Runtime Estimation of Application Memory Latency for Performance Analysis and Optimization

Huanxing Shen
huanxing.shen@intel.com
Intel Corporation

Cong Li
cong.li@intel.com
Intel Corporation

ABSTRACT

Various runtime factors impact memory latency and consequently impact application performance. Unfortunately the causal relationship is buried especially at runtime. In this paper we propose a new method for runtime estimation of application memory latency which helps discover the causal relationship. The new method leverages the hardware performance counters to calculate the average time that memory requests wait before getting fulfilled. We evaluate the method empirically in multiple scenarios and the estimation closely approximates the ground truth. We further demonstrate two examples of using the runtime estimation of application memory latency in application performance optimization and analysis, one in mitigating memory access interference in workload co-location and the other in dissecting the performance problem in the memory subsystem.

KEYWORDS

performance analysis, memory access interference, memory latency, workload co-location

ACM Reference Format:
Huanxing Shen and Cong Li. 2020. Runtime Estimation of Application Memory Latency for Performance Analysis and Optimization. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3422575.3422773

1 INTRODUCTION

Memory latency is not a new topic for application performance tuning. Previous studies (see, e.g., [8, 18, 21]) have shown the impact of memory latency on application performance. Recent trends put the memory latency under the spotlight. First, applications, e.g., big data analytics and deep learning models, start to take more memory, making memory access time significant. Second, with the increasing number of CPU cores on a single server, the number of co-located applications raises as well, exposing mounting pressure to the memory subsystem. Third, cloud applications choose to use tail latency to define their service level agreement (SLA) in order to protect the end-user experience, which is a tough performance goal. As a result, researchers are seeking possible improvements from every aspect. In this paper, we focus on runtime memory access performance analysis.

Static performance analysis methods help to spot code level improvements, e.g., data locality, and inspire the hardware-software co-design (see, e.g., [2]). However, the runtime memory latency of an application can be impacted by many factors such as new architecture, hardware settings, application runtime configurations, and workload co-location (see., e.g., [6, 8, 20, 24]). When memory latency goes higher, it jeopardizes the application performance.

We propose a new method to estimate the runtime memory latency of an application. The method leverages the hardware performance counters to calculate the average time in nanoseconds that memory requests wait before getting fulfilled. Experimental results show that our latency estimation closely approximates the ground truth in different scenarios.

We demonstrate the use of the latency estimation in runtime performance analysis and optimization. In the first scenario where workloads are co-located, we use the runtime memory latency of the latency-critical (LC) application as an input for dynamic throttling of a best-effort (BE) application. With the help of the metric, we are able to detect memory interference, secure the performance of LC applications, and improve the throughput of BE applications. In the second scenario where the application performance drops upon a runtime configuration change, we use the latency estimation to dissect the performance problem. The estimation confirms the impact of the configuration change on the application performance.

The rest of the paper is organized as follows. In the next section we first introduce the hierarchy of the memory subsystem. We then propose the new method for runtime estimation of application memory latency in Section 3. After an empirical evaluation of the method in Section 4, we further demonstrate how the estimation can be applied to memory access interference mitigation in Section 5 and to performance diagnosis in Section 6. In Section 7 we discuss the limitations of our method. After a brief review of the other related work in Section 8, we conclude our work in Section 9.

2 BACKGROUND

In modern computing systems, memory is used for fast data storage and retrieval. To alleviate the memory wall problem caused by the processor-memory performance gap [23], modern multi-core processors employ a hierarchy of multi-level caches to accelerate the memory access performance. Figure 1 shows the diagram of a multi-core sever processor. Each core typically has its dedicated level 1 and 2 caches. The cores share the last-level cache (LLC) which is much larger. A read request from a core is checked in its dedicated caches first. Given a miss in the dedicated caches, the request goes to the shared LLC. If the request also gets missed in the LLC, it is forwarded to the shared memory controller (or one of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422773

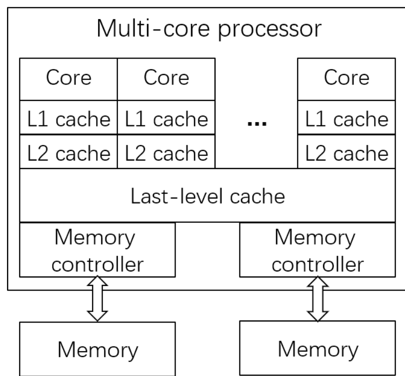

Figure 1: Multi-core server processor.

the several shared controllers) for memory access. Cache accesses are fast, typically taking less than 50 cycles (that is, around 20 to 30 nanoseconds depending on the operating frequency). Memory accesses are slow. With more requests queued for processing by a memory controller, a longer waiting time is included in the total service time. Memory access time can range from 100 to 400 nanoseconds depending on the intensity on some recent server platforms [21]. Once the request is served from the memory, the content is then cached for faster accesses in the future. A write request is performed first in caches as well, making the corresponding cache entries dirty. Dirty entries are typically written back later. In such a design, the performance of read requests is more important than that of write requests [15]. This is because that read requests are more likely to be on the critical path and usually there are more read requests than write requests. In this paper, we limit our study to the read requests only.

