Quantifying the Performance Impact of Memory Latency and Bandwidth for Big Data Workloads

Russell Clapp
Intel Corporation
Hillsboro, Oregon

Martin Dimitrov
Intel Corporation
Chandler, Arizona

Karthik Kumar
Intel Corporation
Chandler, Arizona

Vish Viswanathan
Intel Corporation
Hillsboro, Oregon

Thomas Willhalm
Intel Gmbh
Walldorf, Germany

{Russell.M.Clapp, Martin.P.Dimitrov, Karthik.Kumar, Vish.Viswanathan, Thomas.Willhalm}@intel.com

***Abstract*— In recent years, DRAM technology improvements have scaled at a much slower pace than processors. While server processor core counts grow from 33% to 50% on a yearly cadence, DDR 3/4 memory channel bandwidth has grown at a slower rate, and memory latency has remained relatively flat for some time. Combined with new computing paradigms such as big data analytics, which involves analyzing massive volumes of data in real time, there is a trend of increasing pressure on the memory subsystem. This makes it important for computer architects to understand the sensitivity of the performance of big data workloads to memory bandwidth and latency, and how these workloads compare to more conventional workloads. To address this, we present straightforward analytic equations to quantify the impact of memory bandwidth and latency on workload performance, leveraging measured data from performance counters on real systems. We demonstrate how the values of the components of these equations can be used to classify different workloads according to their inherent bandwidth requirement and latency sensitivity. Using this performance model, we show the relative sensitivities of big data, high-performance computing, and enterprise workload classes to changes in memory bandwidth and latency.**

Keywords— Big Data, Performance Modeling, Real-Time Analytics, Workload Characterization

I. Introduction

Computing time for big data analytics depends on the speed at which a processor is able to reference and analyze the data. The data is often tiered between processor caches, memory, and storage subsystems. Since the processor caches can hold only a limited volume (MBs), and storage subsystems often have large and software-dependent latencies for referencing data, the emphasis of real-time analytics systems is on the memory subsystem. However, as data volumes grow and as compute capabilities continue to increase with Moore's Law, memory (DRAM) density scaling has begun to lag far behind. Fig. 1 shows how this gap is increasing in coming years, and the trend in per channel DDR bandwidth is similar. Furthermore, emerging memory technologies [14,15,16,17] have different latency and bandwidth characteristics from conventional DRAM, and are likely to address the capacity/compute scaling gap. These factors make it important to understand how the memory subsystem and its characteristics (latency, bandwidth) impact the performance of real-time big data analytics.

In this paper, we use straightforward analytic equations to quantify the impact of memory bandwidth and latency on performance. We leverage prior performance analyses for superscalar processors, and create equations whose components can be determined using performance counter data from real systems. Further, we use these components and the measured data to classify about a dozen workloads based on their inherent bandwidth demand and latency sensitivity, including big data workloads that utilize both structured and unstructured data. This classification enables us to create synthetic component values for the performance equations for each workload cluster or *class*. We use this to show performance sensitivity of a big data workload class to changes in memory bandwidth and latency and how it compares to the performance sensitivity for enterprise and HPC workload classes. The performance sensitivities show a continuum, with HPC workloads being most sensitive to changes in memory bandwidth, enterprise workloads showing the most sensitivity to memory latency, and big data workloads showing more modest sensitivity to both. Using this data, we are also able to establish an equivalence relationship between bandwidth increase and latency decrease for each workload class. Finally, we conclude with a discussion of how the model can be modified to account for multiple levels of memory hierarchy.

II. Related Work

There have been several papers discussing memory system characterization of conventional enterprise workloads over the past decade [1,2,3,4,5,6]. In this paper, we focus on big data workloads, and position them with regard to enterprise and high performance computing. Some of the big data usages that we cover include in-memory computing, search engines, and distributed processing frameworks.

Several studies have presented performance analysis of large-scale internet data center workloads, and include analyses

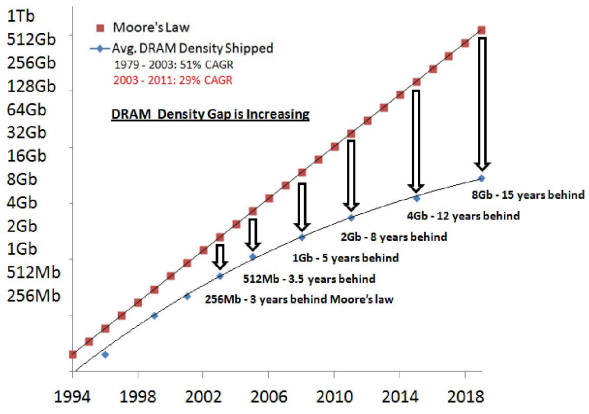

Figure 1: Trends in CPU and DRAM scaling [27].

978-1-5090-0088-3/15 $31.00 © 2015 IEEE
DOI 10.1109/IISWC.2015.32

IEEE computer society

of processor core design limitations [26], the impact of scheduling policies [24], and platform-level design implications [25]. While these studies consider similar workloads in some cases, they do not examine the impact on workload performance when varying the parameters of the memory subsystem.

Previous researchers have also characterized big data workloads. Ren et al. characterize the behavior of a production Hadoop cluster, using a specific case study [7]. Issa et al. [8] present power and performance characterization of Hadoop. Yang et al. [9] propose using statistics-based techniques for modeling map reduce. Basu et al. [11] focus on page-table and virtual memory related optimizations for big data workloads. Jia et al. [12] present a characterization of L1, L2, and L3 cache misses observed for a Hadoop workload cluster. Dimitrov et al. [13] characterize memory usages of Hadoop and NoSQL workloads. However, all these studies focus on specific optimizations or characterizations, and stop short of providing a quantitative framework.

Previous researchers have also extensively addressed performance modeling. Karkhanis et al. [22] provide a means to estimate performance of superscalar processors based on data and instruction cache miss rates for SPEC benchmarks. They provide a detailed mechanism to model microarchitecture. Fields et al. [21] provide a means to understand the cost of an event by means of its interaction, i.e. quantifying the ability of the microarchitecture to mask the cost by performing operations in parallel. Using this mechanism, they provide an aggregator to break down the CPI into subcomponents. Sorin et al. [20] present an analytical model for evaluating various types of architectural alternatives for shared-memory systems, exploiting instruction level parallelism. Emma [10] provides an analytical framework for estimating performance using CPI, breaking it down into both infinite cache and memory subsystem effects. This approach was further extended by Chou [23] to account for the impact of *memory-level parallelism* (MLP). We use Chou's model, but change it slightly to correlate some model components with measurements made with hardware performance counters.

III. Workloads

We select real-world workloads on commercial software that process both structured and unstructured data in real-time for our analysis.

A. Big Data

1) In-Memory Column Stores

As data volumes continue to grow, and the demand for real-time analytics increases, traditional relational databases are shifting to in-memory columnar usages. Holding entire datasets in memory avoids the higher and unpredictable latencies of storage subsystems. Further, columnar storage allows the data to be compressed very easily as data that belong to a column are of the same data type; resulting in 10x or more reduction in size with dictionary compression and bit-packing. In this paper, we use an in-memory columnar database to query a structured dataset of several 100GB, with decision support analytical queries. We use the term 'structured data' to refer to the 'column stores' workload in this paper, and use the two terms interchangeably.

2) Needle In The hayStack (NITS) Search

Big data has brought forth a focus on extracting information from unstructured data such as logs, documents, records, etc. We use a commercial "search engine" to perform search on unstructured data. This type of search involves items that occur sparsely in a dataset. As a result, the application typically has to check the entire dataset for the occurrence of the search term. Probabilistic data structures such as cuckoo hashes or bloom filters are used to reduce the search space for the query in real-time. This type of search tends to be very data intensive as often large portions of the dataset need to be scanned in order to provide a result for the search queries.

3) Proximity Search

Proximity search, or dense search involves a query where there is a proximity metric that binds the elements of the search interval, and this greatly reduces the search space when the query is executed. Thus, the proximity metric enables a quick reduction in the search space. For example, the indexes could be organized by time and the proximity metric could be a specific time window. As a result of this reduction in search space by elimination, only a small portion of the dataset is touched and the performance of the search is extremely core bound.

