A

Dissertation

On

# "AN OPTIMIZATION OF POWER IN LARGE SCALE MEMORY SYSTEM"

In

## SOFTWARE TECHNOLOGY

By

**SHAILESH KUMAR PANDEY (ROLL NO. 2K12/SWT/11)**

Under the guidance of

**PROFESSOR R.K. YADAV**


**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**DELHI TECHNOLOGICAL UNIVERSITY**

**BAWANA ROAD, DELHI – 2015**

# DECLARATION

I hereby want to declare that the thesis entitled **"An Optimization of Power in Large Scale Memory System",** which being is submitted to the **Delhi Technological University**, in partial fulfillment of the requirements for the award of degree of **Master of Technology in Software Technology** is an authentic work carried out by me. The material contained in this thesis has not been submitted to any institution or university for the award of any degree.

**Shailesh Kumar Pandey**

**Department of Computer Science & Engineering**

**Delhi Technological University,**

**Delhi.**

# CERTIFICATE


DELHI TECHNOLOGICAL UNIVERSITY

BAWANA ROAD, DELHI-110042

_____ Date:

This is to certify that the thesis entitled "**An Optimization of Power in Large Scale Memory System"** submitted by **Shailesh Kumar Pandey (Roll Number: 2K12/SWT/11),** in partial fulfillment of the requirements for the award of degree of Master of Technology in Software Technology, is an authentic work carried out by him under my guidance. The content embodied in this thesis has not been submitted by him earlier to any institution or organization for any degree or diploma to the best of my knowledge and belief.

**Prof R.K. Yadav,**

Department of Software Engineering,

Delhi Technological University, Delhi-110042

# ACKNOWLEDGEMENT

I would like to take this opportunity to express my appreciation and gratitude to all those who have helped me directly or indirectly towards the successful completion of this work.

Firstly, I would like to express my sincere gratitude to my guide **Prof R.K. Yadav**, **Department of Computer Science and Engineering**, **Delhi Technological University, Delhi** whose excellent guidance, continuous support, encouragement and timely feedback were always there for me throughout the course of my work. Without his continuous support and interest, this thesis would not have been the same as presented here.

Also I would like to extend my thanks to the entire staff in the Department of Computer Science & Engineering, DTU for their help during my course of work.

**SHAILESH KUMAR PANDEY**

**2K 12/SWT/11**

# CONTENTS

# LIST OF FIGURES

# ABSTRACT

We are increasingly seeing computer systems day by day supporting larger size of RAM, in order to meet workload demands. However, power consumption of memory is significant, potentially up to more than a third of total system power on server systems. Hence with obvious reason, memory becomes the next major target for power optimization on embedded systems and smart phones, and all the way up to large server systems.

Most of the studies in the computer domain predict RAM usage in embedded systems and Personal computer to increase at very fast rate, RAM consumes ~30% of power usage of a system.

Modern memory hardware supports a number of power saving measures - for instance, the memory controller can automatically put memory DIMMs/banks into content-preserving low-power states, if it detects that entire memory DIMM/bank has not been referenced for a threshold amount of time, thus reducing the energy consumption of the memory hardware. We term these power-manageable chunks of memory as "Memory Regions".

The OS needs to know about the granularity at which the hardware can perform automatic power-management of the memory banks (i.e., the address boundaries of the memory regions). On ARM platforms, the boot loader can be modified to pass on this info to the kernel via the device-tree. On x86 platforms, the new ACPI 5.0 spec has added support for exporting the power optimization capabilities of the memory hardware to the OS in a standard way.

# Chapter 1

# INTRODUCTION

## 1.1 Introduction

Most of the studies in computer domain predict RAM usage in embedded systems and personal computer to increase at very fast rate, memory consumes potentially more than one third of total power, this is the major target for power optimization now.

Modern hardware have the capability to put part of memory not used in low power state, like in DDR3 memory if some region from memory is not referenced for threshold amount time, it will be moved this region to low power state. We use this capability for the power optimization.

When Memory used in the system is of large size, it will be split into multiple blocks or chips. This design will be very helpful for power optimization as the block or chip not in use will be put into low power mode. This will have direct impact on power consumption from the device.

Reducing the power usage from memory not referenced is done automatically by hardware, this paper goal is to minimize the memory references to the regions in minimum area by doing the memory allocation and de-allocation such that it is not distributed across the entire memory space, and in addition there is light weight compaction/reclaim support to reclaim very partially memory filled regions. It is an applied research work.

