DiME: A Performance Emulator for Disaggregated Memory Architectures

Master's Thesis Phase I
Submitted in partial fulfillment of
the requirements for the degree of
Master of Technology
by

Abhishek Ghogare
(Roll No. 153059006)

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

October 2017

Acceptance Certificate

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

The master's thesis entitled "DiME: A Performance Emulator for Disaggregated Memory Architectures" submitted by Abhishek Ghogare (Roll No. 153059006) may be accepted for being evaluated.

Date: October 2017

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Abhishek Ghogare
(Roll No. 153059006)

Date: 14 October 2017

List of Publications

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
Git Repository: https://github.com/networkedsystemsIITB/DiME [4]

Abstract

Disaggregating various resources is the recent trend in data center architectures, where resources of same type are disaggregated in resource pools, e.g. compute, storage, network and memory. Disaggregation helps in efficient use of power and resources by sharing among servers based on their usage patterns. Since disaggregated resources are shared over external communication link, it can have significant impact on application performance based on the access pattern. Due to unavailability of disaggregated memory hardware, it is difficult for application developers to evaluate the optimizations. We aim to emulate disaggregated memory for a process, so that application developers can use the emulator in development process for testing purpose to make application more efficient in disaggregated environment. We develop a tool, DiME, to have more fine-grained control over emulator configurations compared to previously proposed emulators and show its features and usefulness by extensive testing using popular workloads.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional data centers are built of collection of servers, and each server has dedicated resources, which are memory, cpu, non-volatile storage (SSD, HDD), network, etc. This tight coupling of resources makes it difficult to manage and use resources efficiently. For example, a memory intensive application running on a server cannot make use of memory on another server which is not being used, resulting overall under utilization of memory in data center. Also, per-server memory demands are increasing due to memory intensive applications, leading to more power to operate the memory. A new memory architecture is required to efficiently use the available memory. By disaggregating resources, multiple compute blades can connect to a single global pool of independent resource and dynamically share its capacity over high speed backplane network, resulting in decreased memory requirement, increased utilization and lower power consumption. Disaggregated architecture also helps scale and maintain each resource independently.

1.1 Motivation

The reason that the access latency of memory in traditional servers is in few nanoseconds, whereas it is few microseconds in case of disaggregated architecture, makes it reasonable to consider partial disaggregation. Each compute node owns a small portion of memory at local, instead of completely decoupling memory with compute node. The performance drop in case of disaggregated memory will depend upon relative location of remote memory, bandwidth of network, size of local memory and access pattern of the application. Applications can make use of knowledge of disaggregated memory by carefully placing the data in remote or local memory based on the requirement to improve performance. It is also possible to make the disaggregation transparent to applications and make operating system take the responsibility of data placement. In any case, we need to verify the efficiency of method on hardware prototype. Since disaggregated memory is still in open research, the non availability of hardware prototype makes implementation and validation of placement techniques difficult.

Earlier work propose emulators which lack fine-grained control over emulator features e.g. page based memory placement, local memory management algorithms, etc. Our tool, DiME, works as a Linux kernel module, traps accesses to remote memory and injects appropriate delay based on emulator configuration and also provide users flexibility to implement their own local memory management policy. In addition to features, flexibility and emulation accuracy, DiME also provide low emulator overhead compared to other existing emulators. We evaluate our emulator with memory intensive applications such as redis and memcached, and show that the results are expected based on specified configuration. We also demonstrate how DiME provide fine-grained control over delay injection based on specific accessed page.

1.2 Contribution

This project is the extension of R&D project done by me and my friend Trishal Patel. We had developed a basic prototype of the emulator as proof of concept.

The following contribution have been made by me towards the completion of this project:

- Design and implementation of multiple emulator instances, modularization of page replacement policies, procfs based dynamic configuration, process tracker feature
- Explored and evaluated different methods to introduce delays in execution flow
- Explored impact of CPU cache and TLB on emulation accuracy, and patched emulator
- Verified emulator correctness and accuracy by evaluating multiple test cases

Chapter 2

Architecture Assumptions

Since memory disaggregation architecture is an open research topic, we make the following architecture assumptions. We assume a partial disaggregation model where each compute node has a small amount of local memory. Access to this local memory will not incur any extra delay. Rest of the memory is located on remote memory "blade" located anywhere across the data center. The granularity of the remote memory is in pages of size 4kB which is default page size in Linux. A remote memory is requested though standard hardware protocol whenever compute node needs to access remote memory. High speed network interconnection is used to setup communication between devices. A local page is evicted based on the eviction policy. Then the requested remote page is transferred over the network to local memory, incurring a network delay. Figure 2.1 shows a high level memory flow when remote memory is accessed.

We assume that the same memory is not shared across multiple compute nodes at the same time, so there will not be local memory coherency issue to be handled. 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.

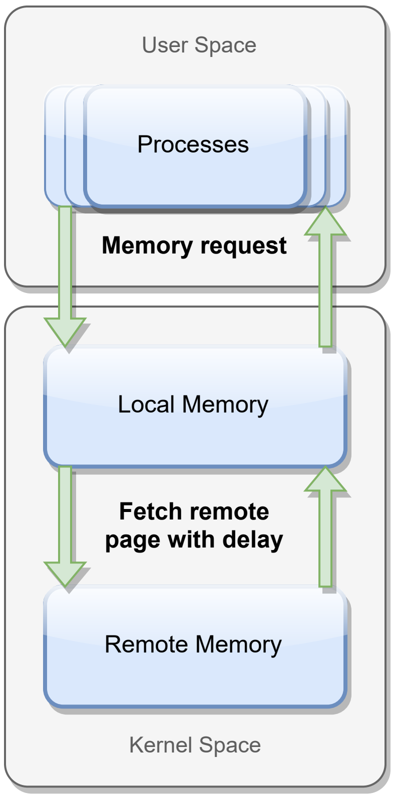


Figure 2.1: Disaggregated memory flow.

Chapter 3

Linux Kernel Background

3.1 Memory Management in Linux

Linux adopts a paging model to manage memory. Physical memory is divided into pages for more efficiency. A separate page table is maintained for each process which contains page table entries (PTE). PTE stores the actual physical address of the page along with various flags (Table 3.1). PTE of a linear address is extracted from page table by doing a page table walk. Since these translations of linear address to physical address are very time expensive tasks, they are kept in translation lookaside buffer (TLB). In x86 architecture, the page table walk is done by special hardware, memory management unit (MMU). When a TLB miss occurs, the MMU has to populate the TLB entry by doing page table walk. In some architectures where MMU is not present, a page fault is generated in case of TLB miss and kernel page fault handler has to do the translation and update TLB entries. Figure 3.1 shows the memory access flow. A page fault is raised in case of various scenarios where it needs special handling, e.g. PTE not present, invalid page access, invalid PTE, etc.

Since module has to inject the delay at a memory access, page fault handler is the place where we can put the delay logic. When MMU cannot serve the address translation request, a page fault is raised. CPU breaks the normal program execution flow and handles the raised fault. In x86 architecture, function named "`do_page_fault`" in file "arch/x86/mm/fault.c" in kernel source code handles the page fault.

There are various reasons for triggering a page fault. We are going to focus on following reasons:

1. Page does not exists, i.e. first reference to the address within the page: demand paging

2. Page is present in memory and protection bit is set in PTE

Table 3.1: PTE flags.

