DiME: A Performance Emulator for Disaggregated Memory Architectures

Dhantu Buragohain, Abhishek Ghogare, Trishal Patel, Mythili Vutukuru, Purushottam Kulkarni
Department of Computer Science and Engineering, Indian Institute of Technology Bombay, India
dhantu,abhishekvg,trishal,mythili,puru@cse.iitb.ac.in

ABSTRACT

Resource disaggregation is a new design paradigm for data center servers, where the compute, memory, storage, and I/O resources of servers are disaggregated and connected by a high-speed interconnect network. Resource disaggregation, and memory disaggregation in particular, can have significant impact on application performance, due to the increased latency in accessing a portion of the system's memory remotely. While applications need to be redesigned and optimized to work well on these new architectures, the unavailability of commodity disaggregated memory hardware makes it difficult to evaluate any such optimizations. To address this issue, our work develops DiME, an emulator for disaggregated memory systems. Our tool can emulate different access latencies over different parts of an application's memory image as specified by the user. We evaluate our tool extensively using popular datacenter workloads to demonstrate its efficacy and usefulness, and show that it outperforms previous emulators in its ability to emulate different access delays at a fine per-page granularity.

ACM Reference format:
Dhantu Buragohain, Abhishek Ghogare, Trishal Patel, Mythili Vutukuru, Purushottam Kulkarni.2017. DiME: A Performance Emulator for Disaggregated Memory Architectures. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 8 pages.
DOI: 10.1145/3124680.3124731

1 INTRODUCTION

Resource disaggregation is a new server architecture design paradigm that has recently been gaining attention both in academia [11, 15] and industry [4–6, 9]. Traditional servers consist of a fixed set of resources—compute, memory, I/O and storage—tightly integrated into a single box. This tight integration makes it difficult for applications to share resources across multiple servers, leading to a fragmentation of resources and inefficient utilization of some resources. For example, most of the memory on a server running a CPU-intensive application may be unutilized, as no new application can be provisioned to consume the free memory due to the unavailability of CPU cycles. Resource disaggregation breaks this limitation by disaggregating server resources into separate components. Each resource can be made available as a global pool of independent resource containers, and the physical containers are all connected by a high-speed backplane network. A disaggregated architecture can enable fine-grained resource provisioning and independent scaling of resources, leading to better resource utilization and eventually, lower costs of data center infrastructure. Figure 1 shows a high-level overview of a futuristic disaggregated data center.

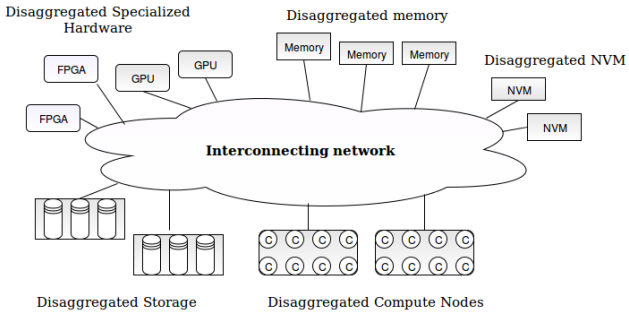


Figure 1: Resource Disaggregated Architecture.

Of all resources, storage has been one of the first ones to be disaggregated. A few recent papers [19, 20] looked at the disaggregation of flash storage. Their findings show that, with some system software support, disaggregation of flash storage is possible using the existing networking technologies for the interconnection backplane, without any performance degradation to applications. However, disaggregation of memory is much more challenging—memory access latencies being at least two orders of magnitude faster

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APSys '17, Mumbai, India
© 2017 ACM. 978-1-4503-5197-3/17/09…$15.00
DOI: 10.1145/3124680.3124731

than storage. Further, memory disaggregation is particularly important since the performance of memory-intensive applications depend on having access to larger amounts of disaggregated memory which can enable the working set size of these applications to reside fully in memory. However, the main challenge of memory disaggregation comes from the fact that, while memory access latency is in the order of few nanoseconds in traditional servers, disaggregating memory from compute will increase this latency to a few microseconds using existing modern networking technologies (e.g., RDMA). This increased access latency from compute to memory can significantly impact application performance. To alleviate this problem, a partial memory disaggregation model is considered a reasonable middleground [21]. Instead of completely disaggregating the memory from the CPU, a small portion of memory is kept local to each compute node. We will refer to the memory collocated with the compute as local memory, and the accessible memory from the disaggregated pool as remote memory. Prior work [16] has shown that with a 25% of local memory and 5μs end-to-end latency, memory disaggregation degrades application performance by only 5% for certain class of application(e.g., Hadoop, Graphlab, Memcached), even when using existing networking technologies in the interconnect.