4) Spark In-Memory Computing

We use a graph analysis workload based on the Apache Spark in-memory distributed computing framework. The solution uses an iterative graph-parallel algorithm to compute associations between two vertices that are n-hop away (e.g., friend to friend, or similarities between videos for recommendation).

B. Enterprise Workloads

We use real-world commercial enterprise software in our experiments, together with industry standard benchmarks. We are unable to mention the software vendors due to confidentiality requirements and the names of the benchmarks due to auditing requirements/run-rule reporting criteria for SPEC and TPC benchmarks. In each case, the workload is multithreaded, using all logical processors in the system.

(1) Online Transaction Processing ("OLTP"): We use an industry standard OLTP workload on a commercial database in order to represent enterprise transaction processing. The workload models transactions that are processed in a brokerage firm, with customers generating transactions related to trades, account inquiries, and market research against the database. The brokerage firm interacts with financial markets to execute orders on behalf of the customers.

(2) Java processing ("JVM"): We use an industry-standard java workload, to reflect real-world XML processing and BigDecimal computations, emulating the middle tier in a three-tier client/database server system, and exercising a Java Virtual Machine with JIT compilation, garbage collection, etc.

(3) Virtualized processing ("virtualization"): We used an industry-standard virtualization workload that tests datacenter servers used in virtualized server consolidation. The consolidated enterprise usages include mail servers, application servers, and web servers.

(4) Web tier caching ("web caching"): We use a web tier memcached-like workload to represent caching between the web and data tiers in an enterprise. The open source code was modified to reduce lock contention and improve scalability.

C. High Performance Computing Workloads

The SPEC CPU2006 benchmarks represent real-world problems of R&D environments or a highly specialized field such as weather forecasting. Some of the more common applications represented in this suite include ray tracing, PERL interpretation, data compression, video compression and XML processing. To represent HPC usages, we select the following workloads from the SPECfp benchmark suite: "milc", "soplex", "bwaves", and "wrf". These components exhibit high memory bandwidth demand when compiled with the Intel version 14 compiler. We use the "rate" configuration of the benchmark components, where each logical processor is executing a copy of the program.

D. Memory Traffic Characterization

As a reference point, we include micro-benchmarks for memory latency and bandwidth in our list of workloads. In particular, we used Intel® Memory Latency Checker [19], which is a tool used to generate traffic to the memory subsystem and measure cache and memory latency and bandwidth. We use this to calibrate the relationship between loaded latency and bandwidth.

IV. Methodology

A. Cycles per Instruction and Pathlength

We use the number of clock *cycles per instruction* (CPI) as the measure of processor performance, as recommended in [10]. CPI is the inverse of instruction throughput, and a lower value of CPI indicates better performance. We define the *pathlength* of a workload as the required number of instructions to complete a unit of work. Thus the pathlength depends on the software code path that is executed in order to complete such a unit of work. The combination of pathlength and CPI then determines the processor-limited performance of the workload, whether it is quantified as run time or throughput. Thus, if the pathlength is fixed, the CPI per processor and the number of processors can directly be converted into a workload measure of throughput.

Table 1: List of symbols and their definitions.

Symbol	Definition
CPI	Cycles per instruction
PL	Pathlength
CPI_{eff}	Effective CPI
CPI_{cache}	CPI with an infinite cache
MPI	Misses per instruction
MPKI	Misses per 1000 instructions
MP	Miss penalty, in cycles
BF	Blocking factor
MLP	Memory-level parallelism
BW	Bandwidth
WBR	Writeback rate @ cache line evictions
LS	Line size
IOPI	I/O events per instruction
IOSZ	Average size of memory read per I/O event
CPS	Cycles per second (Core Speed)
HPC	High performance computing
NITS	Needle in the haystack queries

To simplify our analysis, we will consider the case of a fixed pathlength below. We have found this to be a reasonable assumption in most cases where software is well-tuned with little synchronization overhead and is not IO-limited. It is possible, however, to build more sophisticated models including calculations that compute pathlength as a function of other system parameters, such as memory capacity, the problem or database size, the number of threads (hardware or software), etc. For the workloads and configurations we consider in this study, there is little or no variation in pathlength that occurs. Also, since we are focused on the performance impact of the memory subsystem for fixed workload and core behavior, this approach is reasonable.

B. Relating CPI to Memory Behavior

The first equation of our model assumes that instruction throughput is sensitive to and limited by memory latency, a characteristic often described as *memory bound*. The other possible extremes of processor-limited application behavior, *core bound*, where the performance is solely limited by the processor issue width and functional units, or *bandwidth bound*, where the performance is limited by the maximum amount of memory bandwidth the system can deliver, may also be represented by this model. This is explained further below.

Let's assume we have a processor with a two-level memory hierarchy; one level of cache followed by main memory. $\boldsymbol{CPI_{eff}}$ stands for Effective Cycles per Instruction and denotes the inverse of the effective rate of instruction execution. Eq. 1 shows our model for a single-thread of execution in its simplest form.

$$\boldsymbol{CPI_{eff} = CPI_{cache} + MPI * MP * BF} \quad [1]$$

The terms used to compute this value are as follows: $\boldsymbol{CPI_{cache}}$ is the CPI if all memory references were satisfied by the processor cache, $\boldsymbol{MPI}$ are the misses per instruction at the processor cache (either demand or prefetch), $\boldsymbol{MP}$ is the cache miss penalty, and $\boldsymbol{BF}$ is the "blocking factor", or percentage of the miss penalty that contributes to $\boldsymbol{CPI_{eff}}$ Thus, the equation takes as input the effective CPI with an infinite cache, and adds incremental CPI based on the number of cache misses, the latency to satisfy those misses, and a factor that reflects the impact of that latency. If an application is truly core bound, it will have no sensitivity to memory latency, and the blocking factor in our equation will be zero. This is possible if prefetching ensures that data is readily available in the processor cache once the core requires it. On the other hand, if the workload is bandwidth bound, the equation should still hold, as queuing theory informs us that latency increases dramatically as available bandwidth becomes saturated. This model can be extended in a straightforward manner to account for multiple levels of memory hierarchy and bandwidth limitations, as discussed further below in Sec. VII.

Our Eq. 1 is consistent with the observations made in [22,23], which states that the core stall time resulting from multiple simultaneous outstanding long latency cache misses, call this value $\boldsymbol{MLP}$, is derived from the miss penalty $\boldsymbol{MP}$

divided by the $\boldsymbol{MLP}$. Eq. 2 is taken from [23] and shows this explicitly:

$$\boldsymbol{CPI_{eff} = CPI_{cache} * (1 - Overlap_{CM}) + \frac{MPI * MP}{MLP}} \quad [2]$$

By setting the right hand side of Equations 1 and 2 to be equal and solving for $\boldsymbol{BF}$, we obtain Eq. 3:

$$\boldsymbol{BF = \frac{1}{MLP} - \frac{CPI_{cache} * Overlap_{CM}}{MPI * MP}} \quad [3]$$

Our $\boldsymbol{BF}$ then, is proportional to $\boldsymbol{1/MLP}$, but offset by $\boldsymbol{Overlap_{CM}}$, which represents the overlap of core execution with cache misses, and is computed as a fraction of the average miss penalty. We choose to use Eq. 1, since $\boldsymbol{Overlap_{CM}}$ and $\boldsymbol{MLP}$ are not directly measureable from hardware performance counters. In Sec. V below, we show how to estimate the $\boldsymbol{BF}$ using measured data.

In Eq. 3, we can see that the blocking factor is approximately equal to the reciprocal of the memory-level parallelism, but is also a function of the misses per instruction and miss penalty as shown in the second term. However, in our model, we assume that $\boldsymbol{BF}$ is constant, as we expect the second term in Eq. 3 to be small when compared to $\boldsymbol{1/MLP}$, and it will tend toward zero as miss penalty increases

For clarity, we should also note that the $\boldsymbol{CPI_{cache}}$ is not a CPI with zero stall cycles. As noted in [10], it is possible for data dependencies and serialization events to cause core stalls even when executing with an infinite cache. Further, in our application of the model, we assume an infinite third-level or last-level cache (LLC), meaning delays in accessing data in the LLC due to misses in the L1 or L2 caches can cause stall cycles. While it is straightforward to establish a $\boldsymbol{CPI_{cache}}$ through simulation of the core with an instruction trace, we show how to estimate this value using measured data from performance counters in Sec. V below.