Bit	Function
`_PAGE_PRESENT`	Page is resident in memory and not swapped out
`_PAGE_PROTNONE`	Page is resident but not accessible
`_PAGE_RW`	Set if the page may be written to
`_PAGE_USER`	Set if the page is accessible from user space
`_PAGE_DIRTY`	Set if the page is written to
`_PAGE_ACCESSED`	Set if the page is accessed

3. Page was swapped-out from memory

If MMU is able to translate the virtual address successfully, no fault is raised and program flow is not interrupted.

3.2 Types of Page Faults

3.2.1 Minor Page Faults

A minor page fault occurs only when a page table is updated (and the MMU configured) without actually needing to access the disk. This happens when a page is shared across processes but current process does not have access for it, e.g. when two or more users use a program at the same time, the text section pages are shared to increase efficiency.

Another example is copy on write. When a new process is forked, the child process shares the address space of parent process. A separate copy of the page is created only when a write operation happens on the shared page. Both of these page faults has less work to do and page fault occurs only once for that page.

3.2.2 Demand paging

In this case, the page is not present in physical memory and kernel should allocate the page or load the page from disk. The `_PAGE_PRESENT` flag is not set in PTE. So we can simply add these pages to local page list and execute delay. Two page faults are triggered for this case. First page fault starts loading and allocating the page and second page fault initializes PTE. The `_PAGE_PRESENT` bit is not set in first page fault, but it is set in second page fault. So by checking `_PAGE_PRESENT` only, we can serve multiple page faults for a page only once.

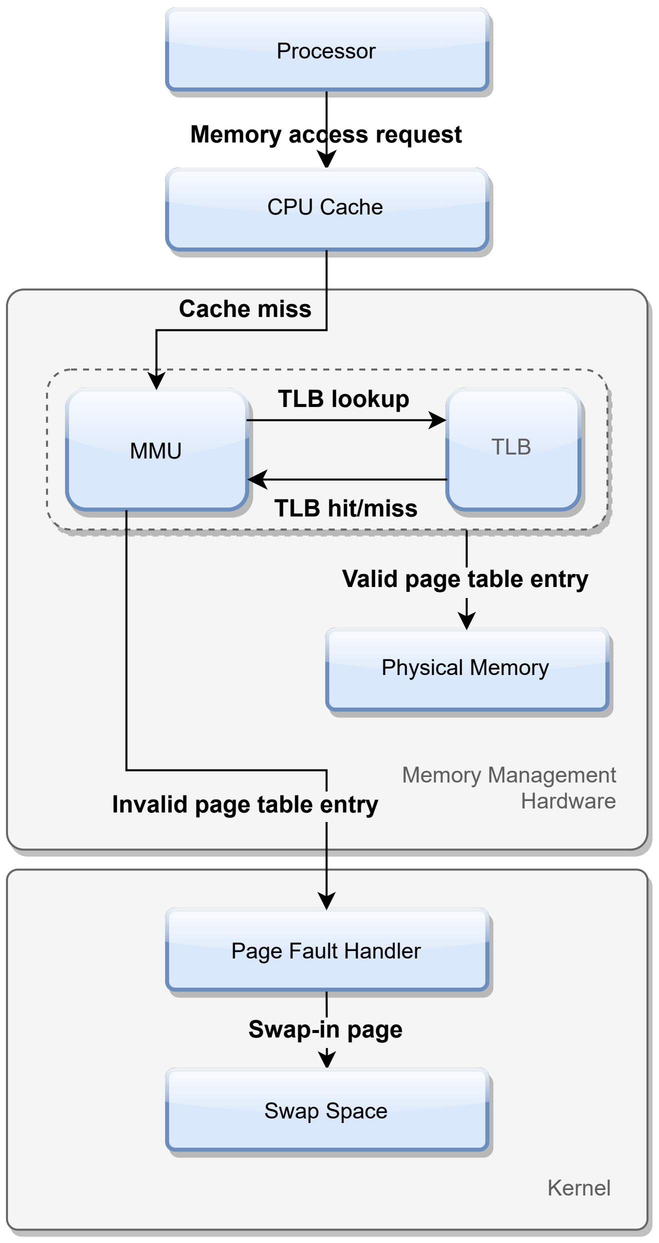


Figure 3.1: MMU and page fault handler interaction.

Table 3.2: PTE flags and corresponding type of page fault.

Bits		Type	Action
_PAGE_PRESENT	_PAGE_PROT_NONE		
0	0	Page not present in physical memory	Check page in local page list, add if not present and emulate delay
0	1	Page is present but protected	Add to local page list and emulate delay
1	0	Page is loaded in physical memory, but not initialized	Ignore page fault, since page is already in local page list

3.2.3 Swap-in Pages

When a page is swapped-in into the memory, its `_PAGE_PROT_NONE` and `_PAGE_PRESENT` flags are reset. So the module has no idea about whether the page was in local page list or not. The only way to find out that is to traverse the list. The size of the list could be very large depending on max number of allowed local pages. Same as demand paging, this type of page fault also triggers two page faults.

There are extra unused bits (`_PAGE_BIT_SOFTW{1,2,3,4}`) of PTE in x86 architecture. We tried setting these bits to represent if page is in local memory or remote memory, but same as `_PAGE_PROT_NONE` bit, they also gets cleared when page gets swapped-in.

Currently we are assuming the physical memory is large enough to accommodate the process and swap space is not used. Current module treats all pages as remote memory for which `_PAGE_PRESENT` is not set and injects the delay.

Chapter 4

Design and Implementation

4.1 Overall Design

DiME is built of two primary parts, a single core module and multiple page replacement policy modules. When a page fault gets triggered, the core module checks if the concerned process is running inside the emulator. The page fault information is then forwarded to the page replacement policy module which does the necessary bookkeeping and evicts a local page to make a spare slot available for the remote page. At the end, core module receives a signal whether to inject delay or not.

DiME takes a list of PIDs for which remote memory is to be emulated along with other configuration parameters, i.e. number of local pages, one-way latency, network bandwidth (Table 4.2). DiME logically divides the process memory into two parts: local and remote. DiME injects delay whenever a remote memory page is accessed. Delay is propagation and transmission delays to fetch the remote page. We will refer both combined as "delay" in rest of the report. Transmission delay is calculated based on network bandwidth and the size of the page that is being fetched.

4.2 Emulator Work-flow

To emulate remote memory, DiME protects a portion of virtual address space of processes under emulation which corresponds to remote memory so that when a protected page is accessed, it will trap to the page fault handler and delay will be injected. DiME supports multiple emulator instance feature, which enables different process groups to be emulated on separate remote memory, network architecture and page replacement policy simultaneously.

To inject the delay to each page fault, DiME follows the steps :

1. Intercept "`wake_up_new_task`" function calls to check if new process was forked by any of the tracking processes

2. Intercepts each page fault

3. Determine if the page fault is raised for a process for which the emulation is running

4. Determine if the faulted address is in the local memory or the remote memory

5. No delay injection if address belongs to local memory

6. If address is in remote memory, forward page fault information to page replacement policy module

   (a) If local memory pool is free, virtually allocate the new local page to the faulted page

   (b) Else virtually evict and protect a page from local memory pool and allocate the new freed slot to the faulted page

   (c) Instruct core module whether to inject delay or not

7. Inject the delay for emulating the transmission and propagation delays

4.3 Intercept Page Fault Handler

Module needs to intercept each page fault so that it can emulate the delay. There are two ways to intercept a kernel function:

1. KProbe

2. Kernel hooks

4.3.1 KProbe

KProbe [6] is a debugging mechanism provided by the kernel to track activities inside kernel. A kprobe can be registered to any function whose address is known at the time of kprobe insertion. Major benefit of using kprobes is that kernel recompilation is not needed and can be registered on live kernel through kernel module.