Usually modern servers have several processor nodes. Each processor node contains a multi-core processor and a set of local memory modules served by the memory controller(s) in the processor. Those nodes are connected with a high-speed interconnect so that the memory modules from all the nodes can be shared globally. Figure 2 shows a topology on how memory and processors are connected. While accesses to the memory of the local node go through the local memory controller(s), accesses to the memory of a remote node need to traverse the interconnect first. This makes the remote accesses slower, forming the non-uniform memory access (NUMA) design [4].

The runtime latency of memory accesses in an application is impacted by multiple factors. For example, the hardware configuration of memory interleaving may improve the parallelism of memory accesses. While enabling hardware cache prefetching probably makes the cache hit ratio higher, it stresses the memory controllers with more queueing requests and hurts the latency. Given the NUMA design, modern operating systems provide different memory placement policies and different CPU scheduling mechanisms that may impact the memory access latency of the applications. With the commercial persistent memory modules on the market, applications can enjoy the higher memory capacity but at the cost of slower and unstable performance [24].

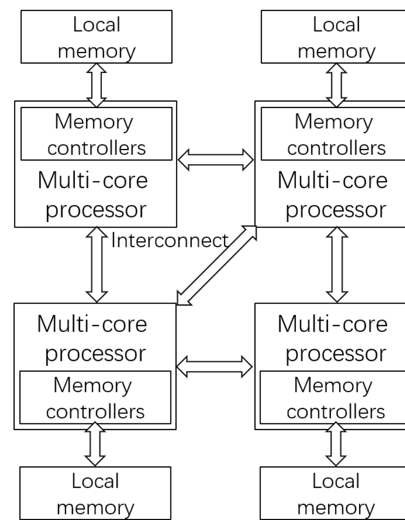

Figure 2: Topology on how memory and processors are connected.

In highly optimized cloud computing environments, many workloads are co-located on a server. This results in more requests waiting in the queues for the shared memory controllers to process. Thus memory accesses of an application may cost more time, impacting its tail latency [9]. For example, Facebook SoftSKU discovers that performance of microservices drops when the number of co-located services reaches a certain threshold [21].

In summary, as a vital factor impacting application performance, runtime memory latency of an application may change with respect to different factors. Estimating the memory latency of an application is important in runtime performance diagnosis and optimization.

3 MEMORY LATENCY ESTIMATION

We propose a method to estimate the average runtime memory latency of an application. While there are cases that read requests can be fed from the data in the cache, in this paper we focus on the requests handled by the memory.

To estimate the average latency of those read requests, we measure their waiting time in nanoseconds before they get fulfilled. The estimation breaks down the life cycle of a data read request which goes to the main memory to cache time and memory time. Let $\bar{l}$ denote the average latency of the requests. We estimate the average latency with

$$\bar{l} = \frac{\bar{c}_{\text{cache}} + \bar{c}_{\text{memory}}}{\bar{f}},$$

in which $\bar{c}_{\text{cache}}$ denotes the average number of CPU core cycles spent in checking and operating the cache, $\bar{c}_{\text{memory}}$ denotes the average number of core cycles spent in serving the requests from memory, and $\bar{f}$ denotes the average core frequency. Compared with $\bar{c}_{\text{memory}}$ which can vary significantly at runtime, the variation of $\bar{c}_{\text{cache}}$ is minor. As a result, we take $\bar{c}_{cache}$ as a processor-dependent constant in the estimation. The denominator, $\bar{f}$, is estimated from

Table 1: Intel's PMU events for memory latency estimation

Notation	Intel PMU event	Semantics
c	CPU_CLK_UNHALTED. THREAD	Unhalted core cycles
c_{ref}	CPU_CLK_UNHALTED. REF_TSC	Unhalted reference cycles
r_{total}	OFFCORE_REQUESTS. L3_MISS_DEMAND_ DATA_RD	Number of unique requests with LLC misses
$\sum_t r_t$	OFFCORE_REQUESTS_ OUTSTANDING.L3_MISS_ DEMAND_DATA_RD	Cumulative cycles that the requests wait before getting fulfilled

the actual CPU core cycles versus the reference core cycles incremented at a fixed frequency as

$$\bar{f} = \frac{c}{c_{\text{ref}}} f_{\text{base}},$$

in which c denotes the *unhalted core cycles* counted at the varing operating frequency, c_{ref} denotes the *unhalted reference cycles* incremented at the fixed reference frequency, and f_{base} denotes the fixed reference frequency. Both c and c_{ref} are directly monitored through the corresponding hardware performance events.