C. Modeling Bandwidth

Once we have computed the effective CPI for a given program phase using Eq. 1, we can use Eq. 4 to compute the bandwidth demand in bytes per second on the memory subsystem. In Eq. 4, $\boldsymbol{WBR}$ is the percentage of cache misses that require writeback of a dirty victim and $\boldsymbol{LS}$ is the line size in bytes. I/O references also result in memory reads and writes. Here, $\boldsymbol{IOPI}$ is the rate of I/O events per instruction, $\boldsymbol{IOSZ}$ is the average size of a memory read or write per I/O event. $\boldsymbol{CPS}$ is the core speed in cycles per second, and combined with $\boldsymbol{CPI_{eff}}$ converts per instruction event rates into bandwidth.

$$\boldsymbol{BW = \frac{\left(MPI*(1+WBR)*LS + (IOPI*IOSZ)\right)*CPS}{CPI_{eff}}} \quad [4]$$

By scaling Eq. 4 with total core count (or hardware thread count in the case of multithreaded processors), we can compute the total system-wide bandwidth required. Using this value, we can estimate the average $\boldsymbol{MP}$ using queuing theory or characterization data that provides a relationship between queuing delay and utilization. Further, by changing the denominator in Eq. 4 to the available memory bandwidth, we can compute a *bandwidth-limited* CPI.

D. Applicability of the Model

In many cases, workloads will exhibit distinct phases of execution, and these phases will have different values for $\boldsymbol{CPI_{cache}}$ and $\boldsymbol{MPI}$, which in turn affect bandwidth demand and observed $\boldsymbol{MP}$. However, we have found that many workloads exhibit sustained periods of steady-state behavior, where the inputs to our equation vary over a range with a certain periodicity. In cases where bandwidth demand does not reach capacity when measured at 100ms to 200ms intervals, we find that this steady-state periodic behavior can be represented accurately with our model. Additionally, we can apply our model to multiple program phases independently when the constraints described above do not apply, provided we are able to apply a weight to each phase based on the relative number of instructions contained in that phase. Also, if a workload's performance is limited by synchronization bottlenecks or available bandwidth from I/O devices, CPI will not provide a good proxy for performance. As stated earlier, we do not observe these effects in the workloads we studied, and they can typically be overcome with tuning and/or device configuration changes.

V. Building the Model

A. Determining Component Values

The goal of our performance modeling methodology is to determine how configuration of the memory subsystem impacts performance for certain types of core behaviors. In order to do this, we need to determine the equivalent of $\boldsymbol{CPI_{cache}}$ and the $\boldsymbol{BF}$ in Eq. 1 for a given workload.

We can estimate these parameters by measuring a workload's $\boldsymbol{CPI_{eff}}$ at different miss penalties. This is achieved by varying the core speed and memory speed of the system under test. When we reduce the frequency at which the core executes (called frequency scaling), we slow down the core speed while keeping the memory speed the same. This makes the memory seem faster relative to the core when miss penalty is measured in core cycles.

Alternatively, when we reduce the speed at which the memory subsystem operates, the penalty for a miss measured in core cycles is increased due to memory speed being reduced. As a result of these variations, we get data points with different ($\boldsymbol{MPI * MP}$), and we measure $\boldsymbol{CPI_{eff}}$ for each of these datapoints using hardware performance counters. We estimate $\boldsymbol{CPI_{cache}}$ and $\boldsymbol{BF}$ in Eq. 1 by obtaining a fit for these data points.

B. Setup

In order to evaluate the model, we used the workloads described in Sec. III. These experiments were conducted on up to four enterprise servers with Intel® Xeon® E5-2600 series processors with dual-socket configurations. Each socket has eight physical cores (16 logical cores or hardware threads with Intel® Hyper-Threading technology), and 2.5MB of Last Level Cache (LLC) per core. The systems were populated with up to 256GB of DDR3 memory and Solid State Drives (SSDs) for storage. We varied the core and memory speeds using operating system governors, BIOS knobs, and Machine Status Registers (MSRs). Measurements of $\boldsymbol{CPI_{eff}}$, $\boldsymbol{MPI}$, and $\boldsymbol{MP}$ were recorded using performance counters. We verified during our experiments that there were very little or no run-to-run

variations in pathlength for each of the workloads, thereby validating our assumptions in factoring out pathlength in our analysis.

C. Modeling Analytics on Column Stores

The measured data in Fig. 2 shows that processing structured data is heavy on processor utilization (close to 100%), and the vast majority of CPI samples are within a narrow range given the ~100ms sample rate. There is a lot of memory traffic, as seen by the high memory bandwidth. We do not expect any CPI sample to be bandwidth-limited based on the memory bandwidth chart. Using the methodology described in Section V.A, we measure the $\boldsymbol{CPI_{eff}}$ at different average miss penalties per instruction ($\boldsymbol{MPI * MP}$) and obtain the curve fit that is shown in Fig. 3(a). It can be seen that there is a very good fit (R^2=0.95) in $\boldsymbol{BF}$ for a fairly large spread of latency per instruction.

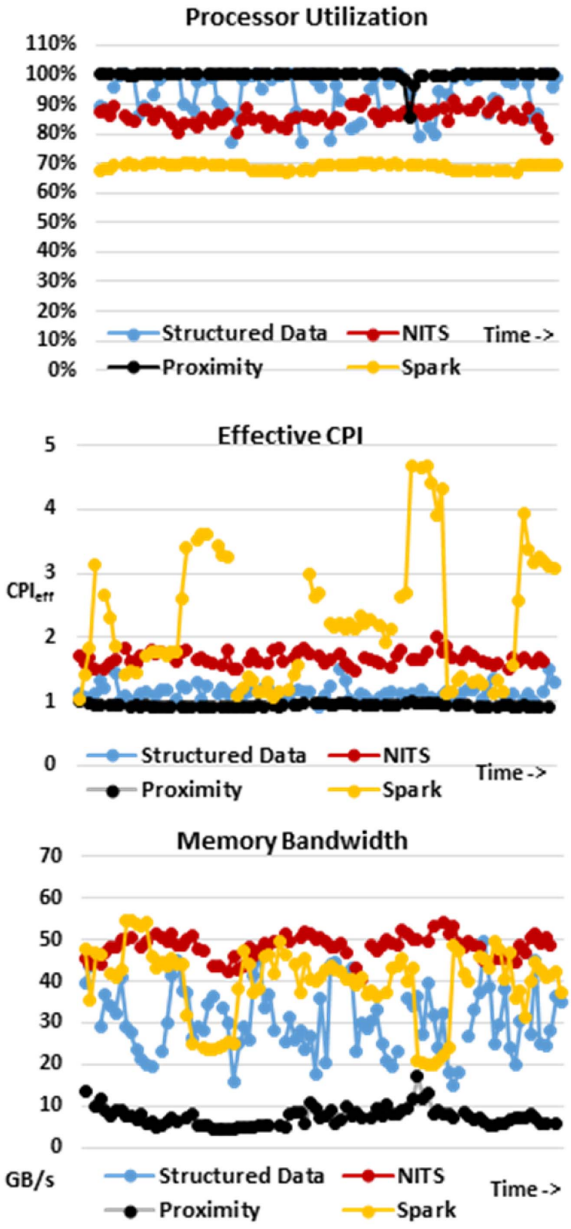

Figure 2: Measured CPU Utilization, CPI, and Memory Bandwidth vs. Time for big data workloads.

D. Modeling NITS Queries

Compared to in-memory structured analytics processing, processing NITS queries on unstructured data tends to be very I/O intensive. This is because NITS queries tend to scan almost the entire dataset. While running these queries, we observed I/O rates of more than 2GB/s, which stressed our high-speed RAID configuration of 4 enterprise-class SSDs. As a result of this high I/O activity, approximately half of the CPU usage is spent in system-time. However, we should note that even at these levels, the I/O bandwidth is still relatively small when compared to the total memory bandwidth, as shown in Fig. 2. The curve fit for $\boldsymbol{CPI_{eff}}$ obtained by varying core and memory speeds is shown in Fig. 3(a).