## 1.2 Motivation

It had been observed significant variations in inter-die and intra–die processes in past years. Most of the research in power domain has predicted that this will further go up by more than 50% in coming years. These variation are due to ambient conditions, wear out from devices or the manufacturing process itself. In spite of having so many variations on hardware, software always assumes homogeneous structure for its stack implementation.

One method of guard banding systems with such hardware variability is to binning semiconductor device to lowest possible operating frequencies so as to reduce the impact of variation in inter-die but this will compromise on performance of the device and optimal power usage.

Also we have now come up to multi-core technology which has further complicated this issue. In order to minimize the effect of these variations we need to exploit the inherent variations in devices so as to improve system performance.

One of major component which suffers from power variations is main Memory systems.

On chip memory consumes nearly around one third of system power consumption.

Above mentioned problems of intra-die and inter-die variations in main memory can be resolved using caching mechanism, DRAM and OS level power management. However such changes can be classified as Hardware change as it requires changing existing memory configurations.

In this work, we present memory manager which is optimized for power and it is a system-level variability-aware solution that adapts to the power variability inherent in DRAM memory modules. With this approach we use power of the variability in DDR3 onward memory. Our approach could be generalized to work at finer granularities of memory, if variability data and hardware support are available.

## 1.3 Related Work

Power optimization is one of the biggest research areas since long, with advent of embedded systems like mobile phones power optimization has become more important now days. Battery usage of embedded system is directly dependent upon power consumption from the device. Similarly many server computers keeping running for long hours, if they power usage from them is optimized this can save lot of power. As much research was done to focus on how much power consumption is from different component in any embedded or server/personal computer system, many of them point to main memory as the most power consuming component. It takes up to one third of power consumption from the whole system. Following are the recent research papers which have proposed the power optimization techniques for memory.

Min Kyu Jeong proposes Balancing DRAM Locality and Parallelism in shared memory CMP systems IEEE 2012**,** to partition the internal memory banks between cores to isolate their access streams and eliminate locality interference. It was implemented by extending the physical frame allocation algorithm of the OS such that physical frames mapped to the same DRAM bank can be exclusively allocated to a single thread. Modern memory systems rely on spatial locality to provide high bandwidth while minimizing memory device power and cost. The trend of increasing the number of cores that share memory, however, decreases apparent spatial locality because access streams from independent threads are interleaved.

Mishra, V.K proposes Performance enhancement of NUMA multiprocessor systems with on-demand memory migration 2013, most efforts are made to research in the area of memory management to reduce power consumption from memory to most optimized minimum. In the paper [2] presents memory migration on-demand policy to enable automatic dynamic migration of pages with low cost when they are actually accessed by a task. Major focus is on the quality of the scheduling has a strong impact on the overall application performance because of process and data affinities, This issue is now becoming critical due to the variable memory access latencies in NUMA architecture, because in NUMA architecture local data access being significantly faster than remote access.

Gangyong Jia Proposes, Memory Affinity: Balancing Performance, Power, Thermal and Fairness for Multi-core Systems, IEEE 2013, this paper firstly proposes memory affinity which retains the active and low power memory ranks as long as possible to avoid frequently switching between active and low power status. Main memory is expected to grow significantly in both speed and capacity for it is a major shared resource among cores in a multi-core system, which will lead to increasing power consumption. It is critical to address the power issue without seriously decreasing performance in the memory subsystem. Also present a memory affinity aware scheduling (MAS) to balance performance, power, thermal and fairness for multi-core systems.

Zhang Haiyang introduces Red-Black Tree Used for Arranging Virtual Memory Area of Linux IEEE 2010, Red-Black Tree's definition, merit and the content of the VAM in Linux Then discusses how to accomplish the Red-Black tree in Linux Kernel, and Red-Black tree used for arranging virtual memory area of Linux.

In all the recent research done and specially the research papers in memory optimization, we can observe there is great focus on optimizing power for multi core architecture when its access memory. Either this research has focus on using NUMA Architecture or it is focused on access of memory from multi-core architecture. Most of the systems in embedded or server domain do not use NUMA as this architecture is very complex and still lot of work needs to be done on it. Also memory ranking and optimization around it is tends to be moving in hardware domain. In comparison, we have found UMA is most common architecture used in embedded systems as well as servers. Also with larger memory systems in place, there is significance requirement to optimize power from memory itself.