We next explain how we estimate $\bar{c}_{\text{memory}}$. On a contemporary Intel server platform, each core (or hardware thread with hyper-threading enabled) has a *super queue* for memory requests missing its dedicated level 1 and 2 caches. Each request is then checked with the shared LLC. If an LLC miss occurs, the request is labeled as as an LLC miss and is then routed to the memory controller for memory access. Once the request is served from the memory, it gets fulfilled and then gets removed from the queue. On contemporary Intel server processors, hardware performance counters can be programmed to monitor the activities of those requests in the queue. Let r_t denote the number of requests labeled as LLC misses in the queue at cycle t. A counter, $\sum_t r_t$, is incremented with number of ongoing read requests that miss LLC in the queue every cycle. Therefore it counts the cumulative cycles that those requests wait before getting fulfilled within a certain monitoring period. Another counter, r_{total} monitors the total number of unique requests with LLC misses within the same period. Dividing the cumulative waiting cycles by the number of the unique requests, we have

$$\bar{c}_{\text{memory}} = \frac{\sum_t r_t}{r_{\text{total}}}.$$

Table 1 shows the hardware performance events provided by Performance Monitoring Unit (PMU) on Intel SkyLake and Cascade Lake platforms[1] to count the terms for latency estimation. Hardware performance counters can be configured to monitor those events. Since all the 4 events can be counted at core level and hardware thread level, we are able to collect the data at the application, process, or thread level through the Linux `perf` tool[2]. For an application, counters count the data for all its threads within the monitoring period, so the estimation reflects the average memory latency of the application.

[1] See https://www.intel.com/content/www/us/en/design/products-and-solutions/processors-and-chipsets/purley/intel-xeon-scalable-processors.html and https://www.intel.com/content/www/us/en/design/products-and-solutions/processors-and-chipsets/cascade-lake/2nd-gen-intel-xeon-scalable-processors.html.

Table 2: Memory latency estimation vs. MLC measurement.

Configuration	MLC (ns)	Estimation (ns)	Relative error
`idle/local`	88.8	86.31	2.80%
`idle/remote`	148.4	144.46	2.65%
`idle/Optane`	174.5	179.81	3.04%
`loaded`	131.1	128.17	2.23%
`loaded (MBA)`	100.8	99.11	1.68%

4 EVALUATION OF LATENCY ESTIMATION

In this section, we compare our memory latency estimation against the ground truth to evaluate its correctness. We then review an ablation study[3] to discuss why the average CPU frequency is introduced to our estimation method.

We choose to use the memory latency value reported by Intel's Memory Latency Checker (MLC) [22] as the ground truth[4]. To measure the latency, MLC launches a thread to traverse an array of data pointers sequentially and calculates the average read latency for reporting. Therefore the measurement represents the memory access latency of the MLC thread. We create multiple scenarios in which the memory latency of the MLC get impacted upon different background memory loads and runtime hardware/software configuration change. For each scenario, the memory latency of the MLC thread is estimated by the proposed method and compared to the ground truth reported by MLC. If the two results are close, we regard that our estimation is accurate.

Note that though MLC provides the ground truth in our evaluation, it cannot be used as an alternative tool for runtime estimation of application level memory latency. This is because that it only estimates the memory latency of its process or thread under certain background load.

We run the evaluation on a server with two Intel(R) Xeon(R) Gold 6252 processors. The server is equipped with 10 dynamic random-access memory (DRAM) modules, each with the capacity of 32GB, and 2 Intel Optane DC persistent memory modules, each with the capacity of 128GB.

We run the evaluation with the scenarios below

(1) `idle/local`: latency within the same NUMA node without any explicit background memory load in the system;
(2) `idle/remote`: latency towards the remote NUMA node without any explicit background memory load in the system;
(3) `idle/Optane`: latency towards the Intel Optane DC persistent memory module without any explicit background memory load in the system;

[2] See https://perf.wiki.kernel.org/index.php/Main_Page.
[3] In an ablation study, a feature in a method is removed to examine how that affects the performance.
[4] We use the so-called idle-latency testing mode of MLC to get the value, and generate the background load with other tools.

Table 3: Intel's PMU events for memory latency estimation at the memory controller level

Notation	Intel PMU event	Semantics
$r_{\text{total}}^{\text{iMC}}$	UNC_M_RPQ_ INSERTS	Number of unique requests in the memory controller
$\sum_t r_t^{\text{iMC}}$	UNC_M_RPQ_ OCCUPANCY	Cumulative cycles that the requests wait in the memory controller before getting sent
$r_{\text{total}}^{\text{iMC-PMM}}$	UNC_M_PMM_RPQ _INSERTS	Number of unique requests to the persistent memory in the memory controller
$\sum_t r_t^{\text{iMC-PMM}}$	UNC_M_PMM_ RPQ_OCCUPANCY.ALL	Cumulative cycles that the requests to the persistent memory wait in the memory controller before getting fulfilled
f_{iMC}	UNC_M_CLOCKTICKS	Frequency of the memory controller

(4) `loaded`: latency of the base scenario `idle/local`, but augmented with a memory intensive workload in the background simulated by `stress-ng`[5]; and
(5) `loaded (MBA)`: latency of the base scenario `loaded`, but with the background memory load throttled with Memory Bandwidth Allocation (MBA) at 10% of the total memory bandwidth[6].