E. Modeling Proximity Search

Proximity queries tend to scan only small portions of the dataset since the search space is reduced due to the proximity factor. As a result, while processing these queries, a lot of time

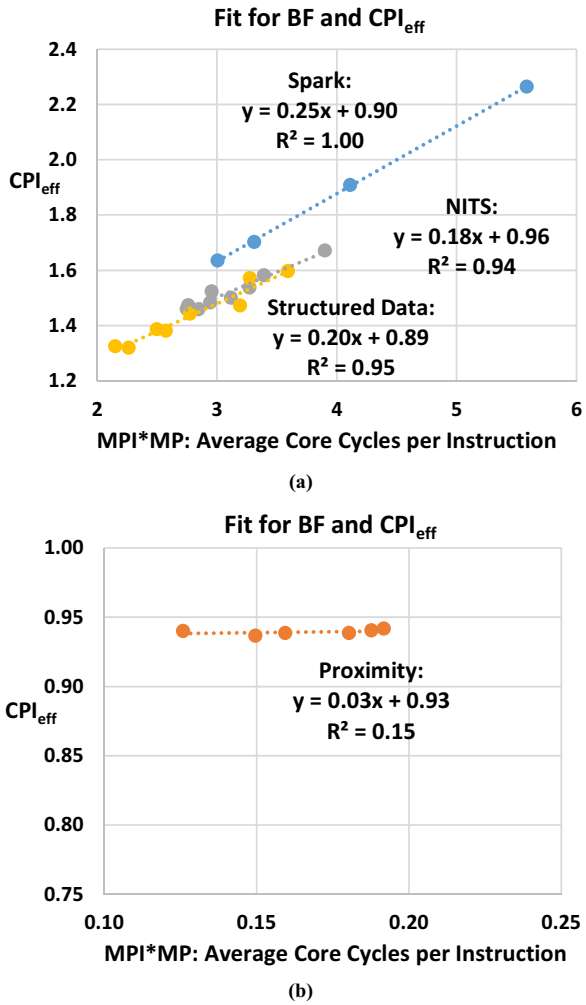

Figure 3: CPI vs. total latency for big data workloads.

is spent uncompressing the dataset, and comparing the results. Thus the operations tend to fully utilize the processors, as observed in Fig. 2. We collected data at various core speeds, and obtained the curve fit for $\boldsymbol{CPI_{eff}}$ shown in Fig. 3 (b). The very low value of the blocking factor indicates that the workload is strongly core-bound, and not sensitive to memory latency. Combined with an $\boldsymbol{MPI}$ that is an order of magnitude lower than the other workloads (as observed by the lower memory bandwidth in Fig. 2) we can expect almost no sensitivity to memory latency for this workload. The poor correlation coefficient is not of concern in this case, due to the small variance in measured CPI and extremely low blocking factor.

F. Modeling Spark In-Memory Computing

The distributed in-memory analytics workload uses the Spark framework to perform iterative graph computations. The unit of progress can be differentiated into various jobs. For our analysis, we create the CPI model for one of the jobs. Fig. 2 shows that the CPU utilization is around 70%, limited by the dynamic thread-level parallelism, and there is a lot of variation in CPI. It can be observed that the memory bandwidth reaches upwards of 50GB/s – not exceeding the total available for our setup. Although the CPI is variable, we still obtain a good curve fit for $\boldsymbol{CPI_{eff}}$ as shown in Fig. 3(a). The $\boldsymbol{BF}$ is observed to be higher than the previous scenarios, indicating the performance of this workload is more sensitive to memory latency.

G. Big Data Summary

Tab. 2 shows a summary of $\boldsymbol{CPI_{cache}}$ and $\boldsymbol{BF}$ for each of the big data workloads. The table also provides the MPKI (LLC misses per 1000 instructions) and writeback rate ($\boldsymbol{WBR}$). These components determine the rate per instruction of memory reads and writes. We express the $\boldsymbol{WBR}$ as a percentage of MPKI. The percentage for NITS exceeds 100% as there are non-temporal writes in the workload.

H. Validation

We can validate the model by showing it computes a $\boldsymbol{CPI_{eff}}$ that matches our measurements. By virtue of the high correlation factors for the blocking factor regression, we expect a low amount of error in this CPI calculation. Tab. 3 shows the detailed measurements we made to construct Fig. 3(a), taking multiple measurements at each core speed to account for any run-to-run variation. Using the values for $\boldsymbol{CPI_{cache}}$ and $\boldsymbol{BF}$ determined in Fig. 3(a), we compute the $\boldsymbol{CPI_{eff}}$ and compare it to the measured value. The last row of Tab. 3 shows the error between the measured CPI for Structured Data and the computed CPI using our model. For the other three workloads, the error in computed $\boldsymbol{CPI_{eff}}$ is less than +/-2%.

Table 2: Workload parameters for big data.

Workload	CPI_{cache}	BF	MPKI	WBR
Structured Data	0.89	0.20	5.6	32%
NITS	0.96	0.18	5.0	179%
Spark	0.90	0.25	6.0	64%
Proximity	0.93	0.03	0.5	47%

I. Enterprise and HPC Benchmark Modeling

Figs. 4 and 5 show system-level characterization data for the enterprise and HPC proxy workloads. All benchmarks were run on Intel® Xeon® E5-2600 series processors with dual-socket configurations like the big data workloads, except the number of cores used was varied depending on the benchmark as detailed below. It is important to keep in mind that the memory bandwidth measurements correspond to different core counts across the benchmarks. In both figures, the data was collected during steady-state behavior after varying amounts of warm-up time. For Fig. 4, the data was collected for about 5 minutes of run time with a sampling granularity of about 100ms. For Fig. 5, the data was collected for about 7 minutes with a 1 second sampling granularity.

J. OLTP

The OLTP benchmark was run on a dual-socket Xeon platform, where each socket has twelve physical cores (24 logical cores or hardware threads with Intel® Hyper-Threading technology), and 2.5MB of Last Level Cache (LLC) per core. The OLTP system was populated with 256GB of memory 56 Solid State Drives (SSDs) for storage, and used two 1 GbE network interfaces.

The benchmark represents a client/server OLTP system with many concurrent users accessing a relational database. As such, this is a structured data usage model. With many concurrent users, the Database Management System (DBMS) must implement concurrency control to protect the integrity of the data. As noted by other researchers, e.g. [28], this can result in variable pathlength and variable CPI due to synchronization bottlenecks, system idle time, and varying I/O rates. Our characterization was done on a well-tuned system that avoided synchronization bottlenecks and with a memory capacity that enabled modest I/O rates. Further, while idle time is minimized for these measurements, we should note that idle time on these systems puts the processor in a halt state that does not include spinning as observed in [28], and thus the CPI is not diluted by this activity. While there is some variability in average pathlength per transaction, $\boldsymbol{CPI_{cache}}$ and $\boldsymbol{BF}$ for varying core counts and memory capacities, it is very small for the large fixed memory capacity and the modest fixed number of cores we are

Table 3: Computed versus measured CPI for Structured Data

Core Speed (GHz)	2.1	2.4	2.7	3.1	2.1	2.4	2.7	3.1
BF	.20	.20	.20	.20	.20	.20	.20	.20
CPI_{cache}	.89	.89	.89	.89	.89	.89	.89	.89
MPI	0.0056	0.0056	0.0059	0.0057	0.0056	0.0056	0.0055	0.0055
MP (core cycles)	402	462	543	631	383	448	502	598
CPI (computed)	1.33	1.39	1.52	1.60	1.31	1.38	1.43	1.53
CPI (measured)	1.32	1.38	1.47	1.60	1.32	1.39	1.44	1.57
Error	1.0%	0.9%	3.0%	-0.1%	-1.1%	-0.6%	-0.6%	-2.5%

modeling. This makes the constant values we are using reasonable for our first-order model.

K. JVM

The JVM benchmark, in contrast to OLTP, does very little I/O which has almost no impact on performance. It also has a small sensitivity to memory capacity. In order to increase memory reference locality in multi-socket systems, the benchmark is run with multiple JVMs, in our case one per socket. The frequency scaling tests we used to establish $\boldsymbol{CPI_{cache}}$ and $\boldsymbol{BF}$ for this benchmark also used Intel® Xeon® E5-2600 series processors but with six physical cores (12 hardware threads) per socket. Our measurements varying core count and frequency show a nearly constant pathlength per unit of work for this benchmark.

L. Virtualization

The virtualization benchmark was run on a 2-socket platform with twelve physical cores (24 hardware threads) per socket. The system was configured with 512GB of memory, 34 SSDs, and one dual-port 10GbE network interface card. This benchmark also used a commercially available Hypervisor. Our frequency and core scaling tests showed an almost constant pathlength per unit of throughput.