In a disaggregated memory architecture, application performance will depend on the access latency of remote memory, the bandwidth of the interconnection network, the fraction of memory available locally, and on the placement of application data across local and remote memory, among other things. Prior work [16] has primarily studied how the characteristics of the interconnect and the amount of local memory available impact application performance. However, questions such as how applications must be designed to leverage this new architecture, and how an application must partition its data structures between local and remote memory (for a given access latency and interconnect bandwidth) to optimize performance, have received little attention. Note that applications can influence data placement in one of two ways. One option is to make the applications aware of the disaggregated nature of the memory hardware, and expose APIs to place code and data in local or remote memories. This option allows applications to be redesigned to efficiently use the disaggregated memory setup, and intelligently partition data between local and remote memories in a way that the overhead of remote memory access is reduced. For example, [14] optimizes data placement for applications over a tiered memory hierarchy with different access latencies (e.g., DRAM and NVRAM), and similar solutions can be developed for disaggregated memory as well. Alternately, memory disaggregation can be made transparent to the applications with the help of operating system and system software support. In this case, applications are not aware of the disaggregated architecture, and a system software stack performs the necessary optimizations to manage the disaggregated resources. In either option, data placement significantly impacts application performance. However, in the absence of a hardware prototype of disaggregated memory, solutions to such problems are hard to implement and validate. While prior work [16] has developed tools to emulate remote memory, such tools only work to emulate differential access latencies over a part of the memory of an entire system and do not provide fine-grained control at the process or application level. Therefore, it becomes difficult to implement or evaluate application design ideas that require fine-grained control over access latency of different parts of an application memory using such coarse grained emulators.

Our work describes the design and implementation of DiME, an emulator for disaggregated memory architectures. DiME works at the granularity of individual pages of processes and can emulate different access latencies for different parts of an application's memory image. A certain user-specified fraction of the application's memory can be accessed normally as local memory, while the rest is accessed with an added delay. The remote access delay is modeled as a function of the access latency of the remote memory and the interconnect bandwidth. Remote memory emulation in DiME is done completely in software. Our tool works as a Linux kernel module, protects and traps page accesses of applications under emulation, and injects delays as appropriate. We ran extensive experiments with our emulator using simple micro-benchmarks and data center application workloads [7, 8]. Our results show that DiME can accurately inject the specified delay over memory accesses with little extra overhead, and the results of application performance degradation observed with remote memory emulation match what is to be expected. We also demonstrate that DiME provides fine-grained control over injecting delays on specific pages of applications and can provide more deterministic results than previous coarse grained emulators.

Note that our emulator is generic enough to study application performance over any system with differentiated access latencies for different parts of memory (e.g., systems with a hierarchy of heterogeneous memories), beyond just disaggregated memory architectures. We believe that tools such as DiME will spur research into new designs for applications and system software that can efficiently leverage newer memory architectures. The rest of the paper is organized as follows. Section 2 explains the design and implementation of DiME. Section 3 evaluates our emulator. Section 4 discusses related work, and Section 5 concludes the paper.

2 DESIGN AND IMPLEMENTATION

We now describe the design and implementation of DiME. We begin with the assumptions we make in our work.

2.1 Assumptions

Given the absence of a widely available prototype for memory disaggregation, we make a few assumptions about the hardware and system architecture in our work, much like those made in prior work [16]. We assume that there will be partial CPU-memory disaggregation, where a small portion of memory will be kept local to each compute node. Access to this local memory will not incur any extra delay, and user-specified delays will only be added on access to the remote disaggregated memory. The disaggregation of memory can either be at a rack scale (with different resource blades placed in the same rack), or across the entire data center, and our emulator is agnostic to the scope of disaggregation. We assume that the memory access between the CPU and remote memory is at the granularity of pages, and we do not consider the details of the virtual memory design that can enable the addressing of this remote memory. Our emulator injects a delay corresponding to the user-specified access delay and transmission bandwidth over the interconnect between CPU and memory. We do not consider queuing delays over the interconnect since there is not enough literature about the type of networking technologies that will be used and the interconnect network architecture. Prior work [22] has proposed queuing models for rack scale data serving systems. We propose to incorporate such models to emulate queueing delay of the interconnect in the future.

