Memory Disaggregation Emulator

A R&D Project Report
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)

2 May 2017

Acceptance Certificate

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

The R&D report entitled “Memory Disaggregation Emulator” submitted by Abhishek Ghogare (Roll No. 153059006) may be accepted for being evaluated.

Date: 2 May 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

Date: 2 May 2017 (Roll No. 153059006)

Abstract

Disaggregating various resources is the recent trend in datacenter architectures, where resources of same type are disaggregated in resource pools, e.g. compute resource, storage resource, network resource, memory resource. Disaggregation helps in efficient use of power and resources by sharing among disaggregated servers based on their usage patterns.

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.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional datacenters are built of collection of servers, and each server has dedicated resources, which are memory, cpu, non-volatile storage (SSD, HDD), network, etc. Per-server memory demands are increasing due to large-memory applications. Leading to more power to operate the memory. A new memory architecture is required to efficiently use the available memory. By disaggregating memory, multiple compute blades can connect to a single memory blade and dynamically share its capacity, resulting in decreased memory requirement and lower power consumption.

We have produced a linux kernel module which tracks the memory accesses by a process and emulates the delay caused by transferring remote memory data to local memory.

Chapter 2

Architecture Assumptions

We have built the module based on following disaggregation architecture. A server has a small amount of local memory. Rest of the memory is located on remote memory "blade". When CPU needs to access an address from an address space which is located on remote memory, it requests it though standard hardware protocol. 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 the local memory.

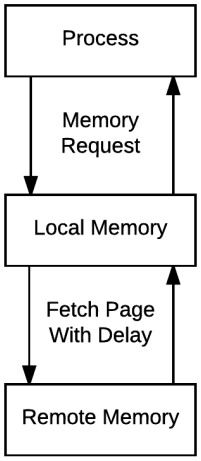

Figure 2.1: Disaggregated memory flow.

Chapter 3

Implementation

3.1 Overall Design

In our module, we can provide the PID of the process for which memory will be emulated as disaggregated memory. The module virtually divides the process memory into two parts: local memory and remote memory. The amount of local memory available in the emulator in terms of number of pages can be specified to the kernel module at the time of module insertion or dynamically using proc filesystem. The module emulates the delay whenever the remote memory is accessed by the process. This delay is the propagation and the 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 bandwidth of the network and the size of the page that is being fetched. Propagation delay and bandwidth can be provided at the time of module insertion or dynamically via configured proc files.

3.2 Memory Management in Linux

Since module has to inject the delay at a memory access, page fault handler is the place where we can put the delay logic. There are various reasons to trigger page fault in linux kernel. The MMU hardware (Figure3.1) is responsible for process memory address translation. MMU uses an internal page table to lookup page table entry (PTE) corresponding to a virtual address. This PTE is used to find the physical address for a given virtual address. PTE also stores various flags. Table 3.1 shows flags set in PTE.

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.

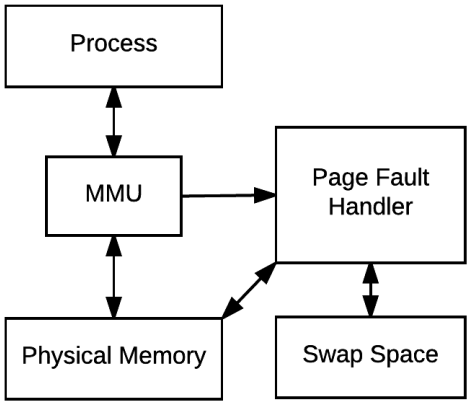

Figure 3.1: MMU and page fault handler interaction.

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 accessable
`_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

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

1. Page was never loaded into the memory, 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
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.

3.3 Emulator Workflow

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

1. Intercepts each page fault
2. Determine if the page fault is raised for a process for which the emulation is running
3. Determine if the faulted address is in the local memory or the remote memory
4. No delay injection if address belongs to local memory
5. If address is in remote memory, virtually allocate a local page
   (a) If local memory pool is free, virtually allocate the new local page to the faulted page
   (b) Else virtually evict a page from local memory pool and allocate the the new freed slot to the faulted page
6. Inject the delay for emulating the transmission and propagation delays

3.4 Induce Page Faults

A page fault can be induced 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. 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.

Kernel module sets this flags combination for all PTEs of the process with given PID when it gets inserted into the kernel. This way every memory access by the given process will trigger the page fault.

3.5 Intercept a Kernel Function

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

3.5.1 KProbe

KProbe is a debugging mechanism provided by the kernel to track the activities inside the kernel. It allows to set breakpoints in running kernel. A kprobe can be added to any function whose address is known at the time of kprobe insertion. Kernel recompilation is not needed in this method.

But `do_page_fault` function is tagged with "`NOKPROBE_SYMBOL(fname)`" macro, which disables kprobe for the given function. To avoid calling any kind of tracing machinery before the function has observed the CR2 value.

3.5.2 Kernel hooks

Adding a hook to the kernel function is another way of intercepting a kernel function, but with added overhead of recompilation of the kernel. 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.