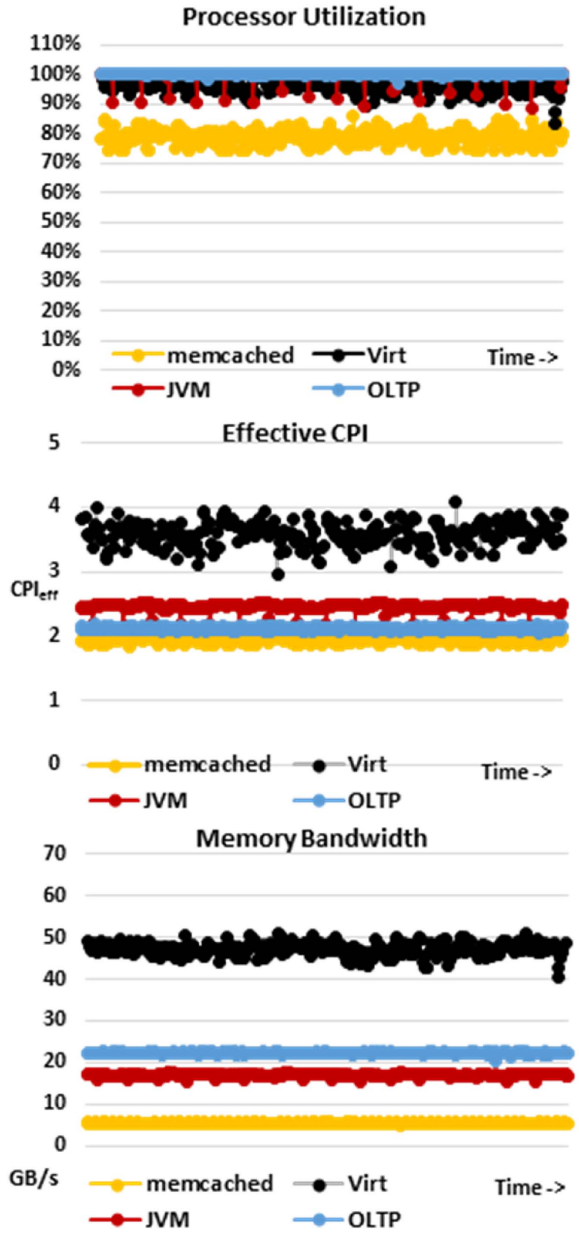

Figure 4: Measured CPU Utilization, CPI, and Memory Bandwidth vs. Time for enterprise workloads.

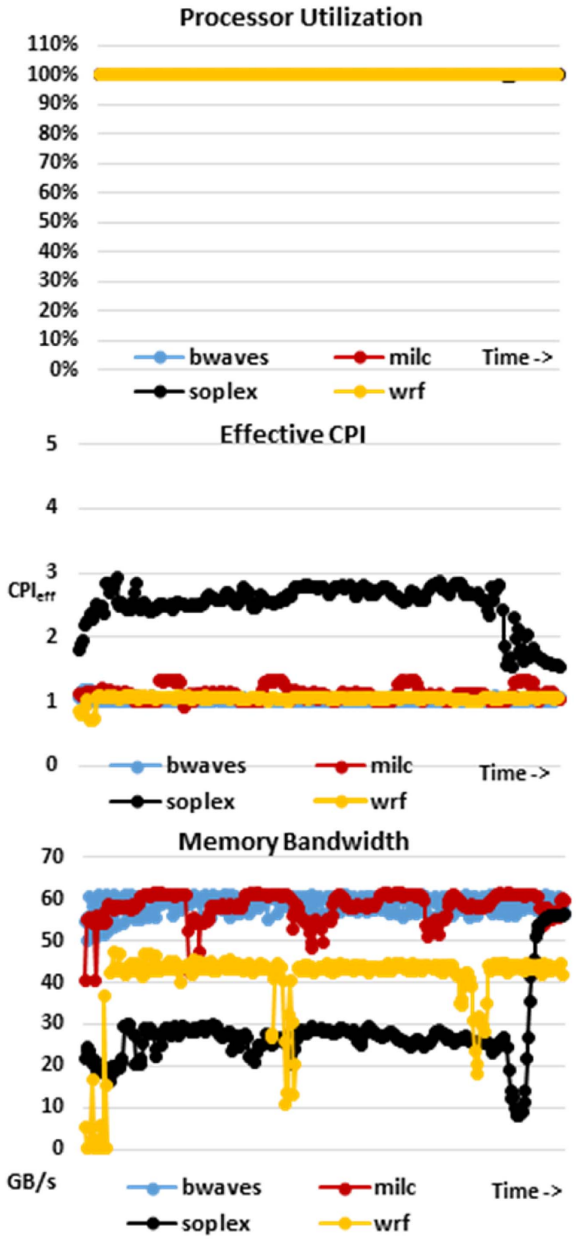

Figure 5: Measured CPU Utilization, CPI, and Memory Bandwidth vs. Time for HPC proxy workloads.

M. Web Caching

We used a modified version of memcached to represent an enterprise usage in our study, as there is no analytical processing component in our example. We used a test harness that makes requests to objects known to be present in the memcached server. For our tests, requests were to 64B sized objects randomly distributed across the database which was wholly contained in memory. Frequency scaling tests were run on two-socket platforms with eight processor cores per socket and two 10GbE network interface cards.

This workload has lower processor utilization than the other enterprise benchmarks. This is due to the configuration, which only allocated one-half of the virtual processors to the application. This left the other half available for network packet processing, but they were not fully utilized. Our measurements showed a nearly constant pathlength per query as all requested objects were located in the memory-resident database.

N. SPEC CPU Rate FP Componenets

Our HPC proxy workloads were taken from the SPEC CPU Floating Point suite as described earlier. In order to get good frequency scaling measurements so that we could apply Eq. 1, we used only three processor cores or six hardware threads per socket. This configuration is not memory-bandwidth limited for these components, and enabled us to get measurements that fit the latency-limited model. In all cases, one copy of the program was run for each virtual processor used. As is always the case for SPEC CPU rate, all program copies are independent and do not share data or synchronize, and there is virtually no I/O for the vast majority of the run time. As expected, the pathlength for each component benchmark is nearly constant per copy.

O. Enterprise and HPC Summary

We applied our methodology as described in Sec. V.A above using the results of frequency scaling tests to estimate the $\boldsymbol{CPI_{cache}}$ and $\boldsymbol{BF}$ for each enterprise benchmark and SPEC CPU floating point component. The summary results for these regressions are shown in Tabs. 4 and 5.

VI. APPLYING THE MODEL

A. Enterprise and HPC Workloads

We compare $\boldsymbol{CPI_{cache}}$ and the $\boldsymbol{BF}$ obtained using our methodology with various enterprise and high performance computing workloads. This helps us position how big data, as a workload class, compares to the other classes. As described earlier, for enterprise, we use workloads representing the following usage models: *virtualization*, *web caching, OLTP*, and *JVM*. For high performance computing, we use *bwaves, milc, soplex* and *wrf* from the SPECfp suite.

As shown in Tab. 4 for the enterprise workloads, on average, it can be observed that the blocking factor is relatively high, likely due to the ineffectiveness of prefetching and branch prediction. This observation is consistent with the high memory stall times reported in [26].

The workload parameters for the HPC workloads are shown in Tab. 5. For high performance computing workloads, on average, the blocking factor is observed to be low. These usages typically involve numerical simulations and analysis that involve large amounts of data and instruction-level parallelism.

Table 4: Workload parameters for enterprise.

Workload	CPI_{cache}	BF	MPKI	WBR
Virtualization	1.77	0.39	15.1	21%
Web Caching	1.75	0.46	1.4	21%
OLTP	1.38	0.34	3.4	32%
JVM	0.97	0.45	7.0	35%

Table 5: Workload parameters for HPC.

Workload	CPI_{cache}	BF	MPKI	WBR
bwaves	0.61	0.04	26.8	10%
milc	0.67	0.05	23.0	35%
soplex	0.98	0.12	39.5	28%
wrf	0.72	0.07	17.3	37%

The data access is also regular, making prefetching highly effective. The low blocking factor indicates that the CPI is not heavily dependent on the latency of memory references, and these workloads have sufficient parallelism to mask the same.