2.2 Emulating Remote Memory

The key idea of our emulator is to protect portions of the virtual address space of processes under emulation that correspond to remote disaggregated memory, and inject delays when the protected pages are accessed and trapped to the kernel. Note that emulation is done on a per-process basis for fine-grained control, and multiple processes can be emulated simultaneously. The address space of processes corresponding to local memory is accessed normally without traps. The user of the emulator specifies the amount of memory to be kept local, and pages allocated beyond this limit are considered to be served from remote memory. We implement our emulator as a loadable Linux kernel module, with a few lines of kernel code modification. The basic architecture of our emulator is shown in Figure 2. We now describe the design and implementation of our module in more detail.

Inducing a page fault. In Linux, a page fault can be induced for any future access to a page by using the page protection bits of each page table entry. Setting the flag _PAGE_PRESENT to 0, and the flag _PAGE_PROTNONE to 1 in the page table entry (PTE) of a page induces a page fault. We will refer to setting this combination of flags as *protecting a page* and setting the flags _PAGE_PRESENT to 1 and _PAGE_PROTNONE to 0 as unprotecting the corresponding page. In DiME, we protect a portion of the virtual address space of a process that corresponds to remote memory, so that any memory access to these pages will raise a page fault. A modified page fault routine helps the kernel module take necessary actions, such as evicting a local page, emulating delay, and so on. No page faults are generated for accesses to *local* memory pages.

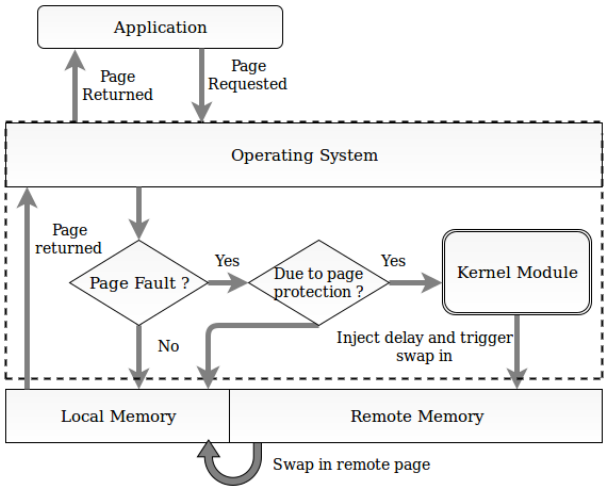


Figure 2: Flowchart of operations within DiME.

Page fault hooks. The Linux kernel function `do_page_fault` handles a raised page fault. `do_page_fault` in turn calls `__do_page_fault`, which is the actual page fault handler. Linux kernel provides the kprobe [1] mechanism to intercept an exported kernel function from a kernel module without modification to the kernel source. The `do_page_fault` function is protected from kprobe to avoid recursive calls from the probe handler. Therefore, we used two callback functions in the linux fault handler `fault.c` file to inject software created delay, one just before `__do_page_fault` and one just after it. These callback functions are defined in the kernel module of DiME. The first callback function performs bookkeeping of the local memory pages and also stores the start timer value `start_time` using inbuilt kernel `sched_clock` function. The second callback function injects the remaining delay after `__do_page_fault` function returns. The second callback function calculates the remaining delay by subtracting `start_time` from the `emulated_latency` where `emulated_latency` denotes the remote memory access latency that the user wants to emulate.