Kernel module first initializes the required data structures then sets the hook function pointer and then sets the PTE flags combination as described above, so that it does not miss any page fault.

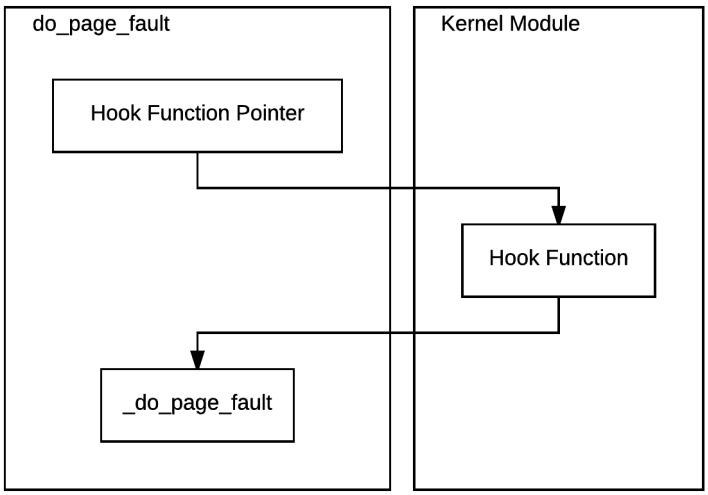

Figure 3.2: Hooking a page fault handler function.

3.6 Delay Execution

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.

3.6.1 PID of faulted process

Now to determine the process which has triggered the page fault, hook uses `current` macro to get `task_struct`. PID is extracted from `task_struct` which is compared with the PID provided at the time of module insertion. The hook function returns without doing anything if the PID does not match.

3.6.2 Virtual memory location

Next step is to determine virtual location of faulted page, i.e. if it is local memory or remote memory.

The module maintains a list of "local" pages, i.e. list of address of the pages which are virtually at the local memory for the process. Local page list helps to select a local page for eviction. When a remote page is brought to local memory, its address is added to the local page list. The page is released so that future address reference would not trigger the page fault. For evicting a local page, the page address is removed from the local page list and the page is protected, so that any future reference to that page would trigger the page fault and the hook function can inject the delay.

Now as the setup suggests, the page fault will trigger only for remote pages, we don't need to traverse the local page list to determine the virtual location of the page. So for every page fault for that process, hook will inject the delay.

This way emulator does not inject the delay for the pages which are in the list, but all other pages. Local page list also helps to select a local page for eviction.

Hook decides to remove a page from the local page list if the local memory is virtually full, i.e. the count of the pages in local page list has reached the max local memory specified during module insertion. After removing a local page from list, faulted page is inserted into the list, then hook executes the delay and returns.

We discuss a special scenario later in the document, where we must traverse the list to determine virtual page location. Currently the module does not check if the page is in the local page list. It requires a data structure which will help fast lookup and update. This feature is discussed in future work section.

3.7 Second Hook

3.7.1 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 calculation, 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.

3.7.2 Delay execution decision

Since there are reasons for which a page fault could occur other than our page protection, e.g. demand paging, swapped-out page. First hook function should decide whether