We use the `perf` tool to collect the hardware performance events listed in the Table 1. The MLC instance runs inside a docker container. For each scenario, we run the MLC instance for 60 seconds. In each second we collect the performance counters and calculate the memory latency. When the run finishes, we compute the average value as the estimated latency of this run. $\bar{c}_{\text{cache}}$, the average number of cycles spent in checking and operating the cache, is set to 44 CPU cycles. f_{base}, the base frequency of the processors, is 2.1GHz.

Table 2 shows the results in those scenarios. In all the scenarios, our memory latency estimations closely approximate the MLC measurements. The highest relative error is around 3%. The results indicate that the proposed method is accurate in estimating the memory latency of the application in the scenarios.

4.1 Comparing with an Alternative Estimation

Suggested in the performance monitoring guide for Intel processors[7], one may use the counters in the shared memory controllers as an alternative method for memory latency estimation.

The alternative approach shares the concept of our memory latency calculation at the super queue level. First, it counts the total clockticks that the memory requests wait at the queue of read requests at the memory controllers before getting sent to the DRAM modules. Then, it divides the total clockticks by the total number of requests to get the average latency in clockticks. Finally, the latency in clockticks is converted to nanoseconds using the frequency of the memory controller.

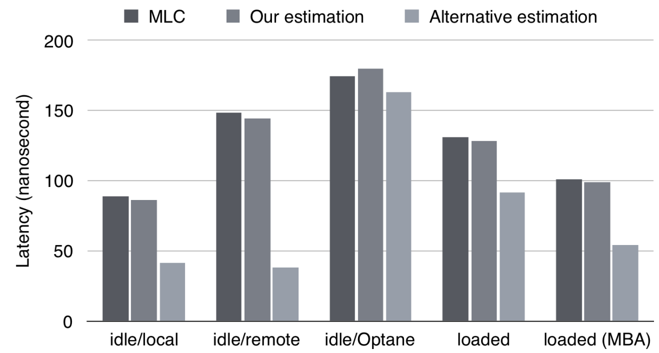

Figure 3: Memory Latency measured by MLC, the memory controller and our estimation for all the scenarios.

Table 3 shows the counters used in the alternative approach. Note that in the memory controllers, for a request going to DRAM, when its read command is issued to the memory module, the request gets immediately removed from the queue. Therefore we add an additional term to account for the so-called *Column Access Strobe (CAS) latency*, the delay time between the read command and the moment data is available. The calculation for requests going to DRAM then becomes

$$\bar{l}^{\text{iMC}} = \frac{\sum_t r_t^{\text{iMC}}}{r_{\text{total}}^{\text{iMC}}} \times \frac{1}{f_{\text{iMC}}} + l^{\text{CAS}},$$

in which l^{CAS} represents the CAS latency. However, for a request going to the persistent memory, it is not removed until it gets fulfilled. The calculation for those requests becomes:

$$\bar{l}^{\text{iMC-PMM}} = \frac{\sum_t r_t^{\text{iMC-PMM}}}{r_{\text{total}}^{\text{iMC-PMM}}} \times \frac{1}{f_{\text{iMC}}}.$$

Figure 3 compared the memory latency values measured by MLC, those estimated by the proposed method, and those estimated by the alternative approach. As we see from the figure, while our estimation closely approximates the ground truth reported from MLC, the alternative estimation deviates away significantly.

There are several reasons making the alternative estimation inaccurate:

(1) Memory controllers are shared by all the CPU cores in a processor. When memory requests from different cores mingle at the controllers, the latency measured is for all the applications running on the cores, not just for the target application running on some specific cores. For example, in the first 2 scenarios, even if there is no explicit background load, there is the implicit background load.
(2) The alternative estimation does not account for the time spent in checking and operating the cache (which may vary with respect to the core frequency). It underestimates the latency in all the scenarios.

[5]Available at https://wiki.ubuntu.com/Kernel/Reference/stress-ng.
[6]See https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation.
[7]See, e.g., https://www.intel.com/content/dam/www/public/us/en/documents/design-guides/xeon-e5-2600-uncore-guide.pdf.

