MITTS: Memory Inter-arrival Time Traffic Shaping

Yanqi Zhou and David Wentzlaff
Electrical Engineering Department
Princeton University
Princeton, USA
yanqiz@princeton.edu, wentzlaf@princeton.edu

***Abstract*—Memory bandwidth severely limits the scalability and performance of multicore and manycore systems. Application performance can be very sensitive to both the delivered memory bandwidth and latency. In multicore systems, a memory channel is usually shared by multiple cores. Having the ability to precisely provision, schedule, and isolate memory bandwidth and latency on a per-core basis is particularly important when different memory guarantees are needed on a per-customer, per-application, or per-core basis. Infrastructure as a Service (IaaS) Cloud systems, and even general purpose multicores optimized for application throughput or fairness all benefit from the ability to control and schedule memory access on a fine-grain basis. In this paper, we propose MITTS (Memory Inter-arrival Time Traffic Shaping), a simple, distributed hardware mechanism which limits memory traffic at the source (Core or LLC). MITTS shapes memory traffic based on memory request inter-arrival time, enabling fine-grain bandwidth allocation. In an IaaS system, MITTS enables Cloud customers to express their memory distribution needs and pay commensurately. For instance, MITTS enables charging customers that have bursty memory traffic more than customers with uniform memory traffic for the same aggregate bandwidth. Beyond IaaS systems, MITTS can also be used to optimize for throughput or fairness in a general purpose multiprogram workload. MITTS uses an online genetic algorithm to configure hardware bins, which can adapt for program phases and variable input sets. We have implemented MITTS in Verilog and have taped-out the design in a 25-core 32nm processor and find that MITTS requires less than 0.9% of core area. We evaluate across SPECint, PARSEC, Apache, and bhm Mail Server workloads, and find that MITTS achieves an average 1.18x performance gain compared to the best static bandwidth allocation, a 2.69x average performance/cost advantage in an IaaS setting, and up to 1.17x better throughput and 1.52x better fairness when compared to conventional memory bandwidth provisioning techniques.**

I. Introduction

Off-chip memory bandwidth is a critical resource in multicore and manycore processors. This is especially important as the number of cores on a single chip is increasing at a rate faster than that of off-chip memory bandwidth. This disconnect may ultimately lead to off-chip memory bandwidth limiting the computational throughput provided by future multicore and manycore processors [1], [2], [3]. Even on current day multicore processors [4], [5], [6], performance can be very sensitive to off-chip memory bandwidth and latency.

As the number of cores on a chip increases, the need to multiplex memory accesses from many different applications or threads across memory controllers that are shared grows. Having multiple applications share a memory controller causes two primary challenges to emerge. First, different applications or threads react with varied sensitivity to delivered memory bandwidth and latency due to differing inherent bandwidth and latency requirements as well as distinct memory sharing patterns. Second, the policy of how much memory bandwidth and with what latency that bandwidth should be delivered to a core or application likely varies based on the use case, the amount of money a user is willing to spend, and system-level goals. Good examples of varied memory sensitivity and policy needs include Infrastructure as a Service (IaaS) Cloud systems and general purpose multiprocessor workloads where the system wants to optimize for application throughput or fairness.

Current IaaS providers charge concurrent applications equally on the basis of CPU-time. However, when using processors with limited bandwidth, CPU-time alone is not a good metric for performance. For instance, in IaaS Clouds, different applications, different virtual machines (VMs), and even different customers share the same on-chip interconnect, memory controller(s), and off-chip memory bandwidth. This can lead to large performance swings depending on the competing customers and applications [7]. Likewise, one customer can receive more access to the off-chip bandwidth than their fair-share simply by running a program that aggressively uses memory, effectively denying service to the other customers. Traditional memory schedulers do not focus on IaaS memory system provisioning needs. In a Cloud setting, having a policy which optimizes for fair access to memory bandwidth and latency is likely not even desirable. Some customers may be willing to pay more for additional off-chip memory bandwidth or lower latency while current memory scheduling schemes primarily focus on fair access to off-chip memory bandwidth instead of providing **higher bandwidth or lower latency proportional to payment**.

We contend that multicore and manycore systems require a more general and distributed way to precisely provision, schedule, and isolate memory bandwidth and latency on a per-customer, per-application, or per-core basis. In this work, we propose MITTS (Memory Inter-arrival Time Traffic Shaping), as a new way to limit and provision off-

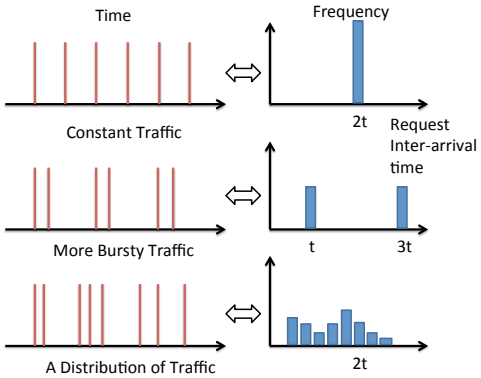


Figure 1: Latency vs. Bandwidth. Distribution captures both latency and bandwidth. The inter-arrival time determines the latency, and the frequency of each inter-arrival time determines the bandwidth.

chip memory bandwidth. **MITTS leverages the insight that application memory accesses can be classified by their statistical distribution of inter-arrival times.** For instance, applications that have very bursty traffic have a different memory request inter-arrival time distribution than applications which have a regular, periodic, access pattern even though they may use the same bandwidth over the lifetime of the program. The memory inter-arrival time distribution encapsulates both memory request latency and bandwidth. Figure 1 shows three examples of different memory access patterns. The top pattern has constant memory traffic which has a distribution of a single pulse, while bursty traffic (middle) has two pulses in its distribution. Real-world memory traffic resembles the last case (bottom), where the request inter-arrival time distribution has a complex shape. MITTS works by having a simple hardware traffic shaper for each core or thread that can shape memory traffic into a prescribed distribution. The memory traffic distribution that an application or thread is shaped into does not have to resemble the application's intrinsic memory traffic.

In a Cloud context, MITTS enables IaaS customers to not only request a desired amount of memory bandwidth, but also ask for different quantities of memory bandwidth which occur at different inter-arrival times. MITTS enables the Cloud customer to only pay for the quantity and distribution of traffic which they require, thereby decoupling the allocation of CPU resources from memory bandwidth. In a real-time system, MITTS enables fine-grain memory source throttling to provide bandwidth isolation. This prevents bad-actor applications from hogging the memory system. In a general purpose setting, MITTS enables per-application memory bandwidth provisioning of inter-request interval distribution to optimize overall application throughput or inter-application fairness.

In order to allocate memory bandwidth at a per-application, per-core, per-thread, or per-VM granularity, we propose a simple distributed hardware mechanism where a local memory bandwidth shaper is placed in a core or after an application's last level of cache (LLC) to limit memory request rate. **We have implemented MITTS in Verilog, have taped the design out in a 25-core 32nm processor, and chips have been tested in our lab**. We find that MITTS uses less than 0.9% of core area.

We study MITTS across a wide range of applications and show that applications have on average 1.18x increase in performance versus a non-distribution-based memory bandwidth limiter and show that when used in an IaaS marketplace, can lead to an average 2.69x gain in performance-per-cost versus a system that only allows static bandwidth provisioning. In a multi-program setting, MITTS is flexible enough to optimize system throughput, fairness, or real-time requirements. Simulation results show up to 17% throughput gain and 52% fairness improvement compared with other memory schedulers (FR-FCFS, Fair-Queue, TCM, FST, MemGuard, and MISE) on an 8-program workload.

Our key contributions include:

1) We propose distribution-based memory traffic shaping that enables allocating bandwidth by memory transaction inter-arrival time thereby opening a rich discussion of how to treat different traffic (burst vs. bulk).
2) MITTS can isolate memory bandwidth and access to the LLC on a per-core basis which can lead to more deterministic performance.
3) MITTS enables Cloud users to choose their own memory bandwidth and distribution, rather than merely relying on Cloud providers to provision based on system level goals.
4) We present an online genetic algorithm to configure MITTS hardware. It is adaptable to program phases and different input sets. It can be used by less savvy customers to optimize MITTS for an objective function.
5) MITTS enables Cloud providers to constructively configure MITTS for co-running programs for better system performance and fairness.
6) MITTS enables **fine-grain pricing** of memory bandwidth, opening opportunities for IaaS Clouds to charge for provisioned memory bandwidth that do not exist with current microprocessors.
7) We implemented MITTS in Verilog and taped it out in a 25-core 32nm processor. We determine its small area impact, and discuss design tradeoffs.