In Linux kernel, `do_page_fault` function has to read CR2 register which contains page fault linear address before any other tracing mechanism get called to properly handle the page fault. Therefore, `do_page_fault` is blacklisted from kprobe and tagged as notrace. Hence we are left with kernel hooks option to intercept page fault handler.

4.3.2 Kernel hooks

Adding a hook to the kernel function is another way of intercepting a function call, but with added overhead of kernel recompilation. We have added a hook function pointer in `fault.c` file initialized to null value. We are calling the hook function before `__do_page_fault` function in `do_page_fault` function. The hook function is called only if the function pointer is not null. The hook function pointer is exported so that the kernel module can set its value to pointer to custom function defined inside the module. Module sets the hook function pointer back to null at the time of module removal, to prevent invalid memory access.

Listing 4.1 shows how hook function pointers are declared, exported and called from page fault handler.

Listing 4.1: Hook functions in fault.c

```
int (*do_page_fault_hook_start)(struct pt_regs *regs,
                                unsigned long error_code,
                                unsigned long address) = NULL;
EXPORT_SYMBOL(do_page_fault_hook_start);

int (*do_page_fault_hook_end)(struct pt_regs *regs,
                              unsigned long error_code,
                              unsigned long address) = NULL;
EXPORT_SYMBOL(do_page_fault_hook_end);

dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code) {
    unsigned long address = read_cr2(); /* Get the faulting address */
    enum ctx_state prev_state;

    // Calling do_page_fault_hook_start hook function
    if(do_page_fault_hook_start != NULL) {
        do_page_fault_hook_start(regs, error_code, address);
    }

    prev_state = exception_enter();
    __do_page_fault(regs, error_code, address);
    exception_exit(prev_state);

    // Calling do_page_fault_hook_end hook function
    if(do_page_fault_hook_end != NULL) {
        do_page_fault_hook_end(regs, error_code, address);
    }
}
```

4.4 Induce Page Faults

In Linux, a page fault can be induced for any future access to the page by setting the `_PAGE_PROT_NONE` bit and resetting `_PAGE_PRESENT` bit of PTE of the required page. This combination of bits prevents page fault handler from behaving unexpectedly and still make MMU to raise a page fault. It suggests that the page is present in memory but access to it is protected. Setting the above combination of flags is referred as protecting the page and setting opposite combination of flags (i.e. `_PAGE_PROT_NONE`=0 & `_PAGE_PRESEN`=1) is referred as releasing the page in rest of the report for simplicity. DiME protects all pages of the emulating processes at initialization. This way every memory access by the given processes will trigger the page fault.

4.4.1 TLB & Cache Flush

In x86 architecture, it is specified that TLB cache are not coherent with page table (cache and TLB flush [1]). It is possible to exists stale translation entries in TLB after page table is altered. Therefore it is necessary to flush relevant TLB entries when page table entry is updated.

In general, when virtual to physical address mapping is changed or a page table entry is updated following actions are required to take:

1. Flush CPU cache
2. Update page table entry
3. Flush TLB

CPU cache flush is required to be first, because it helps to properly handle architectures which strictly requires valid address translation to exist for a virtual address before that address is flushed.

In DiME, it is required to flush TLB after a page is protected, because even if we protect a page in page table, it is possible to exist stale entry in TLB which would serve the address translation and page fault would not raise. We explain the test case where TLB flush is essential in section 5.1.2.

DiME first initializes the required data structures and sets up the hooks before protecting the pages so that it will not miss any page fault.

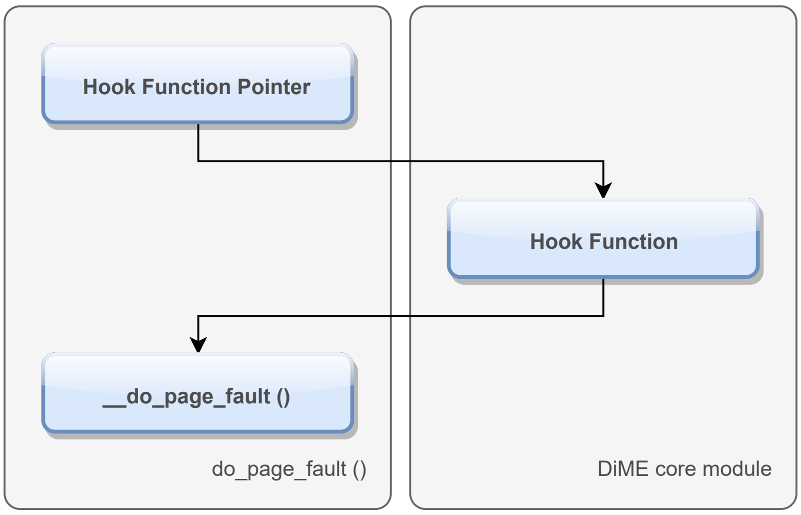


Figure 4.1: Hooking a page fault handler function.

4.5 Delay Injection

Now we have setup the page fault interception and induced the page fault for every memory access from the insertion of our module, which is our first step. The delay injection logic is implemented in the hook function defined in the module.

Linux kernel provides different ways to introduce a delay in an execution flow, viz. `delay` and `sleep` function families [10]. 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 coalesce wakeup with any other wakeup that may have happened for other reasons. At the worst case, 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 jiffies 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

Table 4.1: Delay function families

Function	**Idle for interval**	**Backed by**
ndelay	<~1000ns	busy-wait loop on jiffie
udelay	<~10μs	busy-wait loop on jiffie
mdelay	-	udelay
usleep_range	10μs - 20ms	hrtimers
msleep	10ms+	jiffies / legacy_timers
msleep_interruptible	10ms+	jiffies / legacy_timers

error for a broad range of delays. Comparison of delay accuracy is discussed in Section 5.1.3.

4.6 Page Replacement Policy

A local page is evicted from local memory to accommodate the remote page for which page fault is raised. A page replacement policy decides which local page to evict next. DiME provides an API to register/deregister a separate page replacement policy module, so that users can write their own local page list management mechanism and have control over which page to evict based on their own algorithm. This way user do not need to rely on default algorithm provided by operating system and can experiment with custom algorithms. We have implemented a simple FIFO policy for testing and illustrative purpose, which is used for all evaluations described in this document.

4.6.1 Policy structure hierarchy design

To implement custom policy module, user needs to implement functions described in `struct page_replacement_policy_struct` and pass this `struct` to core module while registering the policy. To avoid casting a void pointer with different policy structure types, user can use `container_of` macro to extract the pointer to parent policy `struct` from pointer to `struct page_replacement_policy_struct`. E.g., as shown in figure 4.2, pointer to object of type `struct prp_fifo_policy` can be derived from pointer to member object `prp` as follows:

```
container_of(prp, struct prp_fifo_struct, prp);
```

Since the policy functions have information about the type of policy structure, we neither need to maintain a void pointer to policy struct nor unsafe casting it to the actual type.