Table 4: Memory latency measured in CPU cycles for different CPU frequency settings

Configuration	MLC (ns)	Estimation in nanoseconds	Estimation in CPU cycles
2.1GHz	81.4	80.2	168.50
2.6GHz	80.0	77.27	200.90

(3) The alternative estimation does not account for the time spent in traversing the interconnect in remote memory accesses. This exacerbates the deviation of the alternative estimation from the ground truth in scenario `idle/remote`.

As a result, the alternative estimation cannot match the accuracy achieved by the proposed estimation approach.

4.2 CPU Cycles vs. Nanoseconds

The memory latency can be measured in CPU cycles without considering the CPU frequency. However, in this way, the memory latency in CPU cycles has a bias towards a lower core frequency. This is because that with a lower core frequency, the cycle time is longer than that at a higher frequency. As a result, a lower latency measured in CPU cycles can actually take more time in nanoseconds.

To illustrate the issue, we perform a simple ablation study. We run the scenario `idle/local` with two different core frequencies, 2.1GHz and 2.6GHz. In the two configurations, we measure the memory latency in both nanoseconds and CPU cycles.

Table 4 shows the latency estimation in nanoseconds and that in CPU cycles along with the ground truth from MLC. As shown by the MLC results, the higher core frequency speeds up the processing so it produces a slightly lower latency. However, without considering the shorter cycle time, the latency in CPU cycles for the higher core frequency is estimated to be higher. This study shows that the memory latency measured in CPU cycles can not be interpreted as the memory access performance directly. Our latency estimation measures the average cycle time in order to translate the memory latency to nanoseconds so it does not suffer from the problem.

5 APPLICATION TO INTERFERENCE MANAGEMENT

We demonstrate the use of runtime memory latency estimation in two scenarios, one for mitigating memory access interference in workload co-location and another for dissecting the performance problem in the memory subsystem.

While workload co-location improves cluster utilization in cloud environments, it brings performance-impacting contentions on unmanaged resources. Memory controllers are shared among all the cores in the processor. When we allocate some cores to a latency-critical (LC) application and some other cores to a best-effort (BE) application[8], memory requests from the two applications mingle at the controllers and may impact each other. Some traditional approaches rely on runtime application level performance data (see, e.g., [10, 27]) to detect and manage the interference. However, application level metrics are not always available for a large variety of workloads. Other approaches are proposed to use low level metrics to detect and manage cache contentions (see, e.g., [17, 19]). However, those methods cannot detect and manage memory access interference.

[8]LC applications are either end-user facing or serving as the basic infrastructure in the cloud. Their workload intensity varies based on the demand. They are typically with a stringent requirement on the tail latency. BE applications, e.g., batch data analytics or machine learning jobs, usually do not have a stringent performance requirement.

Algorithm 1 Memory access interference management

```
procedure manageMemoryInterference(LC, BE)
    θ ← getTargetThreshold(LC, BE)
    quota ← 1
    BE.setCPUQuota(quota)
    while LC.notStopped() and BE.notStopped() do
        l ← estimateMemoryLatency(LC)
        if l < θ then
            if quota < BE.getMaxQuota() then
                quota ← quota + 0.5
        else
            quota ← 1
        BE.setCPUQuota(quota)
procedure getTargetThreshold(LC, BE)
    sum ← 0
    t ← 0
    BE.setCPUQuota(0)
    while t < T do
        if LC.fullyUtilized() then
            sum ← sum + estimateMemoryLatency(LC)
            t ← t + 1
    return sum/T
```

5.1 Use of Memory Latency Estimation to Manage Memory Access Interference

We propose to use the runtime memory latency estimation as the low level metric to detect and manage memory access interference, as is shown in Algorithm 1.

When a new LC application is started, we fully throttle the co-located BE application. During this period, the LC application runs without any memory access interference. When all the cores allocated to the LC application are fully utilized, we estimate its memory latency every second. Let $(\bar{l}_1, ..., \bar{l}_T)$ denote the T samples collected. The average value, $\frac{\sum_{t=1}^{T} \bar{l}_t}{T}$, is then taken as the threshold to detect memory interference.

We then design a dynamic control mechanism to manage the interference from the BE application as follows. We start the BE application with the CPU quota of 1 core. We keep monitoring the runtime memory latency of the LC application every second. If the latency stays below the threshold, we increase the CPU quota of the BE application by 0.5 cores. If the latency violates the threshold, we throttle the CPU quota of the BE application to 1 core.

5.2 Experiment

We use a experiment to evaluate the new approach to manage memory access interference using runtime memory latency estimation.

The deep learning inference workload, Facebook's Deep Learning Recommendation Model (DLRM) [14] benchmark[9], is set up as the LC application. For inference, DLRM provides personalized content based on users' input. Using high dimensional embeddings, the application performance in terms of inference latency becomes vulnerable to the memory access latency [5]. Therefore we choose it to evaluate the mitigation method of memory access interference.