II. Background and Motivation

A. Challenges With Memory Systems

Previously proposed memory schedulers aim to improve system throughput and fairness [8], [9], [10], [11], [12], [13]. Achieving memory system fairness while maintaining performance is intrinsically difficult. Unlike network systems, application performance is not proportional to resources used, as memory performance is influenced by application

features (eg. working set size and memory intensity) and inter-application interference.

Application-aware algorithms [8], [9], [10], [13] improve CPU performance and fairness by prioritizing some applications over others, based on applications' memory access characteristics. These techniques are not robust due to the limitations in each of their designs. TCM [10] provides high performance, but can be highly unfair because of the way it clusters applications. TCM sometimes places high-memory intensity applications in the low memory-intensity cluster, unfairly prioritizing them. Moreover, application-aware algorithms rely on application ranking, which incurs higher hardware complexity in terms of critical path and chip area.

Performance of a shared memory system largely depends on thread-to-core mappings. When multiple cores share a resource, threads running on those cores can constructively or destructively use the resource. Heuristics and adaptive thread-to-core mapping techniques have been developed for better sharing LLC and memory bandwidth [7]. However, if the thread-to-core mapping fails, another technique such as MITTS is required to shape memory traffic before the LLC.

B. IaaS Provisioning and Isolation

Current Infrastructure-as-a-Service (IaaS) Clouds lack the ability to provision memory bandwidth on a per-customer basis according to customers' needs and payments. IaaS systems based on Virtualization technologies such as Xen [14], VMWare, and KVM enable flexible and dynamic provisioning of resources. However, current Cloud provisioning implemented by IaaS providers, e.g. Amazon EC2, is limited to the level of coarse grained VMs. Elastic resource provisioning of CPU, disks, I/O, and memory capacity [15], [16], [17], [18], [19] have been well studied. Memory bandwidth as a critical shared resource, however, is neither shared proportionally to payment, nor do cloud providers typically offer minimum guarantees on memory bandwidth.

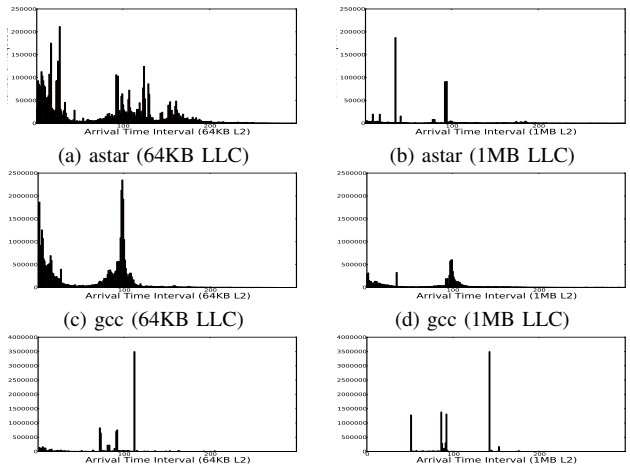
Current Cloud systems lack the ability to flexibly provision resources while guarantee performance isolation. Current Virtual Machine Monitors or Hypervisors provide isolation guarantees on some of the Cloud resources through strict CPU reservations and static partitioning of memory and disk space. Significant research has been dedicated towards providing isolation guarantees for resources such as disk bandwidth [20] and network bandwidth [21]. Some shared resources, including on-chip cache, interconnection network, and memory bandwidth, are difficult to isolate. System software has little control over such resources, and they are almost entirely managed by the hardware in a best effort manner. Therefore, co-located VMs can suffer interference or performance degradation due to static partitioning.

In order to gain the highest economic efficiency, resources can be allocated to the application or user that values them most [22], [23]. The amount of resources allocated to each application can be determined by the amount of money the customer is willing to pay, which is likely proportional to how much the customer values these resources or the utility they gain from their use. Economic efficiency can be enhanced by allocating resources at fine granularity. A fine-grain provisioning of memory bandwidth allows IaaS customers to request the desired amount of memory bandwidth, as well as ask for different quantities of memory bandwidth that occur at different inter-arrival intervals. Current IaaS systems allocate off-chip bandwidth based on the VM size purchased, which is not optimized for economic efficiency. For instance, CPU-intensive applications need large VMs with many cores but do not necessarily require large memory bandwidth. As an alternative, a Cloud system could allow users to decide exactly the amount of bandwidth and inter-arrival time of that bandwidth to purchase, and provision memory bandwidth based on market supply and demand.

C. Key Idea: Memory Traffic Distribution

Conventional memory scheduling fails to manage different aspects of memory bandwidth such as average bandwidth and burstiness in a generalized form. Prior research has categorized applications as latency-sensitive or bandwidth-sensitive. Latency-sensitive workloads are more CPU intensive while the latter are more memory intensive workloads. In conventional memory scheduling, latency-sensitive applications are given higher priority for system throughput [10]. We observe that applications have different sensitivity to different aspects of memory bandwidth, primarily latency. Memory-intensive applications with sufficient memory level parallelism are sensitive to the average bulk bandwidth but are not particularly sensitive to memory latency. Applications with bursty access patterns can benefit from an extra burst of bandwidth sporadically, but do not necessarily require a large average bulk bandwidth. We propose a generalized framework for describing both memory latency and bandwidth, the memory request inter-arrival time "distribution". The distribution describes how an application's memory requests are serviced at different intervals, and what percentage of requests fall into a specific inter-arrival time.

In our proposed memory distribution, the horizontal axis represents the time difference between two subsequent memory requests, while the vertical axis determines how frequent a request falls into a certain inter-arrival category. The inter-arrival time along with the frequency at which memory requests occur with that inter-arrival time determines the bandwidth consumed. With the knowledge of an application's distribution, customers are able to make rational decisions on the amount of bandwidth at a certain inter-request latency to purchase. The distribution reveals the fundamental parameters including memory bandwidth and how often memory requests are clustered in time. As a motivation for this work, Figure 2 shows distributions of memory request inter-arrival time with two LLC cache sizes for three



(a) astar (64KB LLC) (b) astar (1MB LLC)

(c) gcc (64KB LLC) (d) gcc (1MB LLC)

(e) libquantum (64KB LLC) (f) libquantum (1MB LLC)

Figure 2: Intrinsic Memory Request Inter-arrival Time Distribution for Three SPEC Benchmarks (64KB and 1MB LLC): Number of Requests vs. Inter-arrival Time

SPEC2006 benchmarks. We can see that using a larger LLC cache has two effects on the distribution: first it reduces the number of requests, second it moves the distribution right. The plot gives us the intuition that different applications have different requirements on memory bandwidth and memory bandwidth as a function of inter-request interval.

III. Architecture

A. Distributed Management: Source Control

We propose a distributed hardware mechanism where a local memory bandwidth shaper is placed within a core or after a VM's LLC to limit memory request rate for a particular core or thread, as shown in Figure 3. Most existing memory scheduling algorithms are memory controller-driven, and do not provide CPU initiated memory bandwidth provisioning. In contrast, we rely on memory request inter-arrival times at processor cores to shape memory bandwidth into the distribution that a user is willing to pay for or into a distribution which is determined by software policy, for instance a online auto-tuner which is optimizing memory utilization. We leverage insights from Source Throttling (FST) [11], which reacts to application slowdowns by throttling CPU cores. Unlike FST, MITTS not only controls the rate based on processor core's feedback, but also controls the distribution of request inter-arrival times. Also, different from FST, MITTS can shape memory request inter-arrival time distribution reactively using an online auto-tuner as well as proactively using an offline auto-tuner. In order to enforce bandwidth limits, the OS or hypervisor sets up memory control registers to configure an individual core's inter-arrival interval distribution. The use of memory bandwidth source control in a distributed way can scale up with multicore and manycore systems, as it does not rely on centralized hardware structures.