All the recent research in memory power optimization have focus on Non Unified Memory Access Architecture or Hardware based optimizations, none of them has focused to find out power optimized memory solution for Unified Memory Access based architecture with large size memory, which has become the major focus of this theses.

## 1.4 Problem Statement

Our Focus is on the optimization of power consumption from memory as this is most power consuming area for embedded and computer systems. We can target to work on the management of memory to reduce the power consumption by switching of the memory components not in use to the low power state. In order to do so we need to make sure to reduced the used area of memory to the lowest.

Memory management is designed to efficiently provide the required memory to applications from the total memory available and same should be return back to the available memory when not in use. We can additionally make our algorithms to focus on making the memory allocation and de-allocation considering the area from which memory is allocated and de-allocated.

Problem statement for our projects is to develop power manger at system level so as to reduce the allocation of memory to the smallest number of regions available for an application in order to reduce the power consumption.

## 1.5 Scope Of The Work

Memory management is most important part of any operating system. It is designed to keep performance of the system most optimized. The main consumer of power in any embedded or server system is main memory, we have focused to analyze the pattern of memory usage and impact of pattern of usage on main memory power consumption. In this thesis, we purposed a new design for the memory manager which has focus on keeping the power optimized.

This thesis has three goals to achieve power optimization for memory manager:

1. **To design a new architecture for memory manager with target of power optimization.**
2. **To design algorithm which divide main memory of system into regions which are power switchable and then allocated memory to process are kept to minimum number of regions.**
3. **To establish results of the purposed algorithm to demonstrate its impact on power consumption of the system.**

a. In our optimization proposal, we divide the memory zones in regions, which are based on block structure of the memory. Further along with memory regions buddy system is also divided in regions. It will enable the allocation of memory to least number of regions.

b. To achieve our goal, we have modified the buddy system for memory allocation. In Linux memory manager, there is one buddy system per zone but in our proposal we will manage buddy system per region. During process request for memory allocation to allocatore, same will be passed to region buddy allocator, which will pass control to first available buddy region in memory manager. This will allocate the maximum available in that buddy region, after that it passes control to the next buddy region of

the zone where memory is free. In this way it will allocate memory from one region before going to next region.

c. To analyze the results on power consumption from the proposed algorithm change, we have tested our algorithm on multiple machines; First system is Intel core 2 Duo CPU with 2 GByte of RAM and 128 GB HDD. Second system is Intel core I5 with 6 GByte of RAM and 512 GByte HDD, In third system Intel Core I5 with 4 GByte of RAM and 512 GByte of HDD. This will further help in making the results generalized as we tested them across multiple systems with different configuration.

# Chapter 2

# BACKGROUND STUDY

## 2.1 Power Measurement And Optimization

Power optimization and management in general is an ongoing and active area of computer science. Irrespective of the size or function of any computing system, all consumes power, and the consumption of power always carries an economic cost. The economic cost manifest differently for different systems, and on the surface, it would seem the cost would bear more weight on some computing devices such as Smartphone while other computing systems are seemingly immune (such as data center).

For example, it is easier to understand that in a battery-operated, cordless device, such as Smartphone, higher power consumption means either the battery needs higher capacity means increase in direct cost of manufacturing and sale price or have to suffer shorter usage time or potential lost business to competing vendors on the market -- opportunity cost loss. On the other end of the physical size spectrum, a data center that is attached to the AC power grid seems to have unlimited supply of power and doesn't suffer the same power constraint as a Smartphone. However, as an operator of a data center with a fixed budget, every dollar that is spent on paying for the power (both for running the computing systems and cooling them) means one less dollar to spend on increasing computing capacity of the center, and it is the computing capacity that generates revenue for the owner.

Power optimization is very important for embedded or hand-help devices as they require battery to be used, If power consumption is optimized for a device, it can be used for longer duration of time. An optimized mobile system with same amount of battery can be used for double the time if we are able to optimize its current to one half. Similar is the case for large server systems, they are expected to be running for long hours and power optimization is very important as amount of power optimized is directly to cost saved for their usage.

Even though power consumption optimization may look as prime responsibility for the hardware designer but software design has also major role in its optimization.