We place the LC application on one processor of the server used in Section 4. To match the memory capacity of the server, we configure the maximum intensity of the DLRM benchmark as running 24 concurrent worker threads, each running inside a Caffe2 [12] thread with AVX2 enabled. Each worker is pinned to a hardware thread in the processor. The average inference latency is 8.92ms. Therefore adding a small buffer, we assume 9ms as the target service level agreement (SLA)[10] to maintain during workload co-location.

We assume that at runtime the LC application is not at its peak intensity. Instead, it is running at a moderate intensity with 9 worker threads. This provides the opportunity to co-locate a certain BE application. Two applications are used: a computing-intensive one and a memory-intensive one. They are simulated by a system stressing tool, stress-ng [7]. For the computing-intensive application, we configure stress-ng to iterate over all the CPU stress methods. For the memory-intensive application, we configure stress-ng to exercise the memory bandwidth. In both cases, 6 worker threads are used. They run on hardware threads which are not currently occupied by the LC application. The choice of 9 LC workers and 6 BE workers is to make the intensity of both applications moderate. With such a setting, we are not only able to observe the memory access interference in some cases but also able to achieve reasonable productivity of the BE application with the interference under control.

To evaluate how well our method performs, we examine the performance of the LC application and the productivity of the BE application. The average inference latency of the DLRM benchmark is checked against its SLA. The throughput of stress-ng in *bogo ops* [11] is recorded as BE's productivity under the co-location. The goal is to maximize the productivity of the BE application without violating the SLA of the LC application.

We also compare our method to 2 baseline settings:

(1) Without quota: without any control on the CPU quota of the BE applications.
(2) Static quota: with a static CPU quota of 1 core on the BE applications.

Figure 4 shows the inference latency of the LC application and the bogo ops of the BE application for three settings. Without CPU quota control, the BE applications achieve the highest productivity but the performance of the LC application is scarified when the BE application is memory-intensive. It is because the memory access interference is not detected and handled. The static one-CPU-quota setting is too conservative for BEs' productivity, especially when the BE application cannot create any pressure on the memory subsystem. With our method, the BE applications achieve decent

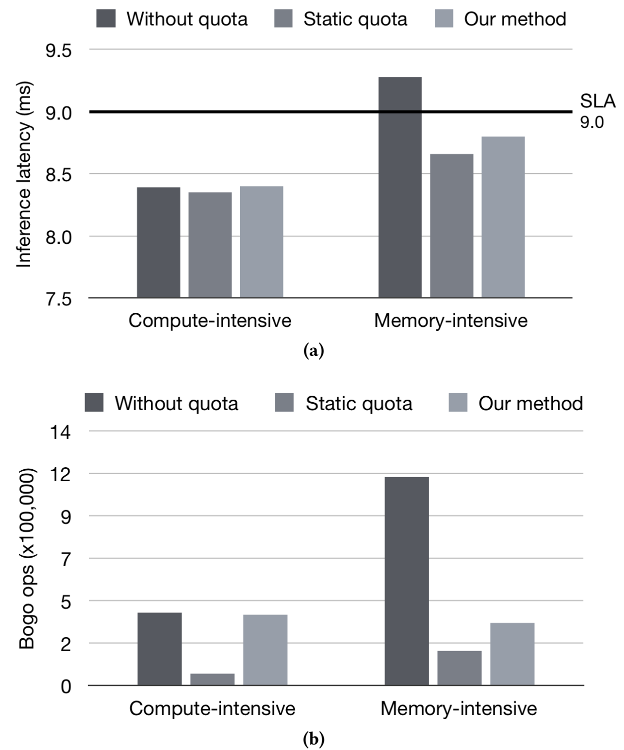

Figure 4: Results of workload co-location (a) inference latency of the LC application; and (b) bogo ops of the BE application.

throughput and the LC performance meets the SLA. Thanks to the runtime memory latency estimation, our method is able to detect performance impacting memory access interference and to dynamic control the CPU quota to mitigate the interference and improve the productivity.

To conclude, the runtime estimation of application memory latency is useful for workload co-location to mitigate the contentions on the shared memory resource. It enables us to detect memory access interference to secure the performance of memory latency sensitive applications. Besides it offers the opportunity for BE applications to thrive as long as they do not impact the memory latency of LC applications.

6 APPLICATION TO PERFORMANCE DIAGNOSIS

In this section we show an example of using the estimation of application runtime memory latency for performance diagnosis.

SPECjbb2015 [16] is a benchmark that simulates the online transaction processing for a wholesale company. The benchmark is memory intensive, as it stores all its data in the memory and accesses the data extensively in processing. In this scenario, We pin the back-end component of SPECjbb2015 to one of the two processors of the server used in Section 4 and place the load generator on a different server in the same local area network. We configure the

[9]See https://github.com/facebookresearch/dlrm.
[10]Note that the SLA information is not known to the interference management algorithm.
[11]*Bogo ops* is the number of completed stressing iterations during the run.