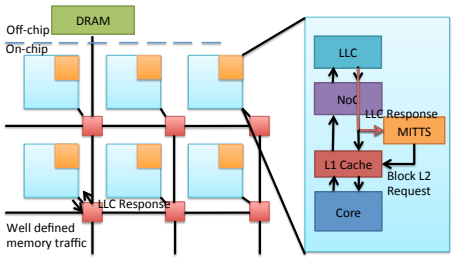


Figure 3: Distributed Memory Bandwidth shaper

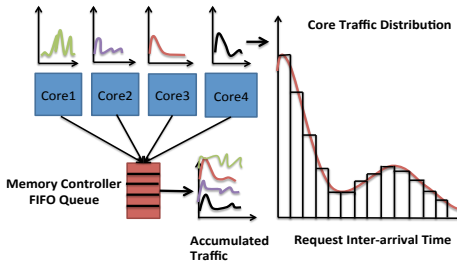


Figure 4: Memory Inter-arrival Time Traffic Shaping. MITTS limits memory traffic to fall into a certain distribution at processor core/LLC side.

B. Inter-arrival Time Traffic Shaping

As shown in the high-level architecture in Figure 4, different cores have different memory inter-arrival time distributions, and these distributions add up at the global level. Inside each distribution, the total number of requests at different inter-arrival time intervals can vary significantly. The larger the interval, the lower the rate of memory requests. Requests at a higher rate consume more instantaneous memory bandwidth, therefore should be charged more. In a Cloud context, customers are charged based on the distribution of traffic they purchase. **Our proposed hardware shapes application memory bandwidth by fitting the application's memory request inter-arrival time into a certain distribution.** The philosophy of provisioning a distribution is to better utilize fine-grain memory bandwidth and charge for memory requests that consume various amounts of bandwidth at various request intervals differently.

1) Bin-based Bandwidth Shaper: We propose a simple bin-based hardware memory bandwidth shaper that shapes memory request rate into a pre-determined arbitrary distribution. The bandwidth shaper gathers cache miss information and stalls the core when its memory bandwidth usage exceeds the pre-determined value. Inside the hardware memory bandwidth shaper, we have multiple traffic bins that contain available credits for memory requests with certain inter-arrival time, as shown in Figure 5. Each bin contains credits. One credit represents one memory transaction at a certain request interval and the issuing of a memory transaction consumes a credit. We allow arbitrary configuration of credits to bins in order to allow the bandwidth shaper to shape the

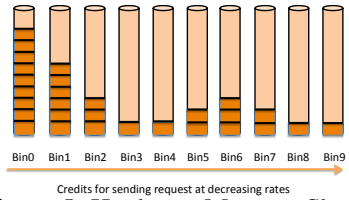


Figure 5: Hardware Memory Shaper

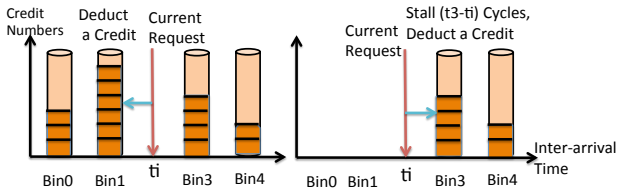


Figure 6: Credit Deduction Given the Request Inter-Arrival Time. Four bins have credits in left example.

memory traffic to any distribution. The maximum number of credits in a bin are bounded by the total memory bandwidth that a core can generate. The bandwidth shaper enforces that a core's memory traffic distribution does not exceed the prescribed or paid-for distribution by delaying (stalling) a memory transaction if there are no credits available in a bin with lower or equal to the memory request's inter-arrival interval. The delayed memory transaction will issue when it has been delayed enough that its inter-arrival interval matches a farther out bin which has credits, or credits have been replenished as described below. Figure 6 shows how a memory request consumes a credit based on the inter-arrival time.

Table I shows the variables used throughout this section. Each bin is a container holding credits for memory requests with a certain request rate (inter-arrival time). We define the total number of bins N, where N can be determined by how fine-grain the quantization of inter-arrival interval should be. For simplicity, we use N=10 (ten bins) throughout our simulations. Memory requests with inter-arrival time $t \in [t_i - \frac{L}{2}, t_i + \frac{L}{2})$ fall into bin_i. We find that for most benchmarks, memory request inter-arrival times are smaller than $100-200$ cycles with a small LLC cache. We decide to use 10 bins, with interval length of $L = 10$ CPU cycles in our simulations. If applications have intrinsically larger inter-arrival times, MITTS can be modified by increasing L to accommodate this large interval requirement.

In an IaaS context, the pricing of the replenishment-rate of bins is not a hardware construct, but rather a market or software constraint. The hardware mechanism that enforces the replenishment rate is described below and is set by software (OS or hypervisor). While we leave the pricing of bins up to software and the market, bins should be priced at least commensurate with the amount of bandwidth they provide. In all likelihood, bins with a lower inter-arrival interval will be even more costly than their bandwidth dictates as they provide preferential treatment of traffic.

N	Total number of bins
L	Time interval length of each bin (10 CPU cycles in our simulation)
t_i	Inter-arrival time represented by bin_i, Requests with inter-arrival time $[t_i - \frac{L}{2}, t_i + \frac{L}{2})$ fall into bin_i
b_i	Equivalent bandwidth bin_i represents
n_i	Number of credits in bin_i
K_i	Number of credits will be replenished to bin_i
T_r	Overall replenishment period
T_c	Current interval counter

Table I: Parameters of Bin-based Algorithm

2) Bin Credits Replenishment: The MITTS hardware shaper is designed to force the memory traffic from a core or LLC into a traffic distribution. In order to achieve this, the credits in different bins need to be replenished at a regular period or at different rates. We define a replenishment period $T_r = \sum_{i=0}^{N-1} K_{max} \times t_i$, where K_{max} is the maximum bin size. Ideally all credits should be used up within this period. In the hardware, we have a register to store the replenishment period T_r, and a small counter T_c used to store time to next replenishment. A table K with N entries stores the number of credits each bin should be replenished/reset to at the end of each replenishment period. Algorithm 1 is a "reset" based replenishment, where all bin credits are reset after T_r cycles.

Algorithm 1 Bin Credits Replenish Algorithm 1

```
if T_c == T_r then
    for i in 0 : N − 1 do
        n_i := K_i
    end for
    T_c := 0
end if
```

C. Global Traffic Management

MITTS is flexible enough to enable both oversubscription (overprovisining) and undersubscription (provisioning) of off-chip memory bandwidth depending on how many credits the software policy globally doles out. The provisioned case only allocates total chip-wide credits that correspond to less than the off-chip bandwidth, therefore MITTS can restrict the global **average** bandwidth. Unfortunately, restricting global **instantaneous** memory bandwidth is challenging because replenishment time may be long relative to memory request processing time. Therefore, it is possible that all cores may choose to spend their bursty credits at the same time. Note that this does not exceed the aggregate average bandwidth, but is instead a short term problem. This is also a probabilistically unlikely event. A simple solution to this challenge is to have a FIFO at the memory controller that is large enough to absorb global traffic burstiness and smooth out accesses to off-chip memory. If the size of the FIFO is deemed too large, the FIFO can be made smaller and memory requests can back up to the cores which will also cause smoothing in the unlikely case that all cores use their high-priority credits at the same time. We use a small (32-entry) fixed-sized FIFO in our simulation results. More complex schemes are possible which communicate short-term

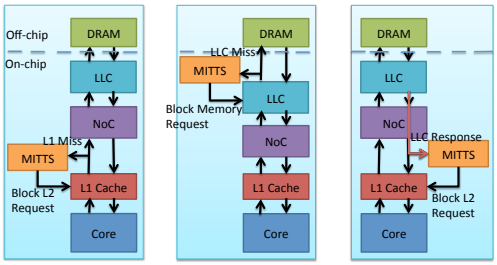


Figure 7: Design Tradeoffs of MITTS. Left: After L1; Middle: After LLC; Right: Hybrid.

congestion to the MITTS units which then proportionally scale-down resources until the congestion is resolved, but we leave this to future work.