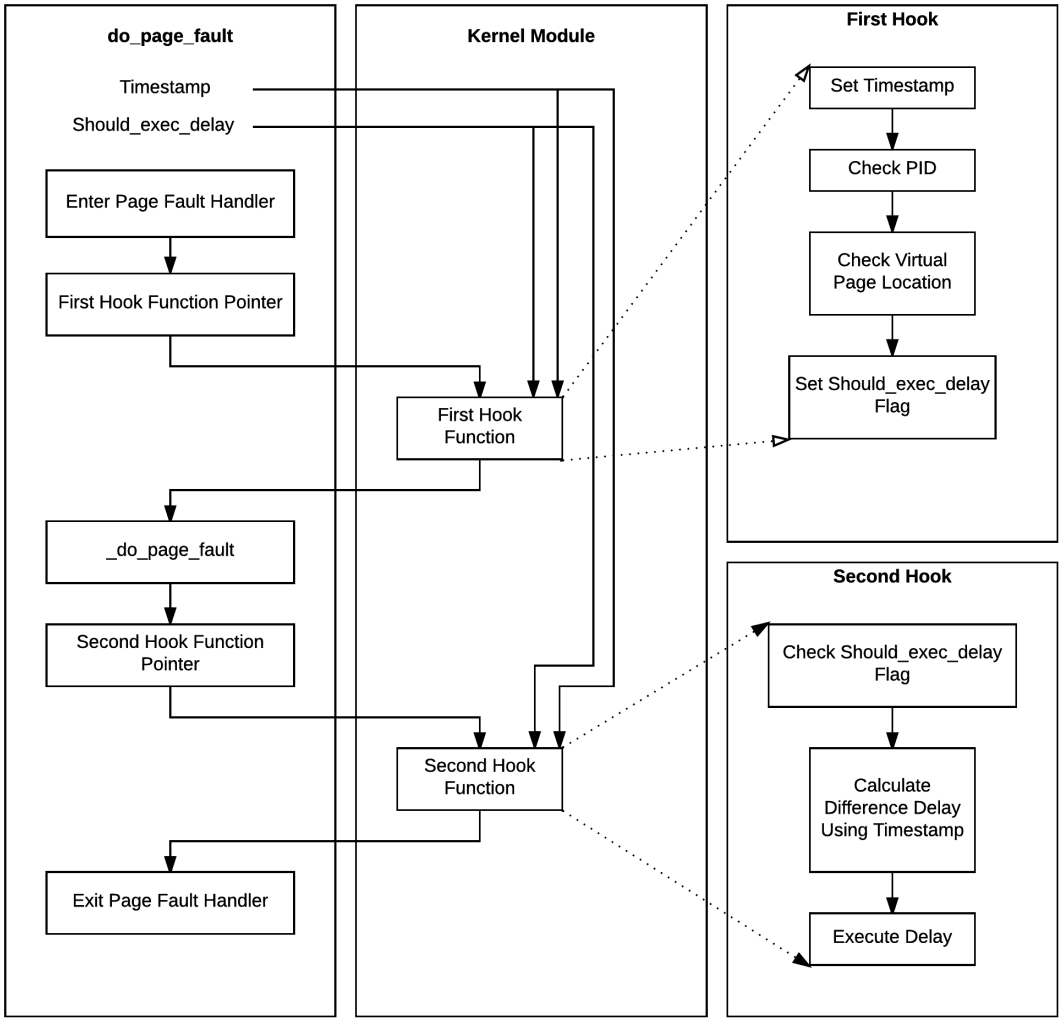

Figure 3.3: Emulator with second hook.

to execute the delay and second hook should execute the delay based on 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.

3.8 Tyeps of Page Faults

3.8.1 Minor Page Faults

A minor page fault occurs only when the page list 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. in case of same binary is executed by different users. 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.

3.8.2 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 initialises 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.

3.8.3 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 ignoring this scenario. Current module treats all pages as remote pages for which `_PAGE_PRESENT` is not set and executes delay. This leads to multiple entries in the local page list, which is not the serious problem but the emulation will not be accurate. Solution for this problem is discussed in future work section.

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

Chapter 4

Verification of Correctness

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 accesses all the allocated pages in sequence. At the end of execution, test script extracts total number of page faults occurred from module parameter `page_fault_count`.

Expected delay per page fault is calculated using module parameters provided (Shown in Table 4.1). Actual delay per page is computed by dividing total execution time with `page_fault_count`.

Table 4.2 shows the test results with different number of allocated pages in test program. Number of page faults are grater than total number of pages allocated even the test program accesses each page only once. This is because local pages are evicted as local memory gets full. Since we are using FIFO policy instead of LRU for eviction, pages containing execution code gets protected after every `local_npages` page faults. Because of this page fault count gets a boost.

Table 4.1: Emulator module parameters.

Parameter Name	Access	Use
`pid`	Read Only	PID of process to emulate, set at the time of module insertion
`latency_ns`	Read/Write	One way propogation delay in nano-sec
`bandwidth_bps`	Read/Write	Bandwidth of the network
`local_npages`	Read/Write	Local memory interms of number of pages
`page_fault_count`	Read Only	Total number of page faults

Table 4.2: Emulator module parameters.

# of allocated pages	Page fault count	Expected delay per page	Actual delay per page
500	564	40.00 ms	39.00ms
1000	1103	40.00 ms	43.00ms
10000	10156	40.00 ms	44.00ms
100000	100697	40.00 ms	44.00ms
1000000	1006127	40.00 ms	43.00ms

Following module parameters were set for testing:

`latency_ns`=20000 ns; `bandwidth_bps`=10000000000000000 bps; `local_npages`=1000.

Chapter 5

Future Work

5.1 Fast local page list lookup

Based on our tests, we found that when a page is swapped-in, all PTE flags are reset. We need to lookup the local page list if the page was in local memory space before it got swapped-out. Only way currently we see is to traverse the local page list. A fast mechanism to lookup if a given page is in local page list is required for every swapped-in page. A hash table can be used to maintain list of local pages.

More efficient method will be explored in parallel.

5.2 Local page replacement policy

A more efficient and accurate in terms of which emulates actual architecture algorithm for local page replacement policy is required. Currently FIFO policy is used.

5.3 Fine grained control

A fine grained control over location of memory range which can be provided dynamically using proc filesystem. A process can also use custom malloc function which uses extra flag argument which suggests location of the memory allocated. Dynamically flush a range of local memory. Dynamically set number of local pages.

5.4 Support for huge pages

Currently we are disabling huge pages in linux. We can enable huge pages and execute accurate transmission delay based on page size being transferred.

5.5 Process groups

To be able to add/remove multiple processes to the emulator dynamically. Automatically add child process to the emulator when forked.

Acknowledgements

This section is for the acknowledgments. Please keep this brief and resist the temptation of writing flowery prose! Do include all those who helped you, e.g. other faculty/staff you consulted, colleagues who assisted etc.

Abhishek Ghogare
IIT Bombay
2 May 2017