B. Creating Workload Classes

Using a variation of Eq. 4, we compute the intrinsic memory read and write demand per core for the various workloads. We can do this by removing the $\boldsymbol{CPS}$, $\boldsymbol{LS}$ and the I/O related terms, and substituting $\boldsymbol{CPI_{cache}}$ for $\boldsymbol{CPI_{eff}}$ in Eq. 4. Fig. 6 shows a graphical view of latency sensitivity and the memory read and write demand for all of the workloads. As discussed earlier, a higher value of the blocking factor indicates more sensitivity to memory latency, and we show this on the x-axis. The y-axis is a measure of the number of memory references per cycle for the workload shown as reads and writebacks per cycle, which determines the bandwidth required when $\boldsymbol{CPI_{eff}} = \boldsymbol{CPI_{cache}}$. Therefore a higher value along the y-axis indicates more sensitivity to bandwidth.

We compute a "mean" for these workloads by averaging the bandwidth demand and blocking factor within each usage class. These "means" for enterprise, big data, and HPC are shown in

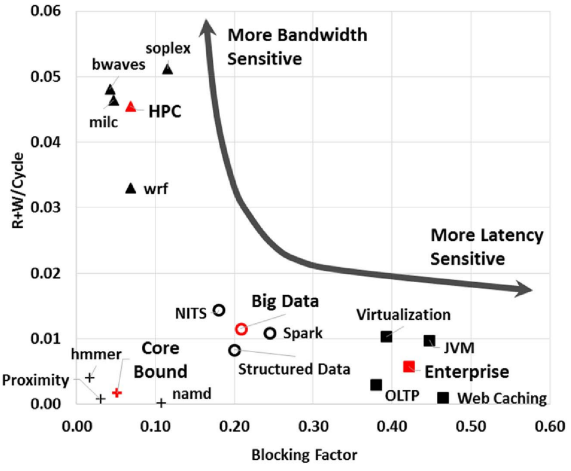

Figure 6: Bandwidth demand vs. latency sensitivity for the various workloads across different segments.

Table 6: Workload class parameters.

Workload Class	CPI_{cache}	BF	MPKI	WBR
Enterprise	1.47	0.41	6.7	27%
Big Data	0.91	0.21	5.5	92%
HPC	0.75	0.07	26.7	27%

red in the figure. Our classification enables us to clearly distinguish between the workload classes in the figure, as each workload class forms its own distinct cluster. The enterprise workload class has the most sensitivity to latency, and least sensitivity to bandwidth. The high performance computing workload class, on the other hand, has the most sensitivity to bandwidth and the least sensitivity to latency. Big data, as a workload class, falls in the middle, and is intermediate in sensitivity to both bandwidth and latency. The mean for each workload class is shown in Tab. 6. At this point, we omit the core-bound Proximity query workload from our analysis, as it will not show any sensitivity to memory latency or bandwidth. We have also observed that some components of the SPEC CPU suite also exhibit this characteristic, and they are shown in Figure 6. This group is shown as another cluster of workloads centered near the origin on our chart.

C. Impact of Memory Subsystem

Using our methodology, we can analyze the performance implications for future memory systems that have different latency and bandwidth characteristics from today's DRAM. In order to demonstrate the capability, we can make simple adjustments to the available bandwidth and memory latency for our test platform using our mathematical model, and analyze the resulting impact on CPI for each workload class.

1) Queuing Delay and Effective Utilization

With our workload classes characterized, our model computes the effective CPI based on the miss penalty. In order to compute the CPI for different system configurations, we need to determine the miss penalty for these alternatives. First, we break the miss penalty down into two components – the *compulsory* or *unloaded latency* and the *queuing delay*. The sum of the two, the miss penalty, can also be referred to as the *loaded latency*. In applying our model, we can select different values for the compulsory memory latency, and determine the queuing delay by comparing the bandwidth demand to the available bandwidth in our target system. We use an iterative calculation to find a stable solution for queuing delay vs. bandwidth demand. Thus, we can build a model that enables us to change the compulsory latency and available bandwidth, and determine the impact on CPI.

Fig. 7 shows measured data for memory channel queuing delay vs. bandwidth utilization for four different combinations of memory speed and read/write mix. This data was obtained by executing the Intel® Memory Latency Checker that generates memory requests on multiple cores to randomly distributed addresses in the memory space at different arrival rates, and collecting performance counter data as it runs. We collected this data for two different read/write mixes and two different memory speeds on our test platform. In order to

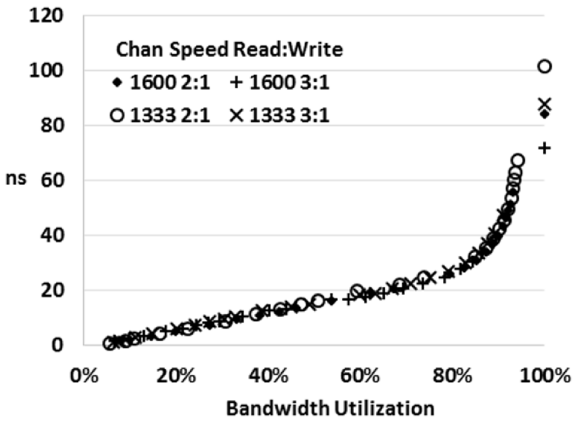

Figure 7: Memory queuing delay vs. utilization.

compute the queuing delay, we can subtract the minimum observed latency for each test case (the compulsory latency) from the total latency observed at the different levels of bandwidth utilization. Further, we can establish the maximum possible bandwidth consumption, or *efficiency*, for each case (which varies with channel speed) from the measured data. Using this value, we can establish a percentage of utilization, enabling us to combine the measurements from each test case.

As the figure shows, up to almost 95% bandwidth utilization, the queuing delay is very similar despite the read/write mix and DDR speed changes. In order to create a queuing latency vs. bandwidth relationship for our model, we average these curves to create a composite model. This enables us to vary memory channel count, memory channel speed, core count, etc. in our model across the multiple workload segments while using a single curve for queuing delay vs. bandwidth consumption. A more detailed model with specific queuing delay/bandwidth utilization relationships is possible, but we don't believe it is necessary in our context. There will be some higher amount of error in the area between 95% and 100% bandwidth utilization, but we compute the bandwidth-limited CPI in our model for this scenario (setting $\boldsymbol{BW}$ to system-available bandwidth and solving for $\boldsymbol{CPI_{eff}}$ in Eq. 4).

2) Impact of Bandwidth

With our ability to set different compulsory latencies and bandwidth capacities, and compute the miss penalty and CPI, we can now perform some simple experiments to compute the impact of bandwidth and latency changes on performance. In a simple example, we start with a single-socket system with an eight core processor, a 75ns compulsory memory latency, and four channels of DDR3-1867.[1,2] The effective bandwidth of our baseline, based on observed efficiency of about 70%, is ~42GB/s, or ~5.25GB/s per core. Next, we model variations of this baseline, including changes in channel speed, efficiency, and number of channels, and record the changes in effective CPI. Fig. 8 then shows how CPI for each workload class increases vs. the reduction in memory bandwidth normalized on a per core basis. As expected, the HPC class shows the most impact, while the enterprise class shows the least. Big data can

[1] For all of our workload models, Hyperthreading is enabled, creating 16 hardware threads or logical processors.

[2] With a single socket, we use miss penalty and memory latency interchangeably, as there are no remote socket memory references.

tolerate some bandwidth reduction, but does show significant impact when peak bandwidth is reduced by more than 2.5GB/s per core vs. our baseline.

The sensitivity to bandwidth is reflected in these workload scenarios in two different ways:

- If the workload is bandwidth bound, bandwidth demand exceeds supply: CPI will change based on the bandwidth supply change as calculated by Eq. 4.
- If the workload is memory bound, queuing delay will change based on bandwidth utilization: CPI will change due to the change in the miss penalty, moderated by the blocking factor, as shown in Eq. 1.

In Fig. 8 we show that it is possible to fit a curve to the CPI increase trend, in this case for HPC workloads. The increase is regular as all points on this curve are bandwidth bound. Big data would also likely show this trend in the bandwidth bound part of the curve, which in this case for the three points between -2.5GB/s and -3.5GB/s. On the other hand, the small and slowly growing performance impact on the enterprise workload class shows that this class is latency limited, with modest increases in queuing delay causing small increases in CPI.