Table 5: Application tail latency and memory latency in two different runs.

Configuration	SPECjbb2015 tail latency (ms) Average / maximum	Memory latency (ns) Average / maximum
Pinned	36 / 91	89 / 149
Not pinned	57 / 640	156 / 261

back-end component with 24 workers. Under this configuration, the throughput score [12] for a standard benchmark run is 38964 transactions per second.

We then run the SPECjbb2015 application twice with the fixed intensity (38964 transactions per second). Each run lasts for 10 minutes and tail latency at the 99th percentile is reported every 35 seconds. In the first run, the back-end component is pinned to one of the two processors. In the second run, the processor pinning constraint is removed. We examine the tail latency for the two runs as the performance indicator for a specific workload intensity.

Table 5 shows the average and maximum value of the tail latency in the two runs. It shows that the tail latency becomes worse when the SPECjbb2015 back-end component is not pinned to one processor.

We speculate that the performance drop is caused by the NUMA design. In the default first-touch NUMA policy in Linux, the memory pages are placed on the local memory node of the processor where they are first accessed. When the back-end component is pinned to the processor, almost all the memory accesses are local ones. However, when the back-end component is not pinned to a specific processor, the Linux CPU scheduler will move the worker threads across the processor boundary [11]. Some of the memory accesses have to go to the inter-processor interconnect thus they are slower than the local memory access. It can eventually affect the tail latency of a memory-intensive application, e.g. SPECjbb2015.

To dissect the performance drop and confirm our speculation, we monitor the memory latency of the SPECjbb2015 back-end component for each run. Table 5 shows the average and maximum value of the memory latency. As we see from the table, in the second run the memory accesses become much slower. This confirms the speculation on how the configuration change impacts the performance. With the help of the runtime memory latency estimation, we can pinpoint the cause of performance deterioration.

7 LIMITATIONS

The estimation of memory latency proposed in the paper applies to read requests only. It assumes that write requests are not on the critical path. For applications in which write requests play a critical role for their performance (e.g., applications using the transactional memory programming), the assumption becomes invalid. The proposed estimation method misses a significant factor in memory latency that may impact application performance.

In the estimation, we assume a constant number of cycles spent in checking and operating the cache. It is possible that the actual number of cycles spent for one request for cache operations can deviate significantly from the constant. For example, if the super queue of a core is full, a request missing the level 1 and 2 caches needs to wait for a free slot in the queue. It is also possible that allocating a new cache line for a read request evicts a dirty cache line. The eviction may result in a chain of evictions in the multi-level cache hierarchy. When all the buffers for data storing are full, it then becomes a write operation costing a significant number of additional cycles. Also in some extreme cases, the LLC controllers in the processor can be too busy in handling the incoming requests, so that the requests in the super queue may need to wait a long time even before knowing the results of LLC misses. Misses in the translation lookaside buffer (TLB) can introduce significant deviation in the cost as well.

Besides we assume that all the read requests missing in the shared LLC become requests to memory. It is possible that in the NUMA design, such a request gets served from the cache in a different processor. The request is then mistakenly regarded as a memory request in our estimation. This becomes a noise factor in estimating the average latency for requests that actually go to memory.

The estimation method proposed in the paper relies on the corresponding counters in the processors. The counters are available in the contemporary Intel server platforms. The method is not applicable to older Intel server platforms. Transferring the method to other non-Intel platforms also depends on the availability of the counters.

8 OTHER RELATED WORK

Efforts have been made to pinpoint the execution bottleneck of an application. For example, Yasin proposes a top-down method to detect the bottlenecks [25] by calculating CPU stalls in different CPU execution pipeline slots. The proposal also relies on hardware performance counters. However, this method provides the relative contribution from a broad hierarchy of different categories (e.g., frontend bound, bad speculation, etc.), but not on the specific estimation of memory latency. Furthermore, many of the categories are more relevant to offline optimization rather than to runtime optimization and performance diagnosis.

Using the right hardware performance counters to analyze the performance of memory subsystem has also been studied. For example, in [13] counters are identified to measure the usage of available bandwidth and the percentage of cycles consumed by the components in the memory hierarchy. However, the work does not address the measurement of memory request latency. It does not demonstrate how the analysis helps in runtime performance optimization as well.

Workload memory characterization is also a hot topic. For example, a performance model is proposed in [3] to evaluate the workloads' sensitivity towards memory bandwidth and memory latency. The model focuses on the characterization of different types of workloads in a static offline environment. However, the static evaluation does not account for the complexity of runtime memory subsystem performance that impacts the application performance.

An analytical memory model is presented in [1] to predict the performance of a program on different processors. The model uses static analysis based on reuse distances to estimate the memory latencies at different hierarchies of the memory subsystem. However,

[12] It is named as *critical-jOPS* in the benchmark report.

the static analysis does not account for the varying runtime factors such as the interference from other co-located workloads.