D. Hardware Design Tradeoff

There are multiple possible locations to place MITTS, as shown in Figure 7. As shown in the left most diagram, we can place MITTS after the L1 cache (or last private cache). In this case, MITTS treats an L1 miss as a memory request. This solution works well for single level caches or chips with only private caches, but when used with shared caches, this is inaccurate because shared LLC hits will be treated as memory requests. Alternatively, we can place MITTS after the LLC, as shown in the middle diagram, where a LLC miss will be treated as a memory request. This is accurate in tracking requests' inter-arrival times, however, it is difficult to implement in a distributed, shared LLC. In a shared LLC, memory requests can be mapped to different cache banks (directories), making per-core/per-thread/per-VM based MITTS infeasible because the needed information is distributed at the cache banks.

In order to solve this problem, we propose a hybrid solution (right). We place MITTS after the L1 cache, but shape traffic using hit/miss information reported back on a per-request basis from the (possibly distributed) LLC. This solution introduces latency between the shaper and the information used to rate limit, but this latency is likely small with on-chip LLC. There are two possible solutions to this delayed information challenge which are described below.

1) Speculate LLC Hit and Rollback if Miss: The first method tracks the request issue time that an L1 miss occurs using a timestamp. If the LLC responds that the request is an LLC miss, the timestamp is compared to the previous LLC miss timestamp to determine which bin a credit should be removed from. The rate-shaper's credits for a given bin may lag what they should actually contain, but only by the number of outstanding L1 to LLC memory transactions which are also LLC misses. This can cause MITTS to fail to block a memory request so this approach is slightly aggressive. In order to track the per-request timestamp, we use a tag-indexed timestamp table to store the L1 miss timestamp for every inflight L1 to LLC request.

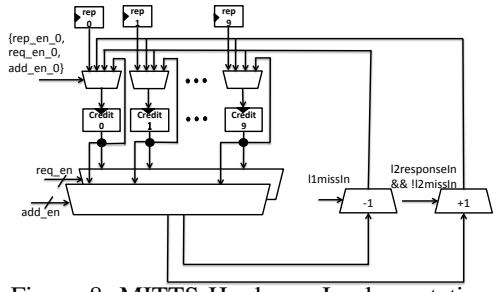


Figure 8: MITTS Hardware Implementation.

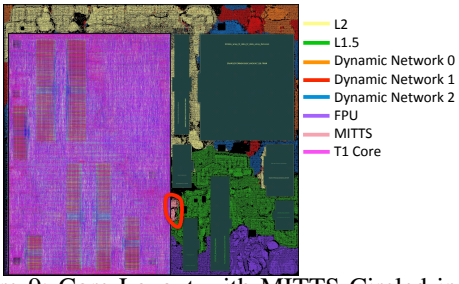


Figure 9: Core Layout with MITTS Circled in Red.

2) Speculate LLC Miss and Rollback on Hit: The second method is more conservative and assumes that an L1 miss is an LLC miss (deducts credits), but then adds back in credits on LLC hit notification. Instead of storing timestamps per L1 miss, this design stores bin number per L1 miss in order to credit to the appropriate bin on LLC hit. The size of this small table is determined by the maximum number of in-flight L1 to LLC requests. Method two is slightly simpler than method one and may have less out-of-sync bin counters because determining an LLC hit is likely faster than waiting for a main memory response to determine an LLC miss in method one. This problem can be rectified in method one if the LLC sends an extra message once it determines miss to overlap main memory hit/miss time with MITTS updating.

In our 25-core chip implementation, we have chosen method two. Figure 8 shows the simple hardware. Each time there is an L1 miss (l1missIn), a credit is deducted from the interval selected bin. On LLC hit, one credit is added back (l2responseIn && ! l2missIn) to the bin which it was removed from. At replenishment time, all bins are replenished.

E. Hardware Cost

We have implemented the entire MITTS design in Verilog RTL in a 25-core processor that was taped out in March 2015 in IBM's 32nm SOI process and we have received chips back which have been tested in our lab. The MITTS RTL code is integrated in and released as part of the open source OpenPition [24] project. The area used by MITTS is $0.0035mm^2$, which is less than 0.9% of the core area for the 25-core OpenSPARC T1-based processor which we have taped-out (16KB L1I/core, 8KB L1D/core, 64KB L2/core). Figure 9 shows the layout of a core with MITTS circled. Each MITTS module contains a

register per bin to track current credits and a register per bin to hold the number of credits to replenish at the end of the replenishment period. Each register corresponds to the maximum number of credits a bin can contain which we set to 1024 (10-bits). This is likely over-designed. A counter is used to track the inter-arrival period since the last memory transaction. A tag-indexed table is used to store the bin number from which a credit is deducted after an L1 miss while it is pending. A subtractor is needed to decrement the credit from one bin after an L1 miss, and an adder is required to add back the credit after an LLC hit. A zero detector signals if a bin that is needed is empty. MITTS' area cost is minimal when compared with complex comparison trees and thread state tables used in most centralized memory controllers.

F. Software Implications

MITTS is a hardware mechanism that enables better policy making. Policy making in the software stack is not the focus of this paper. However, we want to propose several solutions for using MITTS in an IaaS Cloud. The primary question is how an IaaS user determines the optimal bin configuration for their application and use case? A basic solution is to profile their applications with their specific input set and objective functions (utility functions), and set the configuration based on the profile. Profiling is good for stable workloads with fixed input size. Alternatively, customers can search the bin configuration space likely using an online auto-tuner. Amazon AWS and some third-party cloud management services provide schedule-based and rule-based auto-scaling mechanisms. Schedule-based auto-scaling allows users to change bin configuration at a given time, such as "add n credits to bin m between 8AM to 6PM each day". Rule-based mechanisms allow users to define triggers by specifying bin reconfiguration thresholds and actions, such as "run Genetic Algorithm to reconfigure bins when the application's objective function is below a threshold value". The auto-tuner will search the configuration space by varying the bin configurations, and select the best configuration provided a user-defined objective function. In the evaluation section, we use both offline profiling and online auto-tuning to evaluate MITTS.

IV. Evaluation

We show MITTS has up to 1.68x performance gain compared with static bandwidth allocation, up to 17% throughput and 52% fairness gain compared with conventional memory scheduling techniques, and up to 10x performance-per-cost gain for economic efficiency.

A. Simulation and Workloads

We model both the CPU core and memory system using the cycle-accurate simulator SDSim which is adapted from the core simulator SSim [25], [26], and the DRAM simulator DRAMSim2 [27]. We integrated DRAMSim2 with the SSim frontend, which models out-of-order cores with out-of-order memory systems. SSim is driven by the GEM5 Alpha ISA full system simulator [28], and both trace-driven simulation and execution-driven simulation can be performed with SSim. When the simulation completes, SDSim reports the cycles executed for a given workload along with statistics, such as cache miss rates, and aggregate memory latency. We model a shared LLC and memory system for the multi-program workloads. Table II shows the details of the simulated system unless otherwise noted.

Core	2.4GHz, 4-wide issue, 128-entry instruction window
L1 Caches	32 KB per-core, 4-way set associative, 64B block size, 8 MSHRs
L2 Caches	64B cache-line, 8-way associative, Single-program: 64KB, multi-program: 1MB shared
Memory controller	32-entry transaction queue depth
Memory	Timing: DDR3, 1333 MHz Organization: 1 channel, 1 rank-per-channel, 8 banks-per-rank, 8 KB row-buffer

Table II: Base Simulation Configuration

Workload 1	gcc, lib, bzip, mcf
Workload 2	Apache, libquantum, bhm mail server, hmmer
Workload 3	astar, bhm mail server, libquantum, bzip
Workload 4	gcc, gobmk, libquantum, sjeng, bzip, mcf, omnetpp, h264ref
Workload 5	bhm mail server, astar, libquantum, sjeng, bzip, mcf, omnetpp, h264ref
Workload 6	Apache, astar, gobmk, sjeng, bzip, mcf, omnetpp, h264ref

Table III: Multi-program Workloads

We perform our studies on both single-program and multiprogram workloads made of applications from SPECint 2006, PARSEC, Apache, and bhm mail server. We construct multiprogram workloads by randomly combining benchmarks, as shown in Table III. We run each workload for 200 million ROI cycles. For Apache, we run 3000 requests with concurrency of 10.

