. Similar reverse trends can be seen for others stats too.

These reverse trends are harmful to someone who is only looking at the user stats might make a conclusion based on this stats. The figures also show that the interaction between user and OS kernel in the memory subsystem is complicated and can lead to destructive as well as construction interference at various memory hierarchy levels. Even with the same benchmark we see different behavior at different hierarchy levels e.g. MySQL experiences destructive interference at the L1 and L2 cache but show a constructive interference at the LLC and DRAM level.

5.2 Workload characteristics

5.2.1 MySQL server

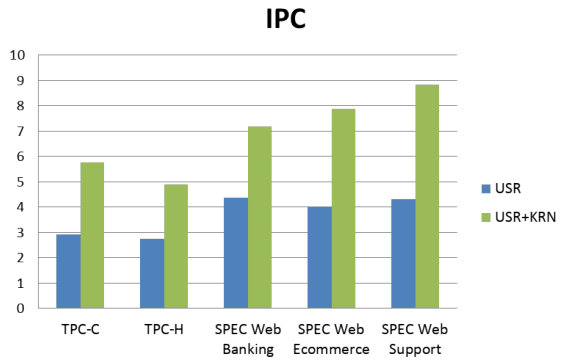
TPC-C and TPC-H show different trends of interference in the memory hierarchy. TPC-C shows constructive interference at the levels whereas TPC-H shows destructive interference at the DRAM and L1 level. Due to the constructive interference at all the levels TPC-C shows a significant amount of performance improvement in the user + OS kernel category. Even though TPC-H show performance improvement owing to the constructive interference at the L2 and LLC, it is hampered due to the destructive interference at the L1 and DRAM level. The code MPKI at L2 is significantly less than L1 code MPKI, this is due to the fact that L2 caches are shared between code and data whereas the L1 has separate caches for code and data. This shows that code memory access has more locality than data access but suffer misses due to the smaller size of L1 instruction cache. Traditionally the L1 data and instruction caches have the same size, but these observations force us to reconsider that decision.

5.2.2 Apache HTTP server

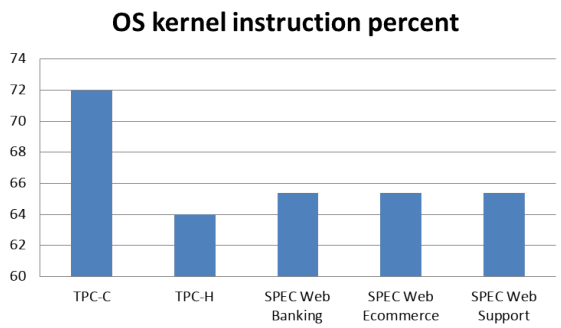
The SpecWeb benchmarks show similar trends in terms of performance and memory subsystem interference. These benchmarks show destructive interference at the first two levels of cache and constructive interference thereafter. As compared to MySQL, a significant portion of Apache server's L1 misses are code misses. As discussed previously the proportion of this misses decreases significantly at the L2, supporting the idea of having a bigger L1 instruction cache. The row misses decrease significantly at the DRAM in OS+kernel category when compared with MySQL.

6. CONCLUSION

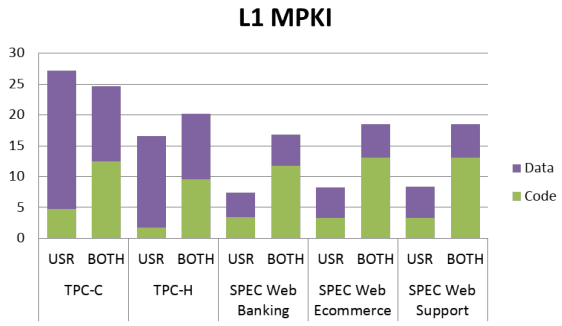
We show that for multithreaded server workloads simulating the full system including the OS kernel is necessary while evaluating new ideas. We propose an infrastructure that can be used as a way



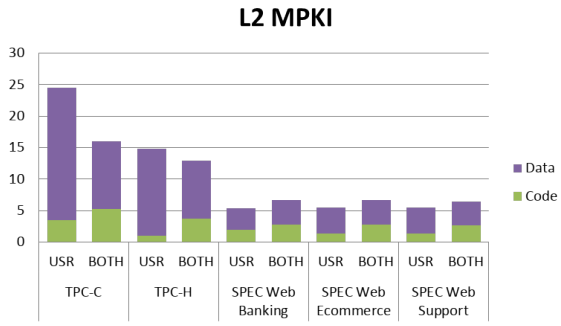
(a) Instruction per cycle, higher the better