Managing interference for workload co-location is a difficult problem. In Heracles [10], application performance information is used as the indicator to guide the throttling of the BE application. CPI^2 [26] does not require the application performance information and only uses the cycles-per-instruction metric as application performance indicator. It cannot discriminate whether the interference come from memory accesses or not. In Kelp [28], the LC application and the BE application are placed in different NUMA subdomains to limit the memory access interference. Different from those methods, we use the runtime memory latency estimation of the LC application to detect and mitigate memory access interference. To the best of our knowledge, our work is the only method that can discriminate the interference in memory accesses without application level information or the strict NUMA placement. This helps to develop more flexible resource management policy for workload co-location.

9 CONCLUSION

In this paper we have presented a new method to estimate the runtime memory latency of an application. The measurement is accomplished with the hardware performance counters. The empirical evaluation indicates that the estimation is accurate.

Memory access latency metric provides a unique perspective for runtime performance diagnosis and optimization. We have demonstrated the use of the metric in two scenarios. For workload colocation, we have proposed a new method to detect and manage memory access interference by monitoring the memory latency of the critical application. Experimental results indicate that our method improves the productivity of best-effort applications and secure the performance of latency-critical applications. We have also shown how the memory latency metric can help to dissect the performance loss. With the help of the metric, we are able to pinpoint the cause to the runtime configuration change which impacts memory access performance.

ACKNOWLEDGMENTS

We thank Tai Huang and Jia Bao for comments on an early draft of this paper. We acknowledge the anonymous reviewers for their valuable comments and criticisms to improve the paper.

REFERENCES

[1] Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Scalable Performance Prediction of Codes with Memory Hierarchy and Pipelines. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Chicago, IL, USA) *(SIGSIM-PADS '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3316480.3325518

[2] Zeshan Chishti and Berkin Akin. 2019. Memory System Characterization of Deep Learning Workloads. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 497–505. https://doi.org/10.1145/3357526.3357569

[3] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. 2015. Quantifying the Performance Impact of Memory Latency and Bandwidth for Big Data Workloads. In *2015 IEEE International Symposium on Workload Characterization*. 213–224.

[4] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[5] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2019. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. *CoRR* abs/1906.03109 (2019). https://arxiv.org/abs/1906.03109

[6] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[7] Colin Ian King. 2017. *Stress-ng*. http://kernel.ubuntu.com/~cking/stress-ng/

[8] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 277–289. https://www.usenix.org/conference/atc15/technical-session/presentation/lepers

[9] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670988

[10] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 450–462. https://doi.org/10.1145/2749469.2749475

[11] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326

[12] Aaron Markham and Yangqing Jia. 2017. Caffe2: Portable High-Performance Deep Learning Framework from Facebook. *NVidia Developer Blog* (2017).

[13] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E. Nagel. 2017. Detecting Memory-Boundedness with Hardware Performance Counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3030207.3030223

[14] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[15] Patrick Ndai, Ashish Goel, and Kaushik Roy. 2010. A Scalable Circuit-architecture Co-design to Improve Memory Yield for High-performance Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 08 (aug 2010), 1209–1219. https://doi.org/10.1109/TVLSI.2009.2022628

[16] Hitoshi Oi. 2016. Power Optimization and Service Level Agreement Trade-off in SPECjbb2015. In *2016 IEEE 7th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. 1–6. https://doi.org/10.1109/UEMCON.2016.7777880

[17] Ioannis Papadakis, Konstantinos Nikas, Vasileios Karakostas, Georgios I. Goumas, and Nectarios Koziris. 2017. Improving QoS and Utilisation in Modern Multi-core Servers with Dynamic Cache Partitioning. In *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017, COSH/VisorHPC@HiPEAC 2017, Stockholm, Sweden, January 24, 2017*. 21–26. https://doi.org/10.14459/2017md1344298

[18] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (2010), 65–79.

[19] Huanxing Shen and Cong Li. 2019. Detecting Last-Level Cache Contention in Workload Colocation with Meta Learning. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 14–23. https://doi.org/10.1109/IISWC47752.2019.9041983

[20] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 63–74.

[21] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. 2019. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA,

513–526. https://doi.org/10.1145/3307650.3322227

[22] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, Blazej Filipiak, and Sri Sakthivelu. 2013. *Intel Memory Latency Checker v3.8*. https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[23] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588

[24] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182.

[25] Ahmad Yasin. 2014. A Top-Down Method for Performance Analysis and Counters Architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[26] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. ACM, New York, NY, USA, 379–391. https://doi.org/10.1145/2465351.2465388

[27] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 33–47. https://doi.org/10.1145/2872362.2872394

[28] Haishan Zhu, David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Mattan Erez. 2019. Kelp: QoS for Accelerated Machine Learning Systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 172–184. https://doi.org/10.1109/HPCA.2019.00036