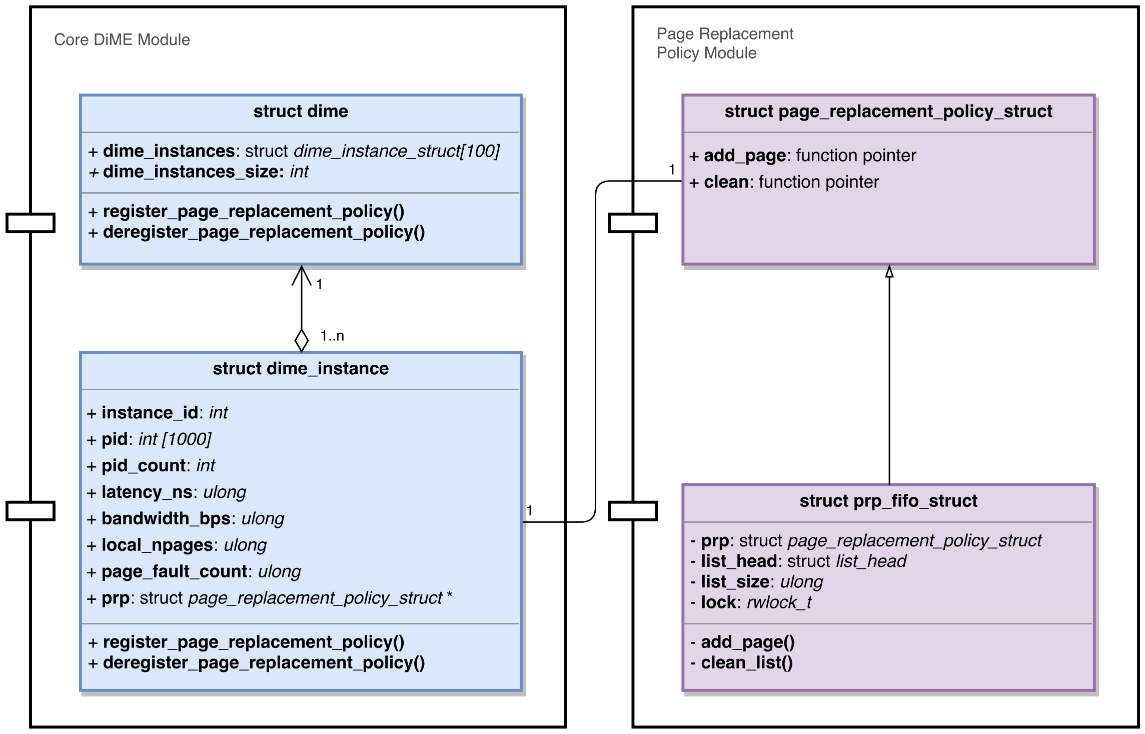

Core DiME Module
struct dime
+ dime_instances: struct dime_instance_struct[100]
+ dime_instances_size: int
+ register_page_replacement_policy()
+ deregister_page_replacement_policy()
1
1..n
struct dime_instance
+ instance_id: int
+ pid: int [1000]
+ pid_count: int
+ latency_ns: ulong
+ bandwidth_bps: ulong
+ local_npages: ulong
+ page_fault_count: ulong
+ prp: struct page_replacement_policy_struct *
+ register_page_replacement_policy()
+ deregister_page_replacement_policy()
1
1
Page Replacement Policy Module
struct page_replacement_policy_struct
+ add_page: function pointer
+ clean: function pointer
struct prp_fifo_struct
- prp: struct page_replacement_policy_struct
- list_head: struct list_head
- list_size: ulong
- lock: rwlock_t
- add_page()
- clean_list()

Figure 4.2: Page replacement policy class diagram

4.6.2 Virtual memory location

It is responsibility of policy to maintain a list of local pages, handle a remote page fetch request and signal whether to inject delay to this page fault based on location of the requested page. Local pages are virtually at the local memory for which page fault will not raise. Local page list helps to select a local page for eviction. To make a spare slot for remote page, a local page is protected and removed from the local page list, so that any future reference to that page would raise a page fault.

4.7 Second Hook

4.7.1 Accurate delay execution

The hook function can take variable amount of time to reach at the delay execution code based on the various tasks it needs to complete, e.g. local page lookup, update, etc. Also, `__do_page_fault` function can take variable time to return based on the type of page fault. Therefore, there will be variable amount of delay injection for each page fault.

To be able to inject the exact amount of delay based on the configuration parameters, the module must calculate the time spent on these extra tasks and inject delay worth of the difference of desired delay and time spent. To achieve the accurate delay execution, we need another hook function which will be called at the end of `do_page_fault` function. Now the first hook function will take the timestamp as a start of the handler and the second hook function will execute the difference worth of delay at the end.

The timestamp variable must be accessible from both hook functions, but it should not be taken as a global variable, since there might be multiple page faults being handled at the same time. So, we declare a timestamp variable in `do_page_fault` function, whose single instance will be common for both hook functions but different for each page fault.

4.7.2 Delay execution decision

The page replacement policy can take decision whether the faulted page is in local or remote memory based on the policy rules that are configured by the user, e.g. pin a portion of memory to local, mark data section of a process as always remote, etc. The delay injection depends upon this decision.

Now again since both hooks are different functions, we cannot have a global flag variable for the decision communication. So, same as timestamp, we declare a flag variable in `do_page_fault` and its pointer is sent to both hooks as an argument.

4.8 Multiple Emulator Instances Design

DiME allows users to run multiple emulator instances simultaneously. The instances has separate local memory and can be configured with different configuration parameter set. This feature is helpful to emulate different processes with different relative location of remote memory, e.g. rack scale or across data center.

4.9 Track Processes Under Emulator

4.9.1 TGID

DiME instance maintains a list of PIDs of processes which are running under that emulator instance. For each page fault, DiME checks the faulting PID in all running DiME instances to determine to which instance this PID belongs. In Linux kernel, the PIDs of threads that belong to same process are different, but `tgid` (thread group ID) is same as the parent thread. Therefore, we use `tgid` of `"current"` `task_struct` rather than `pid` to track the process.