B. Optimizing Bin Configuration

With the capability to configure the hardware bins for fine-grain bandwidth provisioning, we also need an efficient algorithm to determine the optimal bin configurations. As there are 10 bins in total in our hardware bandwidth shaper, the search space could be (K_{max}^{10}), where K_{max} is the total number of credits allowed in a bin. Hill climbing and gradient descent [29], [30] are well-known algorithms, but they are likely to get stuck in a local optimal solution. We decide to use a genetic algorithm, which is suitable for large search space, and is capable of escaping from local optimal solutions. Genetic algorithms work well for non-convex search spaces like the one we are searching. For all of the results presented, we use an online genetic algorithm and an offline genetic algorithm to optimize bin configurations. The online genetic algorithm acts as an auto-tuner that can adapt to program phases and input sets.

The offline algorithm optimizes for a single choice of bin configurations across a whole program with 20 generations and 30 children per generation. A multi-phase offline genetic algorithm optimizes different phases separately. In order to reduce the amount of profiling and adapt to variable phases and input sets, we also designed an online genetic algorithm

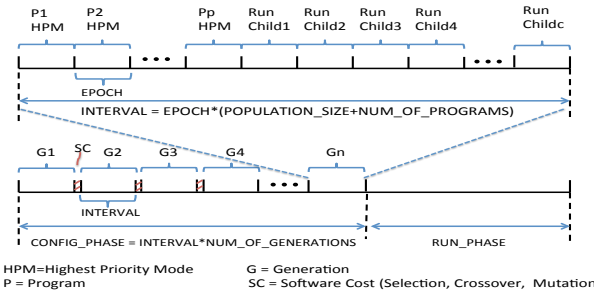


Figure 10: Online Genetic Algorithm. CONFIG_PHASE consists of multiple generations composed of multiple EPOCHs. Genetic Algorithm runs at the end of each generation in CONFIG_PHASE.

that configures MITTS at program runtime.

The online genetic algorithm in Figure 10 configures MITTS at the beginning of the program or a program phase (CONFIG_ PHASE), and uses the optimal configuration for the rest of program or phase (RUN_ PHASE). We leverage an online profiling technique used in MISE [13] to measure application slowdown. The CONFIG_PHASE is composed of n INTERVALs, where n is the number of generations. Each INTERVAL is composed of multiple EPOCHs. The first several EPOCHs are used to measure memory request rate in highest priority mode for each program, and will be later combined to calculate application slowdown when mixed with other applications ($slowdown\ of\ an\ App = (1-\alpha)(\alpha\frac{Request\ \ Service\ \ Rate\ \ with\ \ Highest\ \ Priority}{Shared\ \ Request\ \ Service\ \ Rate})$, $\alpha = \frac{Cycles\ \ spent\ \ stalling\ \ on\ \ memory\ \ requests}{Total\ \ number\ \ of\ \ cycles}$). Then the algorithm runs every child configuration sequentially for EPOCH time, and stores the measured objective function (throughput, fairness, etc.) in a dedicated address. After all configurations in one generation are evaluated, the software runtime selects the best configurations and uses them to create the genomes of the next generation (crossover and mutate). The new configurations are stored at special memory addresses, which are loaded by each program at the beginning of a new generation. The optimal configuration at the end of CONFIG_PHASE is used for the rest of program or program phase. For a phase-based online genetic algorithm, the CONFIG_PHASE occurs at the beginning of each phase so that it can adapt to program phase change. After studying the effects of different parameters (EPOCH, population, etc.), we use an EPOCH size of 20000, population of 30, and 20 generations. The software runtime is only called 20 times, with about 5000 cycles overhead each. The online algorithm has a comparatively small measured software overhead, and we include this overhead in our results.

C. Static Bandwidth Allocation Comparison

We evaluate MITTS's performance gain compared with a static memory bandwidth allocation. The static allocation mimics a less sophisticated memory system limiter that can limit a program's memory requests at or below a constant rate but cannot take into account inter-arrival times. In this section, we show that MITTS always outperforms the static case with the same average bandwidth (1GB/s).

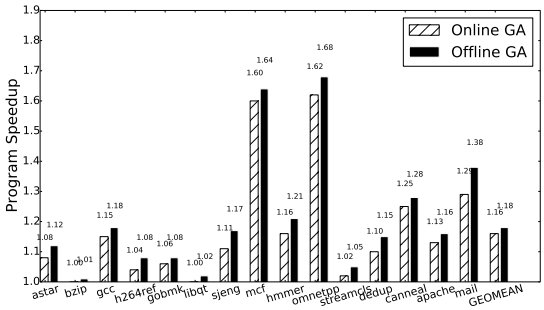


Figure 11: Performance Gain Compared with Static Bandwidth Provisioning

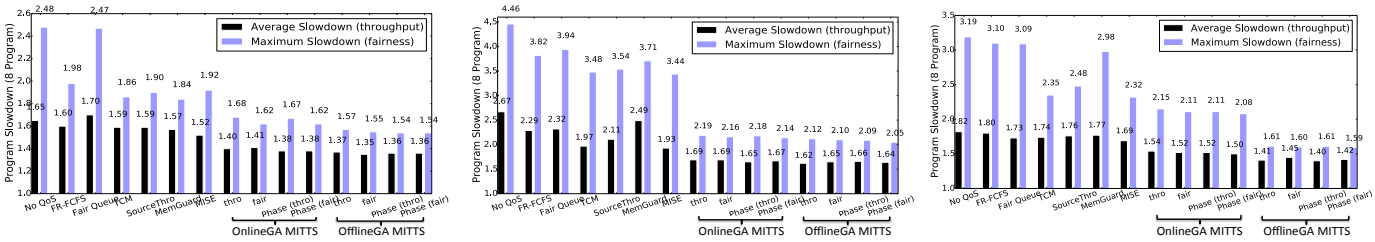
For a given distribution, as shown in Figure 5, the number of credits in each bin and the inter-arrival times together determine the average interval between memory requests. This can be computed as $I_{avg} = \frac{\sum n_i t_i}{\sum n_i}$, where n_i is the number of credits in bin_i and t_i is the inter-arrival time of memory requests that use credits in bin_i. The average bandwidth is computed by dividing the total traffic in a certain period T_r by time T_r.

We configure MITTS with exactly the same average inter-arrival time and bandwidth as the static allocation. We optimize MITTS's bin configuration so that for a fixed average interval and fixed average bandwidth (equal to the static case), the performance is maximized. The constraint functions can be formulated as: $I_{avg} = \frac{\sum n_i t_i}{\sum n_i} = I_{static}$, $B_{avg} = \frac{\sum b_i n_i t_i}{P} = \frac{\sum n_i}{P} = B_{static}$, where b_i is the bandwidth of bin_i.

Figure 11 shows the performance gain that MITTS achieves when compared with static memory bandwidth provisioning. With an offline genetic algorithm, mcf and omnetpp have 1.64x and 1.68x performance gain respectively and the Geometric Mean is 1.18x better. This shows that given a certain bandwidth limit, MITTS better allocates memory bandwidth for performance. The online genetic algorithm performs slightly worse than the offline GA.

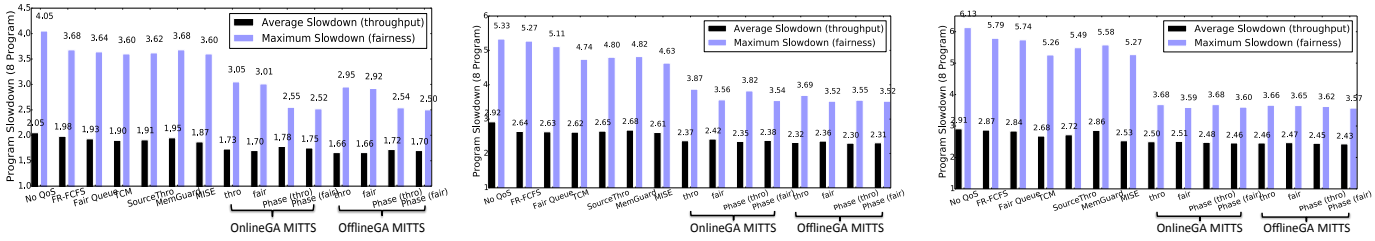
D. Effect on Inter-application Interference

In this section, we show MITTS's effectiveness in coordinating memory bandwidth shared by multiple programs. We use the two conventional metrics, throughput and fairness, to ensure that no application is unfairly slowed down while maintaining high performance. We use application average slowdown $S_{avg} = \frac{\sum_{i=0}^{N-1} \frac{T_{shared}}{T_{single_i}}}{N}$ and maximum application slowdown $S_{max} = max\{\frac{T_{shared}}{T_{single_i}}\}$ as measures of throughput and fairness. The lower the values of S_{avg} and S_{max} the better the throughput and fairness. In the simulation, we use four programs for workloads 1-3 and eight programs for workload 4-6, and let MITTS optimize the bin configurations for throughput and fairness separately. Each benchmark can have a different MITTS bin configuration.