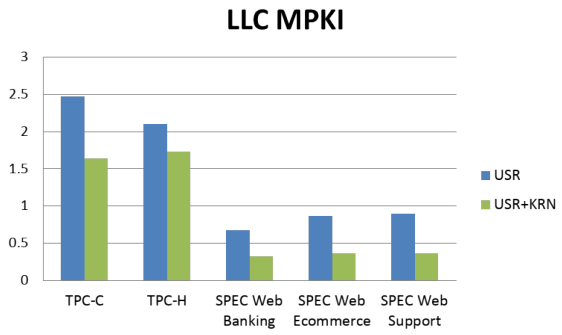
(b) Percentage of kernel instruction in the application traces



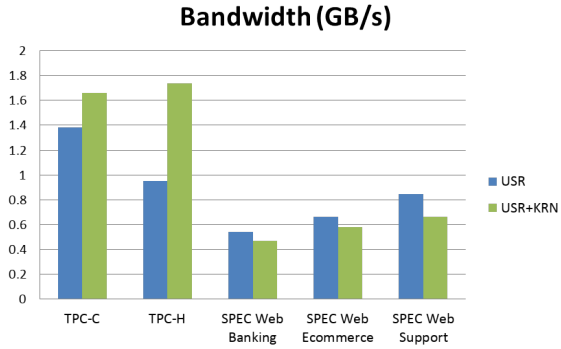
(c) L1 cache misses per kilo instruction, lower the better



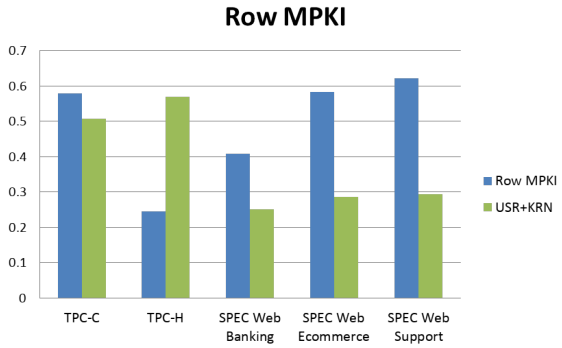
(d) L2 cache misses per kilo instruction, lower the better



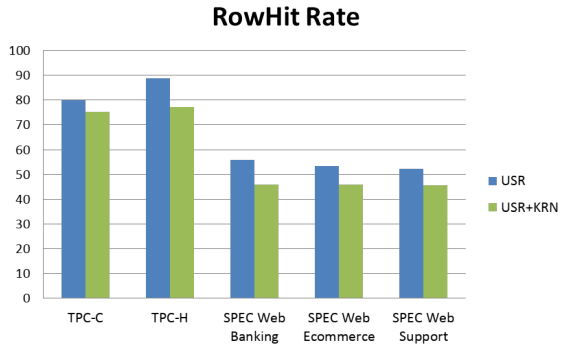
(e) Last level cache misses per kilo instruction, lower the better



(f) DRAM bandwidth required by the application, measured from the LLC misses



(g) Row misses per kilo instruction, a metric similar to cache MPKI



(h) Row Hit rate at the DRAM

Figure 3: Plots represent the difference in various stats seen if we ignore the OS kernel instructions

to simulate multithreaded workloads along with OS kernel activity. We also show that the interaction of OS kernel and userspace code leads to interference at the memory subsystems and this interference needs to be analyzed in detail to improve the performance of such workloads.

7. REFERENCES

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[3] Chad D. Kersey, Arun Rodrigues, and Sudhakar Yalamanchili. A universal parallel front-end for execution driven microarchitecture simulation. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '12, pages 25–32, New York, NY, USA, 2012. ACM.

[4] Pranith Kumar. QSim Version 2.6 User Guide. `http://manifold.gatech.edu/wp-content/uploads/2017/09/user_guide_2.6.pdf`, 2017.

[5] R Ubal, J Sahuquillo, S Petit, and P López. Multi2sim: A simulation framework to evaluate multicore-multithread processors. Citeseer.

[6] Jimmy Yang. New InnoDB Features in MySQL 5.6. `https://www.mysql.com/news-and-events/events/VDD-MySQL-July13/New_InnoDB_Feature_MySQL_5.6_Jimmy.pdf`, 2013.