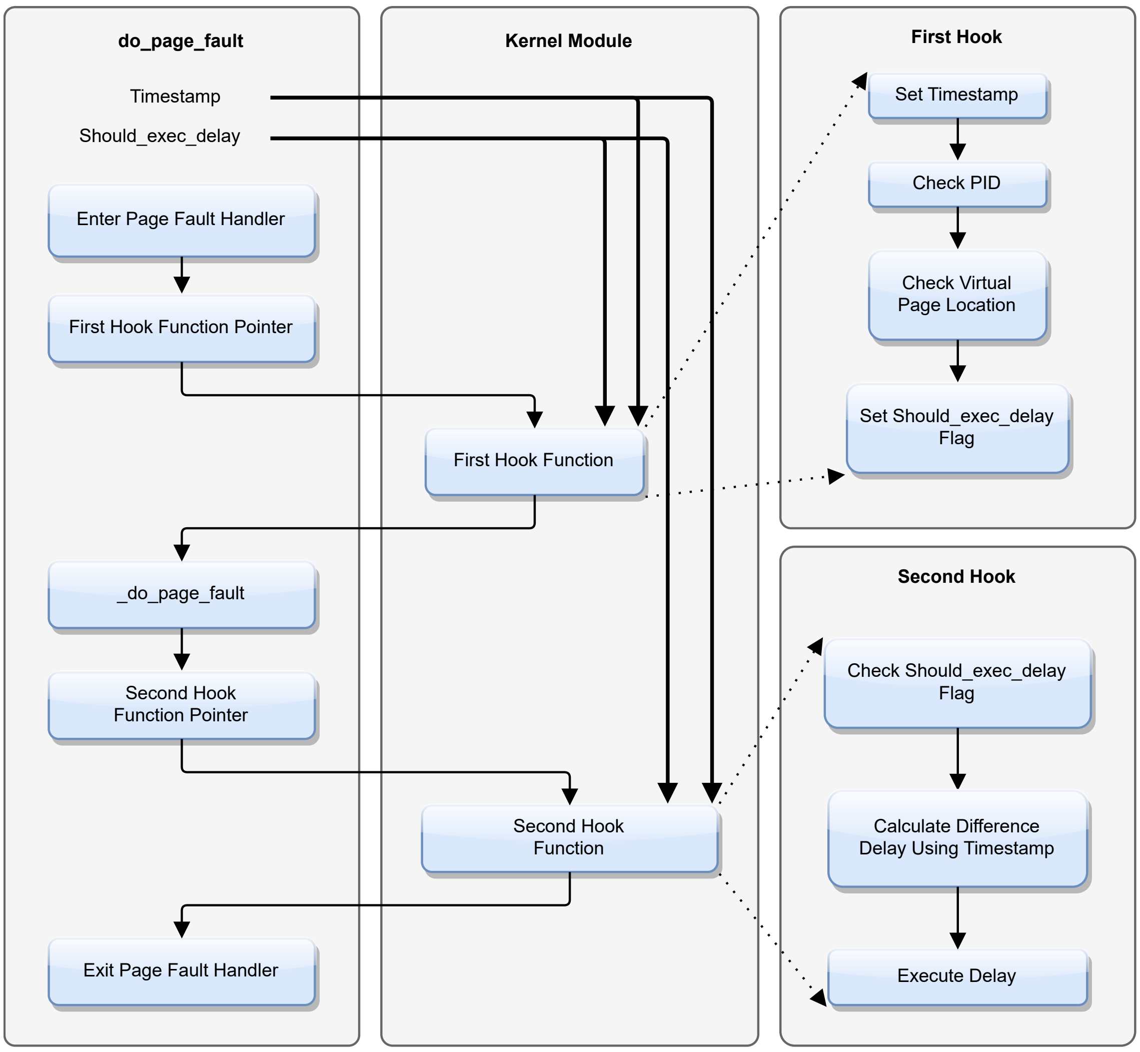


Figure 4.3: Emulator with second hook.

Table 4.2: DiME config parameters in /proc/dime_config

Parameter name	**Use**
`instance_id`	Instance ID of the target DiME instance
`latency_ns`	One-way propagation delay in nano-seconds
`bandwidth_bps`	Network bandwidth in bits per seconds
`local_npages`	Available local memory in number of pages
`page_fault_count`	Number of page faults raised till now
`pid`	A comma separated list of PIDs to add under emulator

4.9.2 Process forks

It is possible that a process running under emulator might fork a new thread or process. The expected behavior is that the new process should also run under the same DiME instance which contains the process which forked it. Since we are tracking based on thread group ID, forking a new thread satisfies the expectation. In case of a process, we need to add its PID to that DiME instance.

Linux kernel function `wake_up_new_task` wakes up a newly forked process after all the process initialization is done. To track process forks, DiME registers a KProbe [6] to `wake_up_new_task` and every time a new process is forked it check if the parent of new process belongs to any of the running DiME instances. If found, the new PID is appended to the PID list and all its pages are protected.

4.10 Dynamic Configuration

DiME supports dynamic modification of configuration parameters (Table 4.2) and add new emulator instances. A proc file, "`\proc\dime_config`", is implemented to handle the dynamic configuration change requests.

Listing 4.2: Procfs based DiME configuration example

```
$ echo "instance_id=1 latency_ns=500 bandwidth_bps=1000000000 local_npages=5000
    pid=8765,4321" > /proc/dime_config
$ cat /proc/dime_config
instance_id latency_ns bandwidth_bps local_npages page_fault_count pid
          0       1000    1000000000         2000                0 1234,5678,
          1        500    1000000000         5000                0 8765,4321,
```

Chapter 5

Evaluation

This section describes DiME accuracy and comparison with other emulator. 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 3 vCPUs and 6 GB RAM running Linux kernel 4.4.70 with huge page disabled.

5.1 Verification of Correctness

We evaluated DiME for correctness based on different scenarios and test cases.

5.1.1 Page faults

We have written a test program in C language, which takes number of pages as an argument and "malloc"ates the memory. Since we have disabled huge pages, default page size is 4kB. The program first accesses all the allocated pages in sequence and then access last 'x' pages. At the end of execution, test script extracts total number of page faults occurred from DiME procfs config file, `/proc/dime_config`.

We setup DiME with following configurations: `local_npages`=1000; `latency_ns`=10000; `bandwidth_bps`=10000000000000000 (infinite). We allocate 2000 pages in test program that runs under emulator. Table 5.1 shows test results where we vary number of last pages accessed ('x') and calculate the page faults. As expected, the number of page faults are constant till x=900, i.e. 2000 for first iteration of page access, and starts increasing after x=1100 by 100. It shows that 900 last pages were already in local memory and no page fault raised for those. Since FIFO page replacement policy is used, the page fault count jumps by 1000 at x=1100, because all local pages are evicted from list once again. The extra 100 page faults that we see in all test cases

Table 5.1: Last ’x’ pages access test results

Number of last pages accessed	**Total execution time (us)**	**Number of page faults**	**Average time taken per page fault (us)**
0	50,989.50	2,099.00	24.29
100	51,113.50	2,102.50	24.31
200	51,145.00	2,100.50	24.35
300	50,733.50	2,099.50	24.16
400	50,289.00	2,100.50	23.94
500	49,209.50	2,099.50	23.44
600	47,617.50	2,100.50	22.67
700	48,605.00	2,102.00	23.12
800	49,755.50	2,102.00	23.67
900	46,625.00	2,101.00	22.19
1000	53,386.00	2,339.50	22.82
1100	72,608.00	3,182.00	22.82
1200	73,316.50	3,269.00	22.43
1300	77,493.50	3,388.00	22.87
1400	79,642.50	3,491.50	22.81
1500	77,371.50	3,595.50	21.52
1600	82,422.00	3,695.50	22.30
1700	81,398.50	3,801.50	21.41
1800	86,171.50	3,910.00	22.04
1900	84,671.50	3,999.50	21.17
2000	90,360.50	4,107.00	22.00

are of test program text section which are always required in local memory for program execution.