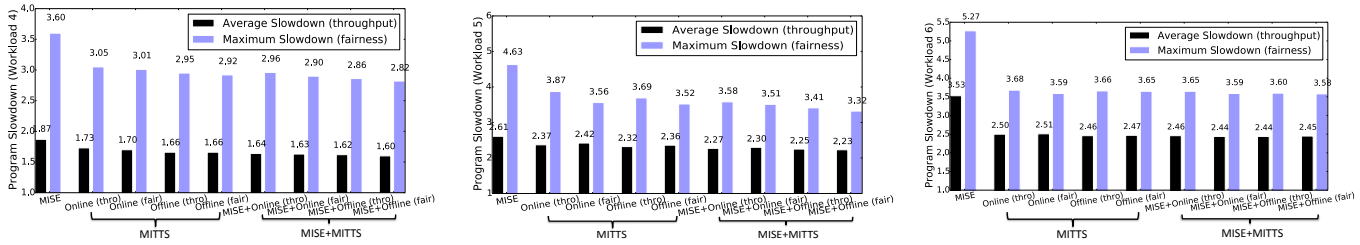
(a) Workload 1 (b) Workload 2 (c) Workload 3

Figure 12: Four-Program Throughput and Fairness Comparison with Conventional Memory Schedulers. Average slowdown and maximum slowdown are measures of throughput and fairness. (lower is better)



(a) Workload 4 (b) Workload 5 (c) Workload 6

Figure 13: Eight-Program Throughput and Fairness Comparison with Conventional Memory Schedulers. Average slowdown and maximum slowdown are measures of throughput and fairness. (lower is better)



(a) Workload 4 (b) Workload 5 (c) Workload 6

Figure 14: Comparison of MISE, MITTS, and MISE+MITTS. (lower is better)

We compare MITTS versus conventional memory scheduling techniques (FR-FCFS, Fair Queue, TCM, FST (denoted as "Source Thro"), MemGuard, and MISE) and show MITTS is more effective in improving system throughput and fairness. We run benchmarks concurrently on a 1MB shared LLC with a shared memory system. For TCM, we use a ClusterThresh of 2/N and a time quantum of one million cycles, as suggested in the paper. In MISE, we use an epoch length of 10000 cycles and an interval length of 5 million cycles as suggested in the paper. We exhaustively searched all parameter configurations of MISE presented in the MISE paper and present the best result here. We do our best to reproduce these algorithms based on suggested parameters in the original papers but have implemented them in our simulator framework. Figure 12 and Figure 13 show that MITTS can be configured for both throughput and fairness for multi-program workloads sharing the memory system. Compared with the best conventional scheduling in each of the six workloads, MITTS improves four-program throughput/fairness by 11%/17% (workload 1), 16%/40% (workload 2) and 17%/52% (workload 3), and improves eight-program throughput/fairness by 11%/30% (workload 4), 12%/24% (workload 5), and 4%/32% (workload 6). The online genetic algorithm performs a little worse compared with the offline genetic algorithm, because of the imperfect online measurement of application slowdown and introduced software overhead. However, it can adapt to phase changes and input set changes, thus is more practical to use for Cloud customers.

We also evaluate phase-based online/offline MITTS by dividing an application into five phases and optimizing MITTS configuration for each phase. In Figure 12 and in Figure 13, we observe small throughput and fairness gain over non-phase optimized MITTS for both four-programs workloads and eight-program workloads. We conclude that phased-based reconfiguration provides better opportunity to improve system throughput and fairness. Phase-based online algorithm reconfigures MITTS at the beginning of each phase, enabling online adaptation for phases and input sets.

The advantages of MITTS compared with conventional

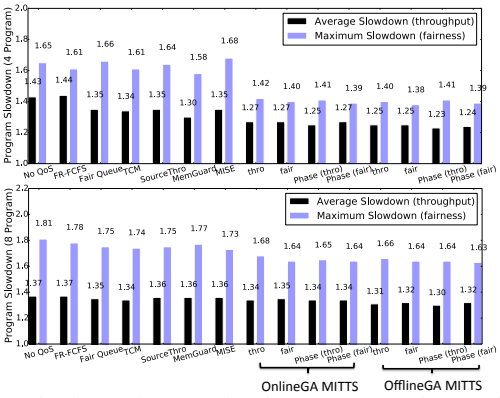


Figure 15: Throughput and Fairness Comparison With 8MB LLC (Workload 1 (top) and Workload 4 (bottom))

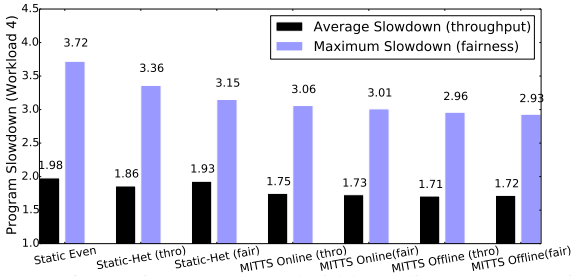


Figure 16: Static Even Bandwidth Split versus Optimal Heterogeneous Static Bandwidth Allocation.

scheduling techniques include: 1. It counters destructive effects at a shared LLC. If an application unfairly slows down other applications by evicting others' data or hogging LLC bandwidth, it is desirable to reduce the memory bandwidth usage for that application. MITTS is able to limit bandwidth at the core before it reaches the LLC which many of the other memory scheduling algorithms are unable to do; 2. It enables a larger memory request scheduling window compared with a centralized memory controller. In a conventional memory system without source throttling, a central queue can get filled with aggressive memory requests from a memory intensive application; 3. It allows applications to have different memory request inter-arrival distributions, which enables some constructive inter-application effects.

1) Larger LLC: In our evaluation, we have primarily focused on future manycore systems which have modest cache sizes such as Tilera, Cavium, and Xeon Phi processors. While our primary results are for the above mentioned cache sizes, we also evaluate MITTS using an 8MB LLC which approximates a current day multicore processor. Figure 15 compares MITTS against prior work for a 4-program (workload 1) and 8-program (workload 4). When using a larger LLC, there are fewer off-chip LLC cache misses, but MITTS still outperforms the best conventional memory scheduling technique by 5.3%/12.7% in throughput and fairness for workload 1 and 2.3%/6% in throughput/fairness for Workload 4.

E. Hybrid Method: MITTS+MISE

One important question that arises is how does performance react when using per-core MITTS with an intelligent centralized memory controller. We evaluate this hybrid model combining MITTS at each core with MISE used as the centralized memory controller as MISE performed best on average. We evaluated this hybrid approach across our eight application workloads and found additional throughput and fairness gains. Figure 14 shows the hybrid method achieves on average additional 4% and 5% throughput and fairness gain compared with MITTS only. This experiment implies MITTS complements existing centralized controllers.

F. Bandwidth Isolation

MITTS can enable better bandwidth isolation (important to real-time systems) and efficiency versus statically allocating bandwidth between applications. To evaluate this, we compare MITTS with a static, even bandwidth allocation where each application gets the same amount of memory bandwidth, and a static heterogeneous allocation where each application gets an optimal portion of bandwidth. We optimally configure MITTS for throughput and fairness separately while guaranteeing that MITTS does not over-provision bandwidth. From Figure 16, we can see for eight-programs (workload 4), MITTS has significantly higher throughput and fairness. Compared with static allocation and heterogeneous static allocation, MITTS is 14%/21% and 8%/7% better in throughput/fairness. This implies that MITTS could be applied to real-time systems to provide better application memory bandwidth isolation while maintaining efficiency.

G. Performance per Cost for an IaaS System

We next apply MITTS to an IaaS Cloud setting and show that MITTS improves economic efficiency by evaluating MITTS's performance per cost (efficiency) versus static bandwidth provisioning. In IaaS systems, customers care about cost as well as performance, making economic efficiency an interesting metric. In order to achieve this, we need a notion of pricing that can compare memory bandwidth and core cost. We assume that a processor core costs the same as 1.6GB/s of bandwidth.