Delay injection. Linux kernel provides different ways to introduce a delay in an execution flow, viz. `delay` and `sleep` function families [2]. The `sleep` function family is backed by high-resolution kernel timers subsystem. The `usleep_range` function takes a time range and schedules a `wakeup` event anywhere in the range before sleeping. As an optimization,

it looks for an existing event scheduled in a given range and uses it to schedule a wakeup. If an existing wakeup event is not found, a new wakeup event is scheduled at the upper bound of the time range. Due to this involvement of the wakeup scheduling and interrupt mechanism, it cannot be used for injecting small delays (e.g., a few hundred nanoseconds) precisely. The delay function family is backed by a busy-wait loop which executes the desired delay by iterating a single instruction which approximately takes 1μs. Since the execution time of this instruction is not exactly 1μs, but has some positive error, the error gets accumulated for a large number of iterations. Therefore, the delay injection error is less for delays close to 1μs, but increases linearly for higher delays. Because of the limitations of the above two techniques, DiME uses a custom busy-wait implementation backed by sched_clock function to enforce delays. The sched_clock function uses the system jiffy counter internally. The module checks the timestamp using sched_clock function and iterates in a loop until the elapsed time is less than the delay to be executed. We used a custom busy-wait loop as our delay injection method since it provides the highest accuracy with a constant small error for a broad range of delays. We compare the accuracy of all delay injection methods in Section 3.

Page allocation. DiME maintains a list of pages of a process that are considered local and keeps track of the local memory usage. When the application process requests a page allocation, the kernel modules first checks whether the request can be served from the emulated local memory. If there is enough space left in local memory then the operating system will allocate a page and return to the application. If our module deems that there is no space left in the emulated local memory, then a page has to be replaced from the emulated local memory to create an empty slot for the new page allocation. We are currently using a first-in-first-out (FIFO) replacement policy to evict a page from local memory and tag it as a remote page using page protection bits for future accesses. The newly requested page is then allocated as a local page. We are working on the implementation of additional page replacement policies(e.g., LRU) which will give the users the flexibility to choose among the different replacement policies.

Processing Read/Write requests. Whenever a process accesses a memory location that is protected and emulated as remote memory, the MMU raises a page fault and traps to the OS. We treat all page faults as an access to remote memory. The page fault handler then unprotects the page, evicts a page from the local page list based on the FIFO eviction policy (if the local memory is full), and adds the faulted page to the local page list. The page fault handler then injects additional software delay to emulate the remote memory access latency (propagation and transmission delay). To maintain the size of the local memory constant, we protect the evicted page by setting the PTE flags. This operation mimics the swapping of pages between local and remote memory. Note that the MMU will not raise a page fault for local memory access since all local memory pages are unprotected. Therefore, the performance of local memory access will not be affected by our kernel module in any way.

Delay model. DiME takes as inputs the one-way remote memory access latency δ, the network bandwidth B of the interconnect between the compute nodes and remote memory, and page size P used in the system. Since we assume a page-level remote memory access and no queuing delays, the delay injected on every remote page access is computed as

$$delay = 2 \times \delta + \frac{P}{B} \quad (1)$$

We assume that the eviction of the local page and the fetching of the remote page happen in parallel. Due to the previously mentioned constraints, we are unable to model the queuing delay on the network. Once a queuing delay model is formulated for such architectures, it can be very easily integrated with our emulator.

2.3 User Interface of DiME

We have released our source code of DiME in order to enable other researchers to use and extend our work [3]. Our tool currently takes a configuration file as input, which contains the following parameters that the user can modify: remote memory access latency, interconnect bandwidth, and local memory size. DiME internally calculates the emulated delay using these parameters. We are in the process of modifying DiME to provide a better interface for the users. We are designing an abstract interface layer for DiME similar to some of the Linux kernel interface (e.g., VFS layer), where a few function pointers will be exposed to the users. The abstract interface will provide the flexibility to the users to implement their own replacement policy and delay functions using the function pointers to test their workloads.

3 EVALUATION

In this section, we evaluate our emulator to answer the following questions:

- How accurately can DiME emulate remote memory access latency, and what is the overhead of this emulation?
- Can DiME be used to emulate real applications in realistic scenarios?
- How does DiME compare with other emulators in terms of accuracy and feature set?

We designed custom microbenchmarks to evaluate the accuracy and overhead of our emulator and used a set of popular real world applications (Memcached [7] and Redis [8]) for

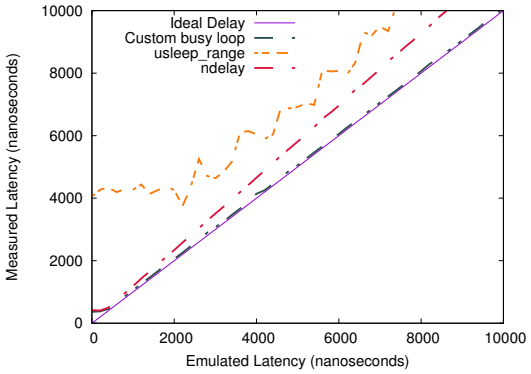


Figure 3: Accuracy of delay injection methods.