5.1.2 Cache flush requirement

As explained in Section 4.4.1, it is mandatory to flush TLB after every page table update. To verify this, we built a C program which allocates 1000 pages, i.e. 1000x4KB memory, and accesses only two bytes from two different pages alternatively 1000 times per byte. We modified DiME to protect only the allocated 1000 pages, i.e. only data section of the process and not the text section. We set `local_npages=1` so that last page always get protected for each page fault. It was found that if we do not flush TLB

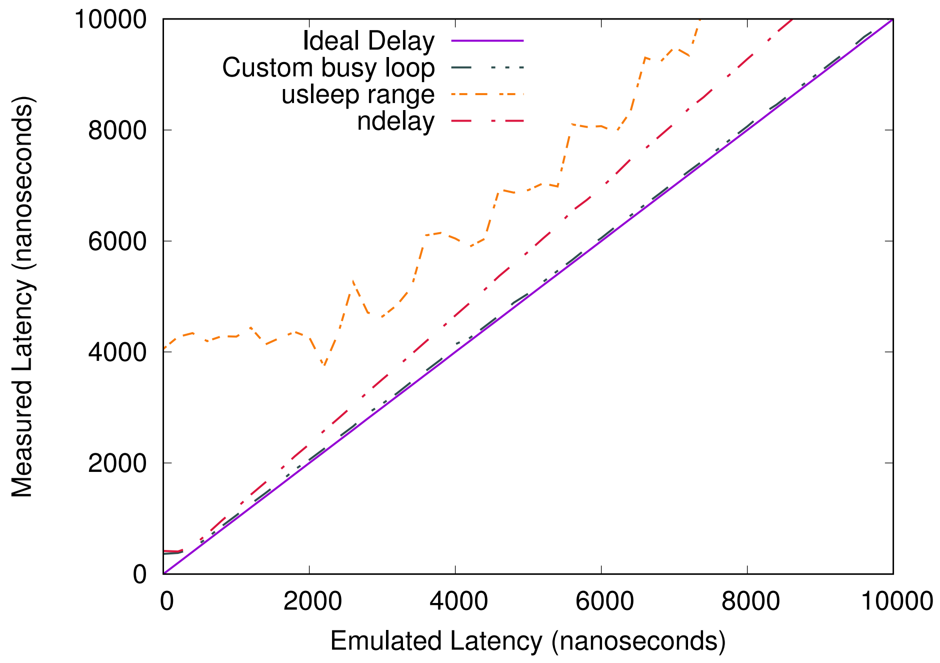


Figure 5.1: Accuracy of delay injection methods.

after protecting previously accessed page, the page fault count was around ~200, which is much less than expected count of 2000. After we patched the module with TLB flush, the page fault count was exactly 2000 as expected.

5.1.3 Delay injection accuracy

As described in section 4.5, there are multiple ways to inject a delay [10]. We evaluate each method by injecting delays from 0 to 10000 nanoseconds. In Linux kernel, different clock sources [2] can be used to measure time [7]. We use Time-stamp counter (TSC [11]) source available in x86 architecture (TSC library [12]) to measure actual injection time in each case. We sampled the measurement over 100 times and calculated the average for each method. Figure 5.1 show the plot between delay injected (x-axis) and actual measured delay (y-axis). The `usleep_range` function deviates most from idle delay injection since it involves interrupts and is nondeterministic delay between the given range. `usleep_range` is not useful for small amount of delays, since the error is higher for delays less than 2000ns. The `ndelay` function injects delay with much higher accuracy for lower values of delay, but the error increases linearly for higher values of delay. Our custom busy loop method of delay injection works most accurately with almost constant error of 70ns for any value of delay.

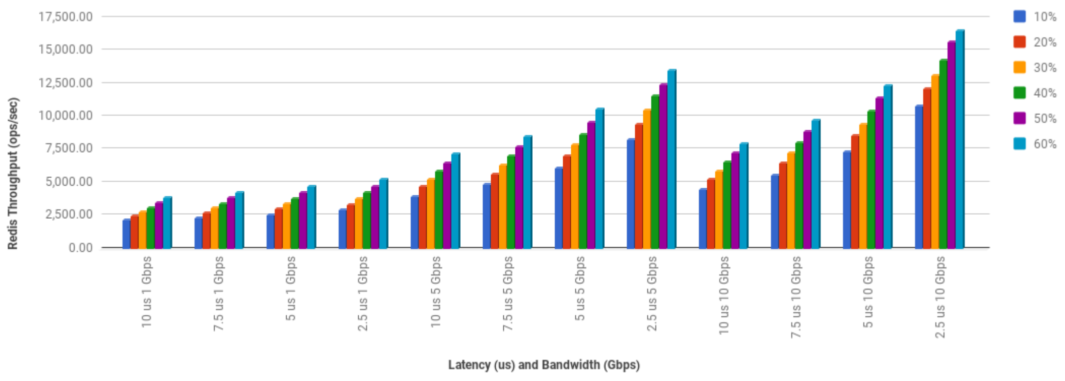


Figure 5.2: Redis throughput with constant latency and bandwidth, varying local memory

5.1.4 Average time per page fault

From Table 5.1, average time taken per page fault is 22.87μs. The expected time was 20μs (two times one-way latency, 2x1000 ns), the error is of 2.87μs. Since the time is measured from user space, the error contains context switching and interrupt handling time.

5.2 Application Throughput Test

The application performance degrades which are running under emulator. To verify if performance degradation is valid based on emulator configuration, we vary each parameter by keeping other constant. We ran Redis [9] server under emulator with 3 server threads. We used YCSB[3] benchmarking to generate the load. Redis client with 10 client threads, running on another VM, generates 1Gb of workload with equal amount of read and write operations. In Figure 5.2, we vary amount of local memory available for Redis server in percent of workload by keeping network bandwidth and latency constant. Results show that server throughput increases as available local memory increases.

Figure 5.3 shows the test results where bandwidth and local memory is constant while latency is changed. As we can see, the server throughput increases as latency decreases.

Similar results were observed when Memcached [8] was used in place of Redis.

5.3 Comparison with System-wide Emulator

The existing emulators (e.g. the one developed in [5]) emulates memory at system granularity. The entire system memory is partitioned into remote and local memory and user

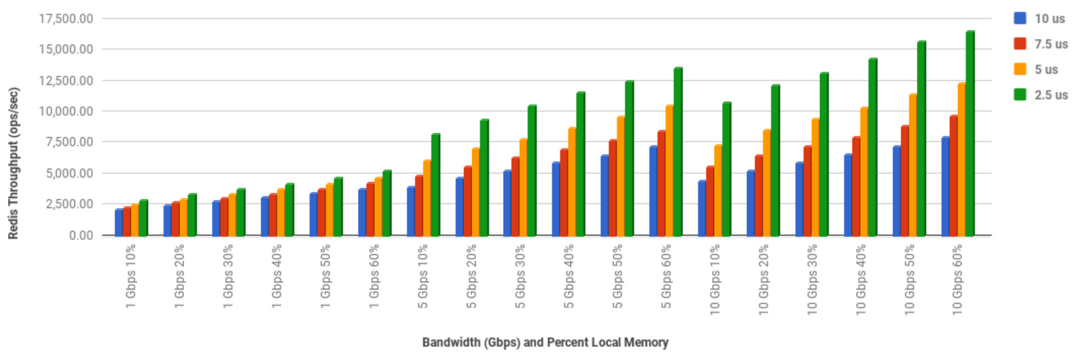


Figure 5.3: Redis throughput with constant bandwidth and percent local memory, varying latency

has no control over which portion of memory should reside on remote or local. In contrast, DiME emulates remote memory at the granularity of a process. DiME also has page level control over amount of local/remote memory and can specify custom page replacement policy. The allocated local memory is fixed and other applications running on the system outside the emulator has no effect on emulation. On the other hand, system-wide emulator cannot eliminate the effect of other applications running simultaneously outside the emulator. There are also other factors which affects the emulation accuracy in case of system-wide emulator, e.g. page cache usage, local memory consumed by other processes running in background, swap-out threshold, etc.