Fig. 8 also clearly shows that the relationship between performance loss and bandwidth reduction is not linear for bandwidth-limited workloads. Thus, it is not possible to compute a simple constant "rule of thumb" that equates performance loss or gain with bandwidth reduction or increase. However, we can consider the performance loss and bandwidth reduction between the points on the curve, essentially computing the derivative of Fig. 8. We show this plot vs. the total memory bandwidth per core available in Fig. 9. This clearly shows that the performance impact of bandwidth reduction is based on the starting configuration.

3) Impact of Compulsory Latency

As an alternative to varying bandwidth supply and estimating the performance impact, we can also use our model to estimate performance while varying the compulsory part of the memory latency. Fig. 10 shows how CPI increases as we model compulsory latency increases in increments of 10ns versus our baseline of 75ns and four DDR channels of memory. As indicated by Fig. 10, the enterprise workload class shows the most latency sensitivity, followed by big data. Interestingly, the HPC workload class shows no latency sensitivity in our example. This is because the workload class model for HPC is bandwidth bound even with four DDR3-1867 channels. It is possible that increased latency can eventually make a bandwidth-bound workload become memory bound, but this does not occur in our example based on the combination of high bandwidth demand and low blocking factor for HPC.

As we did for bandwidth sensitivity in Fig. 9, we can also show the impact on performance for each increase in compulsory latency modeled by taking the difference between each pair of points on the x-axis of Fig. 10. Fig. 11 shows the impact on CPI for each 10ns increase in compulsory latency. As the chart shows, the impact is nearly constant, which is expected as the curves in Fig. 8 are nearly linear.

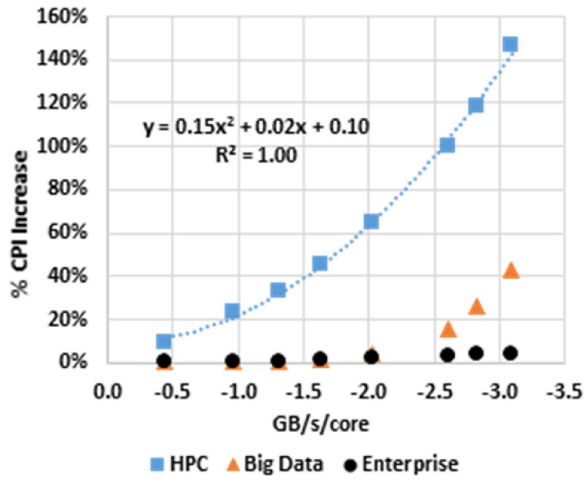

Figure 8: CPI increase vs. memory bandwidth decrease.

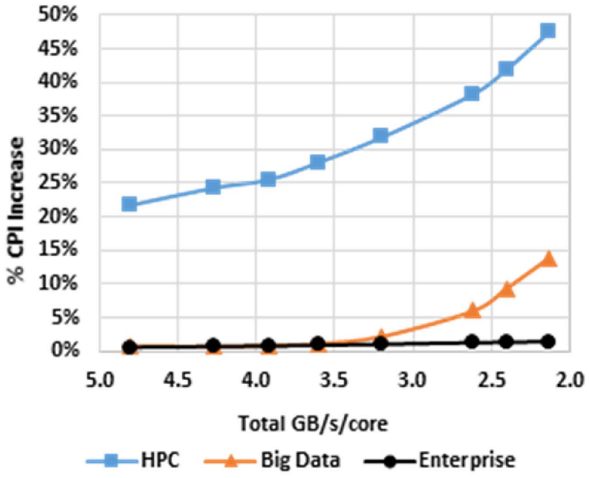

Figure 9: Rate of CPI increase vs. memory bandwidth.

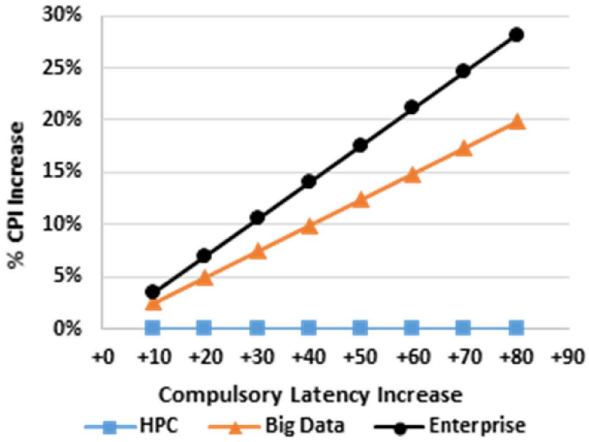

Figure 10: CPI increase vs. compulsory latency increase.

This occurs in our example as the bandwidth utilization in our baseline (four channels DDR3-1867) is fairly low for

enterprise and big data. Thus, the loaded latency is fairly close to the compulsory latency, and changes to the compulsory latency do not change the queuing delay significantly. HPC, on the other hand, is bandwidth bound for each compulsory latency point modeled, and the loaded latency is the compulsory latency plus the maximum stable queueing delay from Fig. 7. Based on Fig. 11, we can see in our example that the enterprise workload class is showing approximately 3.5% CPI increase for every 10ns increase in compulsory latency, while big data is showing about 2.5% increase. This is consistent with our expectations based on Fig. 6, where it is evident that enterprise and big data are the workload classes that are sensitive to memory latency.

D. *Design Tradeoffs*

The analysis provided in the previous section provides system architects an ability to understand the performance impact of high-level design choices for a broad range of workloads. If an architect has a choice between improving latency or bandwidth, which would be the better choice for performance? Tab. 7 presents this comparison by providing a summary of performance compared to our baseline for a difference of 8GB/s/socket (or 1GB/s/core) of bandwidth or 10ns of compulsory latency for the workload classes. For every 1GB/s/core increase in memory bandwidth, enterprise workloads and big data workloads have performance benefits of under 1%, and HPC workloads have benefits of about 24%. Likewise, every 10ns reduction in memory latency improves performance by about 3% for enterprise and big data, but HPC workloads see no performance improvement.

This can also be described by a statement of equivalence between latency and bandwidth. In our example, improving latency by 10ns gives the same performance benefit, on average, as 39.7GB/s improvement in bandwidth for enterprise workloads, or 27.1GB/s improvement in bandwidth for big data, and as no improvement in bandwidth for HPC. Likewise, improving bandwidth by 8GB/s/socket (or 1GB/s/core), on average, is the same as 2.0ns latency reduction for enterprise workloads, and 2.9ns latency reduction for big data workloads.

Using our model and the constant blocking factor assumption, we see that no amount of latency reduction can compensate for bandwidth constraints for our HPC mix. With

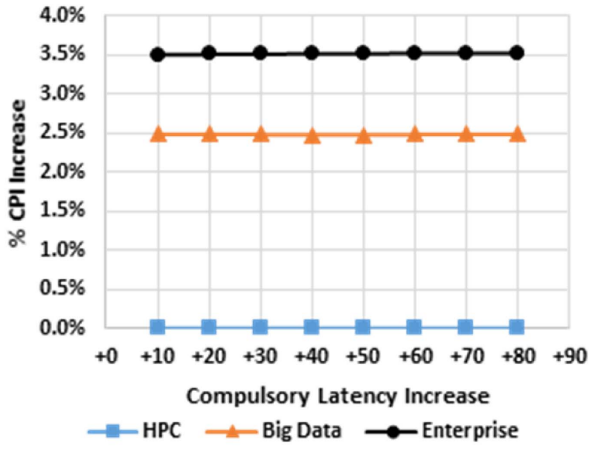

Figure 11: Rate of CPI increase per 10ns increase in compulsory latency.

Table 7: Latency-Bandwidth equivalence.

Class	8GB/s increase	10ns latency decrease	GB/s == 10ns	ns == 8GB/s
Enterprise	0.7%	3.5%	39.7 GB/s	2.0 ns
Big Data	0.7%	2.5%	27.1 GB/s	2.9 ns
HPC	24.3%	0.0%	zero	infinite

the performance equivalence from our model, it is easy to see that optimizing a design for latency reduction would be easier and more profitable for enterprise and big data workload classes, while optimizing for bandwidth increase would be the same for HPC.