	Without DiME	With DiME
Protected page	160 ns	241 ns
Demand paging	704 ns	780 ns

Table 1: Time required to handle page faults.

performance characterization and comparisons. All experiments were performed on an Intel Xeon CPU E5-2650 server with 16 hyper-threaded and CPUs operating at 2.60 GHz. Each application was executed from within a Linux/KVM virtual machine with 6 vCPUs and 6 GB RAM running Linux kernel 4.4.70.

3.1 Accuracy and Overhead of DiME

Accuracy. We now compare the various delay injection methods we considered in the design of DiME: the `ndelay` and `usleep_range` methods in the Linux kernel, and our custom busy-wait function, as described in Section 2.

We used linux `Read Time-Stamp Counter(rdtsc)` function to measure the actual delay that DiME has emulated. Figure 3 shows the various methods of delay emulation that we considered. The x-axis plots the delay to be emulated and the y-axis shows the actual delay measured in the page fault handler. From the graph, it can be seen that custom busy-wait loop provides the highest accuracy and the mean error is 68 nanoseconds. Therefore, we use our custom busy-wait loop for injecting delays for remote memory access emulation in DiME.

Overhead. To measure the page fault handling overhead of DiME , we ran a program to allocate and access pages with and without DiME and measured the time required to handle page faults. To measure the execution time of a single page fault on the x86 architecture, we used the Time Stamp Counter(TSC) [10]. Page faults can occur either due to demand paging or our page protection mechanism required for emulation. Table 1 shows the measured page fault handling durations for both these cases. We see from the table that DiME has a mean overhead of 78 nanoseconds per page fault, which is quite small compared to remote memory access latencies.

3.2 Performance of real applications

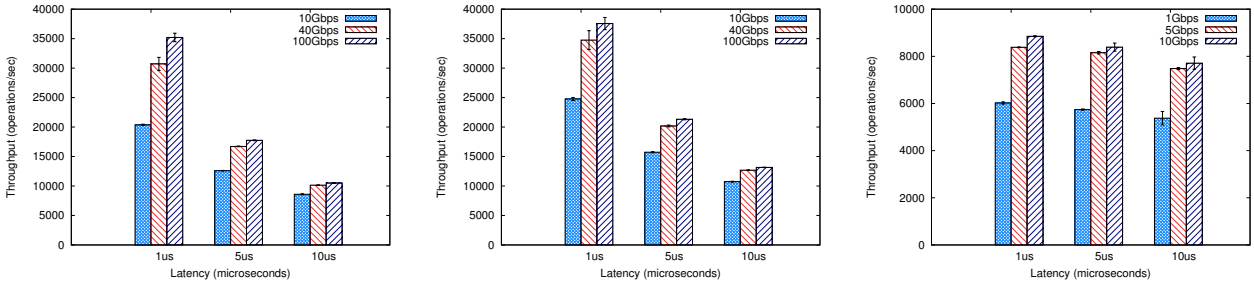
We executed two popular key-value stores, Memcached [7] and Redis [8] with DiME to study their performances in various latency, bandwidth, and local memory size configurations. We used the YCSB [12] benchmark for measuring the performance of these key-value stores. In all experiments, we use a key-value dataset of size 2 GB at the server, and performed a series of read and write operations on this data.

Figure 4 shows how the throughput of an application changes with emulated memory parameters like access latency and interconnect bandwidth. We can see from the figure that the throughput of both applications (Redis and Memcached) degrades monotonically as we increase the remote memory access latency and decrease in network bandwidth. We also see from the figure that, for a fixed network bandwidth, the increase in local memory has a greater positive impact on performance in setups with lower interconnect bandwidths. For example, for a 1μs remote access delay, the Redis throughput changed from 20000 ops/sec (for 20% local memory) to 25000 ops/sec (for 40% local memory) on a 10 Gbps interconnect. The corresponding numbers for 100 Gbps did not change by the same factor. These evaluations prove that our emulator is capable of executing large applications in realistic scenarios and that the performance of emulated applications changes as expected with changes in emulated parameters.