We compare the performance degradation of popular real world applications (Redis [9] and Memcached [8]) in case of DiME and system-wide emulator[5]. In evaluation setup, we vary local memory while keeping latency and bandwidth constant for both emulators. We run multiple tests based on number of instances of application and how they share the local memory described in Table 5.4. Note that we allocate local memory two times in case of shared memory compared to single instance case.

Figure 5.4a shows how number of page faults decreases as local memory increases while Figure 5.4b shows how Redis throughput increases as local memory increases. The number of page faults and application throughput is expected to be same in any test scenario described in Table 5.4 for a given emulator and a fixed set of configuration parameters. As we can see, in case of DiME, all four graphs almost overlap, while in case of system-wide emulator, there is very significant difference. Similar trends are shown by Memcached except the throughput in case of single instance of Memcached server running under DiME has more throughput compared to two instances of Memcached. The reason behind this behavior is that, Memcached is more CPU intensive compared to Redis and has more difference in throughput in both cases without any emulator (Table 5.2).

Table 5.2: % Degradation of throughput without emulator

	Redis	Memcached
Two instances of server running simultaneously	30,557.88	17,038.53
Single instance of server running	35,863.53	23,670.20
% Degradation	14.79%	28.02%

Table 5.3: Emulator overhead

	Number of page faults observed	**Execution time (us)**	**Time per page fault (us)**
System-wide emulator	2487746	19426305	7.808797602
DiME	2497209	12101210	4.845893956

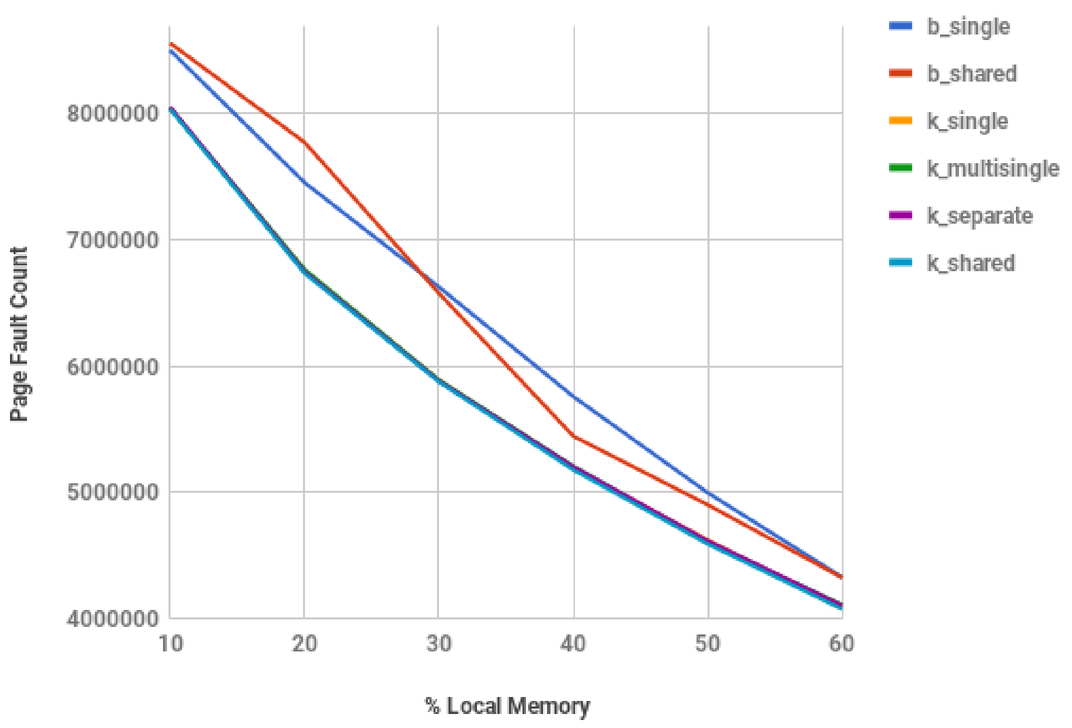
The results prove that DiME is capable of eliminating any background noise efficiently for more emulation accuracy compared to other emulators.

5.3.1 Emulator Overhead

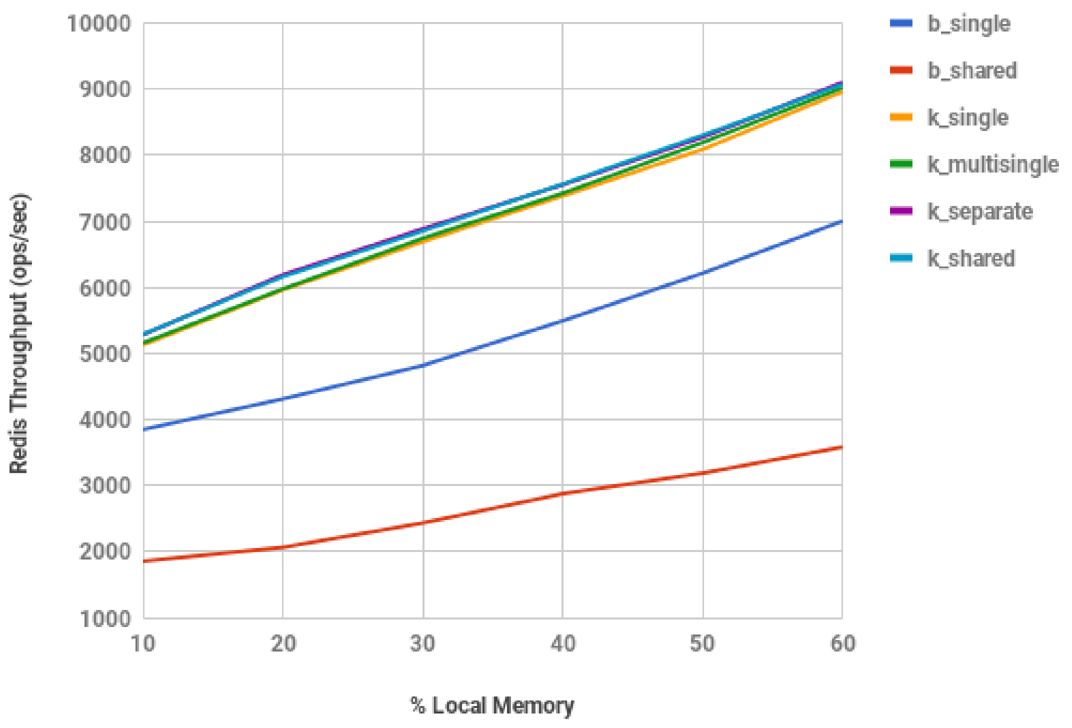
DiME has very low overhead of managing local pages since only operation it needs is to set PTE flags. This enables us to emulate low latencies. In a test setup, we set `latency_ns`=0 and infinite bandwidth for both DiME and system-wide emulator. We compared total time taken per page fault to execute a simple test program (described previously). Test result populated in Table 5.3 shows DiME has half overhead compared to system-wide emulator.