1) Bin-based Credit Pricing: Credits in different bins represent different memory request rates. The lower latency the credit represents, the higher instantaneous bandwidth the credit enables, and the more expensive the credit likely should be. For instance, one credit in bin_i represents a higher instantaneous bandwidth than one credit in bin_j where i is smaller than j. A memory request with inter-arrival time t can only be issued if there are credits available in bins whose t_i is smaller than t.

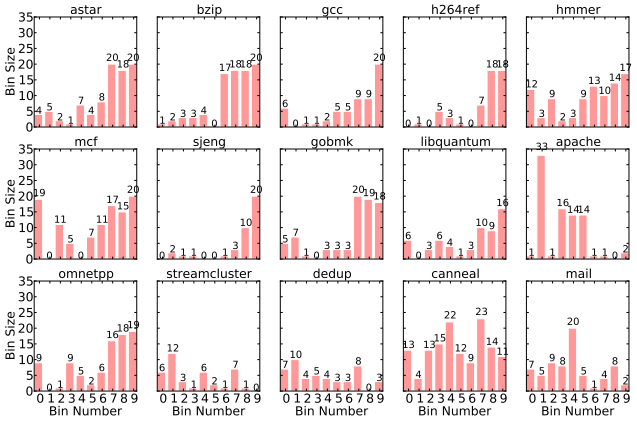


Figure 17: Optimal Bin Configurations per application for Performance/Cost. Credit price is proportional to the bandwidth it stands for, credits of higher-request rate are penalized by a linear scale factor $2-(t_i/t_N)$.

2) Optimal Bin Configurations for Performance/Cost: Different applications have different optimal bin configurations for a given objective function. Figure 17 shows the best bin configurations found across our benchmark suite. We optimize for performance-per-cost using the pricing scheme in Section IV-G1. Different benchmarks have dramatically different bin configurations. Memory intensive applications, such as mcf, have a large number of credits in high-request-rate bins (bin_0), and a significant number of credits in all other bins. Less memory intensive applications (sjeng, bzip, etc.) have only a few credits in high-request-rate bins due to the high cost and low reward. PARSEC benchmarks have lower overall memory intensity compared with SPEC, and their bin sizes are small compared with SPEC. Ideally, MITTS can provision a distribution of memory bandwidth that best satisfies an application's need.

3) Performance/Cost Gain Compared with Static Bandwidth Provisioning: MITTS improves economic efficiency by enabling fine-grain resource allocation and pricing. We compare performance-per-cost of MITTS versus static bandwidth provisioning with only one fixed inter-arrival time allowed (configurations with only credits in one bin). These configurations only allow memory requests being sent at or below a certain rate. As credit price is proportional to the request rate it allows, there is a tradeoff in balancing expensive bandwidth and low-cost bandwidth. As a baseline, for each benchmark, we find the optimal fixed inter-arrival time configuration with highest performance-per-cost. We do this by searching all configurations with any number of credits in only one bin while optimizing for performance-per-cost. This static best case is the optimal static bandwidth provisioning. We then find the optimal configurations for MITTS, and compare it with the best static bandwidth provisioning. Figure 18 shows that MITTS can achieve significantly higher efficiency. The geometric mean shows a

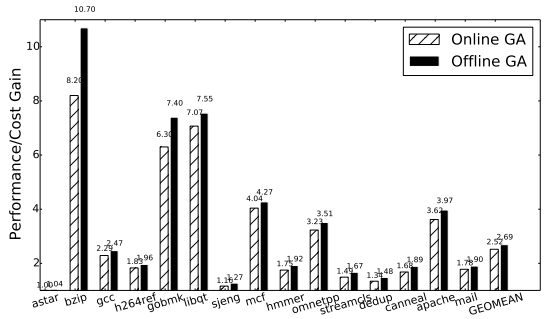


Figure 18: Efficiency Gain (Performance/Cost) Compared with the Optimal Static Bandwidth Provisioning. The optimal static case has a fixed request rate with highest performance/cost.

2.69x increase over static optimal bandwidth provisioning. MITTS enables Cloud customers to value memory bandwidth with different latencies differently and purchase the right amount of resources for the job.

H. Threaded Applications

Using MITTS for multi-threaded applications presents some interesting design options. MITTS could have per-thread bin configurations with different configurations in each thread, or a configuration which shares bin credits for all threads in a single program. From an OS perspective, context swapping MITTS state is simple as the MITTS bin configurations are exposed in a set of configuration registers can be swapped as part of the thread state.

We studied the effectiveness of per-thread MITTS for threaded applications as we expected to see different bandwidth requirements for different threads within a multi-threaded program. We ran x264 and ferret, both with a shared MITTS and per-thread MITTS. To our surprise, the results of a shared MITTS are over 2x better than a per-thread MITTS. A shared MITTS allows threads to share bin credits, which is useful when some threads are idle or cannot use up their credits within a replenishment window. A per-thread scheme wastes credits when a thread cannot use up its credits during some program phases. We think a phase-based, per-thread MITTS can perform better, and we leave this as future work.

I. Varying Bin Numbers

As a design sensitivity analysis, we have explored varying the number of credit bins used in the MITTS memory shaper. Using the methodology and workloads of Section IV-D we have found that more bins outperform fewer bins, but with diminishing returns. On average, 6-bins outperforms 4-bins by more than 10% in throughput and fairness, 8-bins outperforms 6-bins by 5%, and 10-bins outperforms 8-bins by 2%. While having more bins enbles higher performance, they are also more difficult to configure.

V. Related Work

Main memory is a critical resource for all computing systems. Techniques that minimize memory capacity and bandwidth waste have been proposed [31], [32], [33]. Fine granularity access/storage and data compression in data caches [34] and memory [34], [35], [36] effectively conserve memory bandwidth and better utilize cache capacity.

FR-FCFS [37] aims to maximize DRAM throughput by prioritizing row-buffer hits. It unfairly favors applications with higher row-buffer hits or higher memory intensity. A Fair Queuing memory scheduler [38] ensures each thread receives its allocated fraction of the memory system regardless of the load placed by other threads. It overcomes the drawbacks of FR-FCFS [37] and FCFS. Adaptive bandwidth management [39] periodically tunes the bandwidth assigned to the sharers with OS or hypervisor support, which improves average and worst case service latencies. MITTS allocates fine-grain memory bandwidth based off memory traffic distribution, not a constant fair share of bandwidth.

Slowdown-based memory schedulers are proposed for memory system fairness. STFM attempts to estimate each application's slowdown, aiming to improve fairness by prioritizing the most slowed down application [40]. MISE [13] proposes a new way to estimate the slowdown of an application as the ratio of its uninterfered and interfered request service rates. These measurement-based scheduling algorithms introduce hardware complexity (worse clock and area). In contrast, MITTS is very simple to implement. MITTS provisions fine-grain memory bandwidth at a per-core basis, which is compatible with manycore systems.

Thread Cluster Memory Scheduling [10] addresses system throughput and fairness separately by dividing threads into two clusters and employing different memory scheduling policies. Application-aware Channel Partitioning [41] achieves the similar goal by mapping data from applications that severely interfere with each other to different memory channels. Categorization is not accurate and scalable, because some applications cannot fit any coarse categorization. MITTS generalizes memory access patterns into a distribution, and provisions bandwidth accordingly.

MemGuard [42] is a memory bandwidth reservation system, that distinguishes memory bandwidth as two parts: guaranteed and best effort. It provides bandwidth reservation for the guaranteed bandwidth for temporal isolation, with reclaiming to maximally utilize the reserved bandwidth. It does not account for system fairness as a demanding application can potentially get the most memory bandwidth.

Source-based rate control techniques for QoS [43] use clock modulation to reduce out-of-core resource contention. Source throttling [11] provides higher fairness and performance than fair partitioning by constraining the memory injection rate. Efficiency-aware QoS DRAM schedulers [44] provide different QoS based on different memory access requirements by heterogeneous functional units. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth [45] uses a novel mechanism to track progress of GPU workloads. MITTS proactively regulates per-core traffic based on the pre-determined traffic distribution of each application. MITTS not only controls the rate based on processor core's feedback, but also controls the distribution of request inter-arrival times. This has proved very important to achieve better performance. MITTS is additive to QoS-aware memory scheduling.