3.3 Comparison with other emulators

Existing emulators for disaggregated memory (e.g., the one developed in [16]) emulate memory disaggregation at the system granularity. In other words, the entire system memory is partitioned into local and emulated remote memory, and an application has no control over which portions of its address space are emulated as remote. In contrast, DiME emulates remote memory at the granularity of individual applications. We now compare DiME with a system-wide emulator used in [16] to contrast the differences in the two types of emulation.

Our experimental setup consists of two applications, Redis and Memcached, running together on the same machine. When running with DiME, we emulated remote memory in Redis by varying the local memory size, and Memcached ran as a separate application outside our emulator. There was enough RAM to satisfy the local memory requirements of both applications. When using the system-wide emulator, both the Redis and Memcached applications shared the emulated local memory. To make the comparison fair, the



(a) Redis with 20% Local Memory . **(b) Redis with 40% Local Memory.** **(c) Memcached with 30% Local Memory.**

Figure 4: Evaluation of Redis and Memcached with varying latency and bandwidth.

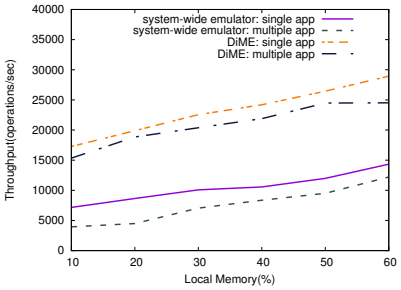
local memory allocated in the system-wide emulator test was twice of that used in the experiment with DiME, to account for the fact that two applications were now using the local memory. With the system-wide emulator, both the applications first used the common local memory and the memory pressure resulted in using the emulated remote memory. The total memory available in the system was 8 GB, the remote access latency was set to 5μs and the network bandwidth was emulated to be 100 Gbps. We vary the fraction of local memory available in the emulator from 10% to 60% of the workload size and measure the performance of the Redis application, both when Redis was running in isolation and when it was run together with Memcached. The results of the comparison are shown in Figure 5.

We see from Figure 5a that the Redis application has a higher absolute throughput when emulated with DiME as compared to when using the system-wide emulator, both when running in isolation and when running with Memcached. We attribute the primary reason for this to be the overhead of using the swapping mechanism to emulate remote memory accesses in the system-wide emulator. On the other hand, DiME only needs to toggle bits in the page table entries to move a page from emulated remote memory to local memory, which is much faster than executing the swap related operations, even if the swap device itself is emulated in memory. We measured the average overhead of tagging local and remote memory pages to be around 240 ns, while the reported overhead of swapping-related operations is 2.46μs [16]. It is worth mentioning that it is important to emulate the swapping overhead only if we are interested in the absolute value of the application performance. But since we are more interested in the relative performance of the applications under different configurations, we can achieve a lower overhead of emulation in DiME.

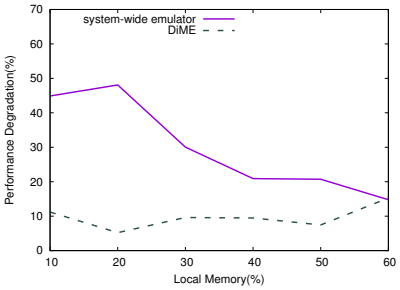
We now study the impact of running multiple applications together in DiME and the system-wide emulator. Figure 5b shows the relative performance degradation of Redis when another application, Memcached, was running simultaneously on the same machine, as compared to when it was running in isolation. Ideally, we expect the performance of an application to only depend on the fraction of its local memory, and not on whether another application is running on the same machine. That is, we expect the relative performance degradation to be always zero. Redis suffers an approximately 10% degradation in performance with DiME when the Memcached application was collocated, possible due to some interference when sharing CPU resources. On the other hand, the impact of a collocated application is much more in the case of the system-wide emulator, and the performance of Redis degrades by as much as 50% when another application starts to share the emulated memory. This can be explained by the fact that with the system-wide emulator, all applications contend for the local memory and deterministic provisioning of local memory is not possible. On the other hand, DiME allows per-application configuration of local memory and provides a much better isolation of the same across applications. This experiment highlights the fact that DiME can be used to model application behavior more accurately with memory disaggregation, due to its fine-grained, per-process, and low overhead emulation of remote memory.