Based on all data presented in our example, it should be clear that bandwidth-bound workloads respond with significant performance increase when additional bandwidth is provided. We see this is true for HPC and the big data workload classes in Fig. 9. On the other hand, when workloads are not bandwidth bound, they respond more to latency improvement than they do to bandwidth increase. We see this is true for enterprise and big data workload classes in Fig. 10 and Fig. 11. Further, the magnitude of performance sensitivity to latency increase for latency limited workloads is less than the impact of bandwidth increase for bandwidth-bound workloads – at least in the units of 1 GB/s/core versus 10ns. Ideally, system architects will create designs that provide sufficient bandwidth for target workloads before turning their attention to latency reduction. Further, if bandwidth-bound workloads are not the target for a particular design, cost savings can be achieved by reducing available bandwidth without significantly impacting performance.

VII. FUTURE MEMORY TECHNOLOGIES

Emerging memory technologies have different characteristics compared to DRAM: typically they have larger capacities, different power consumption, and lower cost, but also higher latencies and lower bandwidth. In order to take full advantage of the benefits of these technologies, it is critical to design the memory subsystem to mitigate the impact of the higher latency and lower bandwidth. The typical techniques used to hide memory latency are (1) predicting the next memory reference and prefetching onto a faster tier of memory, and (2) caching frequently used data in a faster tier of memory. To improve bandwidth, the techniques typically employed are adding additional buffers, or adding a faster, high bandwidth tier to the memory hierarchy.

As computer architects focus on new, tiered memory designs in order to accommodate these emerging technologies, it becomes important to understand the implications for different workloads. The analysis presented in this paper allows architects to estimate what the impacts of improving the latency or bandwidth may be. It is straightforward to extend the modeling methodology presented in this paper to account for hierarchical memories: for example, Eq. 1 would become:

$$\boldsymbol{CPI_{eff} = CPI_{cache} + (MPI_1 * MP_1 + MPI_2 * MP_2) * BF} \quad [5]$$

Here, $\boldsymbol{MPI_1}$ and $\boldsymbol{MP_1}$ correspond to the miss count and miss penalty for those requests satisfied by the first level of the memory hierarchy, and $\boldsymbol{MPI_2}$ and $\boldsymbol{MP_2}$ correspond to the miss

count and miss penalty for those requests satisfied by the second level of the memory hierarchy.

As another example, the modeling methodology could also be used to estimate the effectiveness of a prefetching technique by analyzing the variation in the blocking factor when using the technique. Since the blocking factor quantifies the ability of the system to hide the penalty of a memory reference, an improved prefetching technique will increase memory-level parallelism and will lower the blocking factor.

VIII. SUMMARY AND OUTLOOK

This paper characterizes the memory references of big data workloads, and positions them with respect to enterprise and HPC workloads. Further, it quantifies the impacts of increasing memory latency or reducing memory bandwidth for these workload classes. To summarize some of the observations and outlook:

1) Architects should provide enough bandwidth for their target workload class first, and then optimize for latency. The model provides a means to guide where one is positioned in the trade-off space.

2) Big data as a workload class fits squarely between more extreme cases of HPC and enterprise, except for core bound cases.

The model we have provided can be extended in a straightforward way to model additional memory architectures such as multi-socket, multi-level in hierarchy, and/or use of a new memory technology. We plan more specific, detailed analyses in our future work.

IX. ACKNOWLEDGEMENTS

The authors would like to acknowledge the contributions of many colleagues that enabled the creation of this paper. For the Spark workload characterization data, we would like to thank Liye Zhang and Jie Huang. Jian Chen and Sai Prashanth Muralidhara reviewed early versions of the architectural performance model. Thanks to Chitra Natarajan and Adrian Moga for collaboration on similar models for many years, and to Fayé Briggs for support and encouragement.

REFERENCES

[1] J.-S. Kim, X. Qin, and Y.Hsu, "Memory characterization of a parallel data mining workload," in *Workload Characterization: Methodology and Case Studies,* IEEE, pp. 60-68, 1999.

[2] M. Karlsson, K. Moore, E. Hagersten, and D. Wood, "Memory characterization of the ecperf benchmark," in *Second Annual Workshop on Memory Performance Issues (WMPI),* 2002.

[3] Y. Xu, E. Frachtenberg, S.Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol.99, p.1, 2013.

[4] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation--a pin-based memory characterization of the spec cpu2000 and speccpu2006 benchmark suites," VSSAD Technical Report, Intel Corporation, 2007.

[5] F. Zeng, L. Qiao, M. Liu, and Z. Tang, "Memory performance characterization of spec cpu2006 benchmarks using tsim," *Physics Procedia*, vol.33, no. 0, pp. 1029-1035, 2012.

[6] Y. S. Shao and D. Brooks, "Isa-independent workload characterization and its implications for specialized architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 245-255, 2013.

[7] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, "Workload characterization on a production hadoop cluster: A case study on taobao," in *IEEE International Symposium on Workload Characterization*, pp. 3-13, 2012,

[8] J. Issa and S. Figueira, "Hadoop and memcached: Performance and power characterization and analysis," *Journal of Cloud Computing*, vol.1, no. 1, 2012.

[9] H. Yang, Z. Luan, W. Li, D. Qian, and G. Guan, "Statistics-based workload modeling for mapreduce," in *Parallel and Distributed Processing Symposium Workshops PhD Forum*, pp. 2043-2051, 2012.

[10] P. Emma. "Understanding some simple processor-performance limits," *IBM J. Res. Dev*. vol. 41, no. 3, pp. 215-232, 1997.

[11] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *The 40th International Symposium on Computer Architecture*, pp. 237-248, 2013.

[12] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," preprint arXiv:1307.8013, 2013.

[13] M. Dimitrov, K. Kumar, P. Lu, V. Viswanathan, and T. Willhalm, "Memory system characterization of big data workloads," *IEEE Big Data Conference BPOE*, 2013.

[14] A. Makarov, V. Sverdlov, and S. Selberherr, "Emerging memory technologies: Trends, challenges, and modeling methods," *Microelectronics Reliability*, vol. 52, no. 4, pp. 628--634, 2012.

[15] Y. Xie, "Future memory and interconnect technologies," in *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE) 2013*, pp. 964-969, 2013.

[16] J. Chen, R. C. Chiang, H. H. Huang, and G. Venkataramani, "Energy-aware writes to non-volatile main memory," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 3, pp. 48-52, 2012.

[17] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang et al., "Advances and future prospects of spin-transfer torque random access memory," *IEEE Transactions on Magnetics*, vol. 46, no. 6, pp. 1873-1878, 2010.

[18] SPEC CPU 2006 benchmarks. https://www.spec.org/benchmarks.html

[19] Intel® Memory Latency Checker. http://www.intel.com/software/mlc.

[20] D. Sorin, V. Pai, S. Adve, M. Vernon, and D. Wood, "Analytic evaluation of shared-memory systems with ilp processors", *25th Annual International Symposium on Computer Architecture*, pp. 380-391, 1998.

[21] B. Fields, R. Bodík, M. Hill, and C. Newburn, "Using interaction costs for microarchitectural bottleneck analysis," *IEEE/ACM International Symposium on Microarchitecture*, pp. 228-239, 2003.

[22] T. Karkhanis and J. Smith. "A first-order superscalar processor model," *31st Annual International Symposium on Computer Architecture*, pp. 338-349, 2004.

[23] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," *31st Annual International Symposium on Computer Architecture*, pp. 76-87, 2004.

[24] Li. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. Soffa. "The impact of memory subsystem resource sharing on datacenter applications," *38th Annual International Symposium on Computer Architecture*, pp. 283-294. 2011.

[25] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. "Server engineering insights for large-scale online services," *IEEE Micro* vol. 30, no. 4, pp. 8-19. 2010.

[26] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pp. 37-48, 2012.

[27] Netlist (2012, July 24). *HyperCloud HCDIMM: Scaling the High Density Memory Cliff.* Figure: http://www.netlist.com/media/blog/hypercloud-memory-scaling-the-high-density-memory-cliff/

[28] T. Wenisch, R. Wunderlich, M. Ferdman, A.Ailamaki, B. Falsafi, and J. Hoe, "Simflex: statistical sampling of computer system simulation." IEEE Micro, vol. 26, no. 4, pp. 18-31, 2006.