Reinforcement learning (RL) has been used to automatically allocate resources in data centers [46], [47]. They focus on assigning processors and memory to applications in software, but not memory bandwidth provisioning in hardware. A RL-based, self-optimizing memory controller [48] adapts DRAM scheduling policy based on its interaction with the system to optimize performance. The hardware cost is relatively high requiring an extra multiplier and on-chip storage. MITTS requires only minimal hardware cost. But, we feel that RL could be applied to configure MITTS.

We take inspiration from network traffic shaping, such as Leaky Bucket [49] and Token Bucket [50], where the bucket size determines the burstiness of traffic.

VI. Conclusion

MITTS shapes memory traffic based on request inter-arrival time. This enables source limiting of memory usage and charging based on the memory request inter-arrival distribution. MITTS has up to 1.68x performance gain compared with static bandwidth allocation, up to 17% throughput and 52% fairness gain compared with conventional memory scheduling techniques, and up to 10x economic efficiency for an IaaS system. We implemented MITTS in Verilog and it has been taped-out in a 25-core microprocessor.

Acknowledgements

This work was partially supported by NSF under Grants No. CCF-1217553, CCF-1453112, and CCF-1438980, AFOSR under Grant No. FA9550-14-1-0148, and DARPA under Grants No. N66001-14-1-4040 and HR0011-13-2-0005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] B. Rogers *et al.*, "Scaling the bandwidth wall: challenges in and avenues for cmp scaling," ser. ISCA, 2009, pp. 371–382.

[2] A. Kagi *et al.*, "Memory bandwidth limitations of future microprocessors," in *ISCA 1996*, 1996, pp. 78–78.

[3] M. Pavlovic *et al.*, "On the memory system requirements of future scientific applications: Four case-studies," ser. IISWC, 2011, pp. 159–170.

[4] S. Bell *et al.*, "TILE64 Processor: A 64-Core SoC with Mesh Interconnect," in *International Solid-State Circuits Conference*, 2008.

[5] D. Wentzlaff *et al.*, "On-chip interconnection architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sep. 2007.

[6] ——, "Configurable fine-grain protection for multicore processor virtualization," in *International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 464–475.

[7] L. Tang *et al.*, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011.

[8] O. Mutlu *et al.*, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," ser. ISCA, Washington, DC, USA, 2008, pp. 63–74.

[9] K. Yoongu *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010, pp. 1–12.

[10] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," ser. MICRO, Washington, DC, USA, 2010, pp. 65–76.

[11] E. Ebrahimi *et al.*, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," ser. ASPLOS XV, 2010, pp. 335–346.

[12] R. Ausavarungnirun *et al.*, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," ser. ISCA, 2012, pp. 416–427.

[13] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013, pp. 639–650.

[14] P. Barham *et al.*, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[15] Y. Zhou *et al.*, "CASH: Supporting IaaS customers with a sub-core configurable architecture," ser. ISCA, 2016.

[16] E. Kalyvianaki *et al.*, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," ser. ICAC. New York, NY, USA: ACM, 2009, pp. 117–126.

[17] J. Rao *et al.*, "Self-adaptive provisioning of virtualized resources in cloud computing," ser. SIGMETRICS. New York, NY, USA: ACM, 2011, pp. 129–130.

[18] J. Pfitscher *et al.*, "Customer-oriented diagnosis of memory provisioning for iaas clouds," *SIGOPS Oper. Syst. Rev.*, vol. 48, no. 1, pp. 2–10, May 2014.

[19] L. Popa *et al.*, "Faircloud: Sharing the network in cloud computing," ser. SIGCOMM. New York, NY, USA: ACM, 2012, pp. 187–198.

[20] J. Zhang *et al.*, "Storage performance virtualization via throughput and latency control," *Trans. Storage*, vol. 2, no. 3, pp. 283–308, Aug. 2006.

[21] D. Gupta *et al.*, "Enforcing performance isolation across virtual machines in xen," ser. Middleware '06. New York, NY, USA: Springer-Verlag New York, Inc., 2006, pp. 342–362.

[22] C. Waldspurger *et al.*, "Spawn: a distributed computational economy," *Software Engineering, IEEE Transactions on*, vol. 18, no. 2, pp. 103–117, 1992.

[23] J. Kurose *et al.*, "A microeconomic approach to optimal resource allocation in distributed computer systems," *Computers, IEEE Transactions on*, vol. 38, no. 5, pp. 705–717, 1989.

[24] J. Balkind *et al.*, "OpenPiton: An open source manycore research framework," ser. ASPLOS. New York, NY, USA: ACM, 2016, pp. 217–232.

[25] Y. Zhou *et al.*, "The sharing architecture: Sub-core configurability for iaas clouds," ser. ASPLOS. New York, NY, USA: ACM, 2014, pp. 559–574.

[26] Y. Fu *et al.*, "PriME: A parallel and distributed simulator for thousand-core chips," in *ISPASS*, March 2014, pp. 116–125.

[27] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[28] N. Binkert *et al.*, "The GEM5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[29] M. Avriel, *Nonlinear programming : analysis and methods*. Mineola, NY: Dover Publications, 2003.

[30] J. Snyman, *Practical mathematical optimization an introduction to basic optimization theory and classical and new gradient-based algorithms*. New York: Springer, 2005.

[31] D. Yoon *et al.*, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," ser. ISCA. New York, NY, USA: ACM, 2011, pp. 295–306.

[32] A. Udipi *et al.*, "Rethinking dram design and organization for energy-constrained multi-cores," ser. ISCA. New York, NY, USA: ACM, 2010, pp. 175–186.

[33] S. Kumar *et al.*, "Exploiting spatial locality in data caches using spatial footprints," in *ISCA*, 1998, pp. 357–368.

[34] G. Pekhimenko *et al.*, "Base-delta-immediate compression: Practical data compression for on-chip caches," ser. PACT, 2012, pp. 377–388.

[35] L. Zhang *et al.*, "The impulse memory controller," *IEEE Trans. Comput.*, vol. 50, no. 11, pp. 1117–1132, Nov. 2001.

[36] I. Schoinas *et al.*, "Fine-grain access control for distributed shared memory," ser. ASPLOS VI, 1994, pp. 297–306.

[37] S. Rixner *et al.*, "Memory access scheduling," ser. ISCA. New York, NY, USA: ACM, 2000, pp. 128–138.

[38] K. Nesbit *et al.*, "Fair queuing memory systems," ser. MICRO, 2006, pp. 208–222.

[39] N. Rafique *et al.*, "Effective management of dram bandwidth in multicore processors," in *PACT*, 2007, pp. 245–258.

[40] O. Mutlu *et al.*, "Stall-time fair memory access scheduling for chip multiprocessors," ser. MICRO, 2007, pp. 146–160.

[41] S. Muralidhara *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," ser. MICRO, 2011, pp. 374–385.

[42] M. Caccamo *et al.*, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," ser. RTAS. Washington, DC, USA: IEEE Computer Society, 2013, pp. 55–64.

[43] A. Herdrich *et al.*, "Rate-based qos techniques for cache/memory in cmp platforms," ser. ICS. New York, NY, USA: ACM, 2009, pp. 479–488.

[44] S. Menghao *et al.*, "Efficiency-aware qos dram scheduler," in *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on*, 2009, pp. 223–226.

[45] M. Jeong *et al.*, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," ser. DAC, 2012, pp. 850–855.

[46] G. Tesauro *et al.*, "Online resource allocation using decompositional reinforcement learning," ser. AAAI, 2005, pp. 886–891.

[47] D. Vengerov *et al.*, "A reinforcement learning framework for dynamic resource allocation: First results." in *ICAC*, 2005, pp. 339–340.

[48] E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008, pp. 39–50.

[49] M. Butto *et al.*, "Effectiveness of the 'leaky bucket' policing mechanism in atm networks," *IEEE J.Sel. A. Commun.*, vol. 9, no. 3, pp. 335–342, Sep. 2006.

[50] J. Kidambi *et al.*, "Dynamic token bucket (dtb): A fair bandwidth allocation algorithm for high-speed networks," *J. High Speed Netw.*, vol. 9, no. 2, pp. 67–87, Oct. 2000.