4 RELATED WORK

Memory Disaggregation. Among the recent research in the area of memory disaggregation [11, 16–18, 23], Gao et al. [16] and Han et al. [18] developed performance emulators to study application performance on memory disaggregated architectures. To emulate the remote memory, they divide the physical memory into two parts using an in memory swap partition. A special swap device driver intercepts every swap request and adds artificial delay to emulate remote memory latency. While the basic concept of DiME is similar, we divide memory into local and remote at the level of the virtual address space of individual processes. This allows us to control the page replacement policy at the application level,



(a) Throughput of Redis.



(b) Performance degradation of Redis when collocated with Memcached.

Figure 5: Comparison of application throughput of Redis with DiME and a system-wide emulator.

rather than leaving it to the operating system to identify and handle remote memory delay emulation. Also, unlike previous memory disaggregated emulators, our emulator provides better programmability and can be easily extended to provide different APIs to the user for data placement.

Persistent Memory Emulation. Some recent work on NVM emulation is also similar in functionality to our emulator, and NVM is emulated over DRAM by injecting additional software delays. Dulloor et al. [13] developed an NVM emulator by injecting delays on memory access using special hardware support. Similarly, Volos et al. [24, 25] also developed a persistent memory emulator which uses hardware performance counters to accurately inject delays on every memory access. However, both these emulators require special hardware (multi-socket processors) for emulating hybrid memory (DRAM and NVM) on the same system. In contrast, our emulator does not need hardware support for emulation and hence can be used on any machine.

5 CONCLUSION

In this paper, we presented DiME , a performance emulator for disaggregated memory architectures. We showed that DiME provides good accuracy and low overhead in remote memory emulation and that it is capable of emulating a range of remote memory access latencies. We also show that it is capable of emulating large real applications in the disaggregated memory architectures, and can be used to draw accurate conclusions on how these applications would be effected by memory disaggregation. DiME allows for fine-grained partitioning and provisioning of emulated memory at a process granularity, and provides memory isolation between emulated applications.

DiME is a work in progress and our present implementation has a few limitations that we are addressing as part of ongoing work. The current version of DiME is capable of supporting emulation for only one application at a time. We are updating DiME to enable supporting multiple applications that share the emulated and local memories. We are also updating our code to support other replacement algorithms like LRU, in addition to the existing FIFO algorithm swap out local memory. Because the disaggregated memory architecture is still in its infancy, we have made several assumptions in the design and implementation of our system, and we will continue to add more features as the architecture evolves.

REFERENCES

[1] An introduction to KProbes. https://lwn.net/Articles/132196/.
[2] delays - information on the various kernel delay / sleep mechanism. https://www.kernel.org/doc/documentation/timers/timers-howto.txt.
[3] DiME source code repository: https://github.com/networkedsystemsIITB/DiME.
[4] Facebook Disaggregated Rack. https://goo.gl/WQcg15.
[5] HP The Machine. https://goo.gl/ugImWf.
[6] Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://intel.ly/1SUMgP1.
[7] memcached - a distributed memory object caching system. https://memcached.org/.
[8] Redis - an in-memory data structure store. https://redis.io/.
[9] seamicro technology overview. https://goo.gl/bePyYT.
[10] Time stamp counter. https://goo.gl/v5gprq.
[11] K. Asanovic and D. Patterson. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In USENIX FAST(2014).
[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of the SoCC '10.*
[13] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proc. of the EuroSys '14.*
[14] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *Proc. of the EuroSys '16.*
[15] K. K. et al. Rack-scale Disaggregated cloud data centers: The dReDBox project vision. In DATE(2016).
[16] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Proc. of the OSDI'16.*
[17] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17).*

[18] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proc. of the HotNets-XII*.

[19] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash storage disaggregation. In *Proc. of the EuroSys '16*.

[20] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash ≈ Local Flash. In *Proc. of the ASPLOS '17*.

[21] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proc. of the ISCA '09*.

[22] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The case for rackout: Scalable data serving using rack-scale systems. In *Proc. of the SoCC '16*.

[23] P. S. Rao and G. Porter. Is memory disaggregation feasible?: A case study with spark sql. In *Proc. of the ANCS '16*.

[24] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proc. of the Middleware '15*.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of the ASPLOS XVI*.