Table 5.4: Graph Legends for 5.4 and 5.5

b_single	System-wide emulator [5] with single instance of application running
b_shared	System-wide emulator with two instances of application running with shared local memory
k_single	DiME with only one instance of application running
k_multisingle	DiME with two instances of application running with one under emulator and other outside emulator
k_separate	DiME with two instances of application running in separate instances of DiME
k_shared	DiME with two instances of application running with shared local memory in one instance of DiME

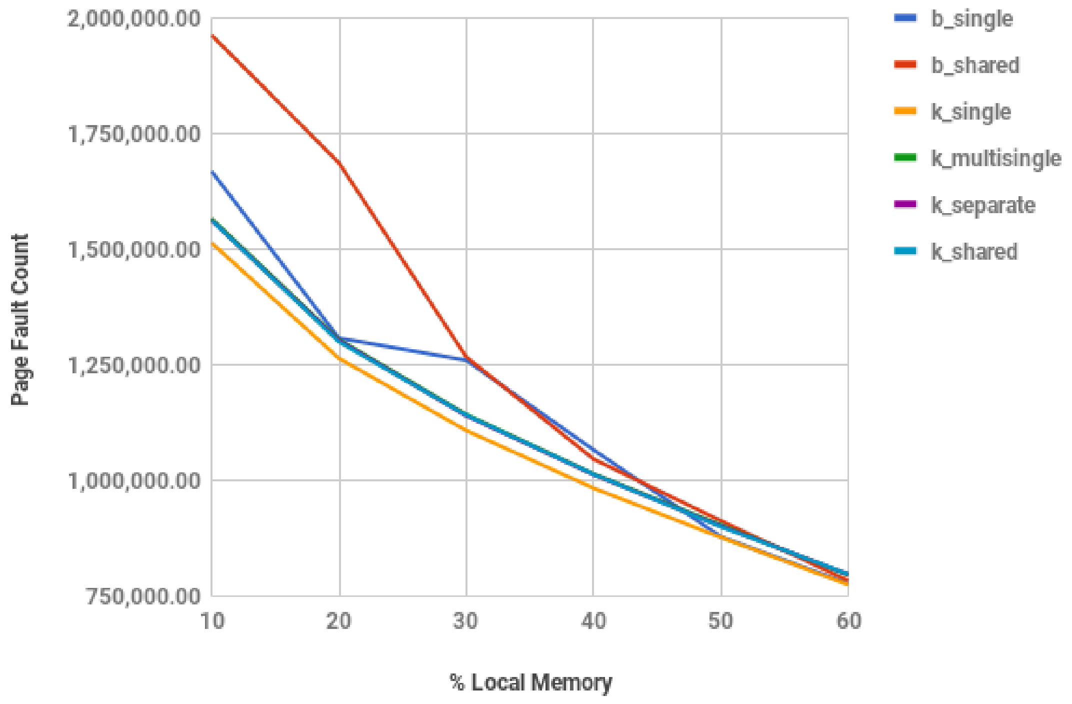


(a)

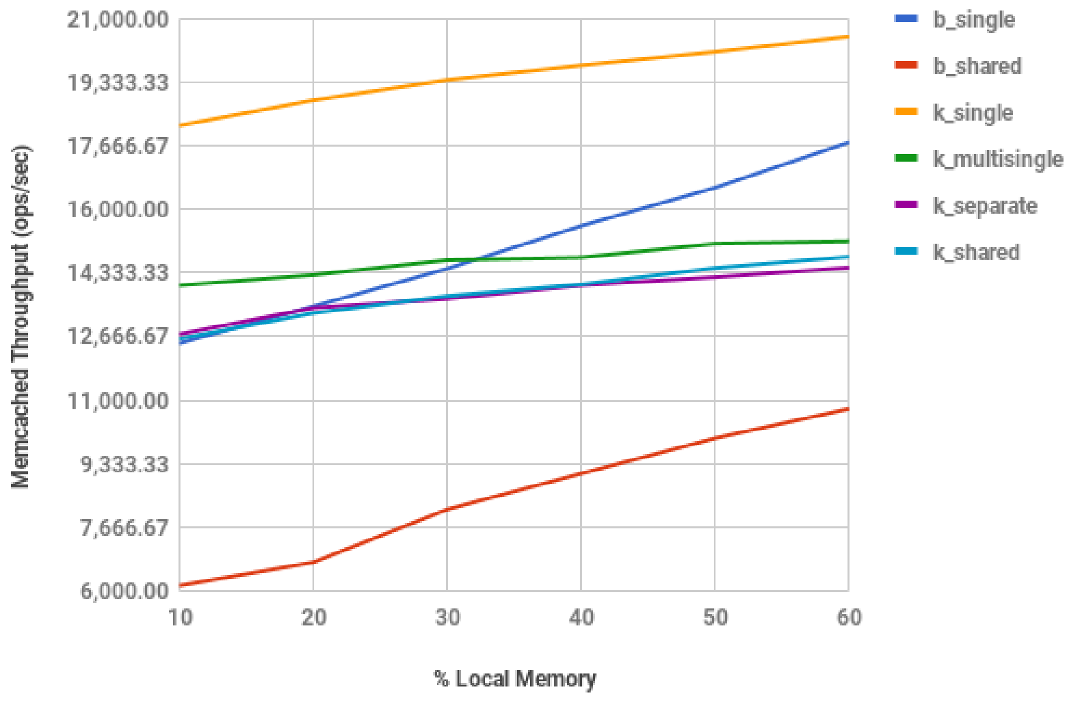


(b)

Figure 5.4: Redis performance comparison (Refer Table 5.4 for legends)



(a)



(b)

Figure 5.5: Memcached performance comparison (Refer Table 5.4 for legends)

Chapter 6

Conclusion & Future Work

DiME provides good accuracy in terms of delay injection, local memory management and emulator isolation compared to existing emulators. We also show that it is capable of emulating real world applications and can be used to evaluate application performance in disaggregated memory environment. DiME allows fine grained partitioning of memory and flexibility to manage local memory with custom algorithms.

Since the disaggregated memory architecture is still in exploration phase, we will continue to add more features as the architecture evolves.

We plane to build a realistic emulator by involving more physical machines. A single machine will act as a remote memory blade and other servers will be able to map its memory into their own address space over RDMA.

References

[1] Cache and tlb flushing under linux. `https://www.kernel.org/doc/Documentation/cachetlb.txt`, 2017.

[2] Clock sources, Clock events, sched_clock() and delay timers https://www.kernel.org/doc/Documentation/timers/timekeeping.txt, June 2017, 2017.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[4] DiME source code repository: https://github.com/networkedsystemsIITB/DiME, 2017.

[5] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, Berkeley, CA, USA, 2016. USENIX Association.

[6] Kprobes - an introduction to kprobes. `https://lwn.net/Articles/132196/`, 2017.

[7] Measuring time lapses http://www.makelinux.net/ldd3/chp-7-sect-1, june 2017, 2017.

[8] Memcached - a distributed memory object caching system. https://memcached.org/, 2017.

[9] Redis - an in-memory data structure store. https://redis.io/, june 2017, 2017.

[10] Timers in linux - information on the various kernel delay / sleep mechanism. https://www.kernel.org/doc/documentation/timers/timers-howto.txt, june 2017, 2017.

[11] Time stamp counter. https://goo.gl/v5gprq, june 2017, 2017.

[12] Tsc library. `https://github.com/dterei/tsc`, 2017.