**Basics of power consumption**

There are many factors which impact on the power consumption of a device; four major factors are application, Voltage, frequency and processor technology. In below details we will find out why these factors are so important.

Application of device is very important to understand how power profile of the device will differ. Let's take the example of hand help media player, Media player will require to be used for long duration of time. We need to focus of data flow optimization and algorithm rather than idle power mode.

Similar is the case of a mobile device but the focus of power optimization is different. It needs to be use during a call and while all other time it remain in idle mode. If we are able to reduce the power usage of idle mode in a mobile device than it will have major impact on its power usage.

**Types of power consumption**

In any device there can be two type of power consumption, Static and Dynamic.

Total power of a device is calculated as:

Total power = Dynamic power + Static power

Or

Total power = idle power/base power + variable running power due to processing/ongoing activity.

From software, we can control over the large part of dynamic consumption but static consumption there is not much control. From static power consumption we mean the power consumption from a device without consideration of the work ongoing.

Other factors which can impact on static power consumption are temperature, process and supply voltage. Temperature increase will increase electron mobility which will increase the flow of electrons and results in more static power consumption.

**Dynamic power consumption**

Dynamic power consumption from embedded devices includes many factors like core subsystems, peripherals, Main memory, Clocks etc. At lower level we can say this as power used when switching transistors. Dynamic power consumption will be dependent on the voltage supply level but it will not dependent on temperature.

There are four type of power consumption which needs to be considered for a device.

Typical power consumption, Average power consumption, Maximum power consumption and worst case power consumption.

Maximum power consumption as the name suggests is the highest power reading measured from the device at any particular instance. This indicate to us the maximum power required by us to operate the device.

Average power indicates the power usage from the device divide by time. This helps us to understand the battery or the supply required to operate device for a particular amount of time.

Worst case power indicates the average of full power usage from a device over a period of time. This means that we will operate a system with all its parts operating to maximum capacity for some period of time and than average out its power usage.

Typical Power consumption is also very important. In normal usage, a device rarely operates to the worst case power, For example we may not be always using all core of a processor when using our personal computer. Typical power usage refers to power consumption from the device from general power use cases.

**Power Optimization** There are several methods to optimize static and dynamic power consumption. We can optimize the system for power by selecting right components in system and designing dynamic software architecture for them.

DDR means dual data rate, DDR SDRAM take advantage of both edges of DDR clock to write and read data. This means we are effectively doubling the effective data rate

There are various features in Main memory and these features affect the power usage from it, EDC (Error detection), ECC (Error correction codes), Type of Bursting, data refresh rate which are programmable, Programmable memory configurations.

Various methods to optimize power from a main memory are optimizing power by data flow, timing, Interleaving, DDR configurations, burst access and software data organization.

## 2.2 Memory System Architecture

There are some important terms related to memory and memory management, we need to understand them in order to understand our proposed architecture. We worked in this thesis with DDR3 DRAM memory. This memory has the capability to put regions of memory which are not referenced by CPU into low power mode.

A typical DRAM based memory has memory controller connected to multiple memory channels. Further these channels will have multiple DIMM in them which are further divided into ranks on opposite sides of the module. Memory controller can access these ranks independently. There are further banks of memory which are composed of columns and rows. We can access any memory location with unique combination of rows, columns, bank and rank.

Our approach is the optimized the power consumption by targeting the least selectable memory part which can be put into low power mode and make sure there is no data present in this area if possible with working on pattern of data.

**Architecture of memory management in Linux**

Memory management system for Linux is simple but yet scalable to handle various type of processor architecture from x86 to ARM or power PC etc. There are two parts of memory management, architecture dependent code and non-architecture dependent code.

Most of the code is kept in Non-architecture dependent part so that our memory manager can work on new architecture with major modification. Architecture dependent part of the code is for the parts which have major dependencies on hardware or need to be optimized for any architecture.

There are two major type of memory architecture, NUMA Architecture and UMA Architecture.

NUMA means non-unified memory access. This is applied to the system which has memory where access to one area of memory takes different time than accessing other part of memory. This means access to memory is non-unified. Typical example for this is a multiple processor system which has main memory connected to each processor but also accessible from any processor in system. In this case processor will have faster access to memory connected directly to it rather than memory connected to other processor.

UMA is unified memory access. In these systems all the memory access to any location in a system takes same time independent of the location being accessed. Typical example of UMA systems is personal computer with single main memory. Any location access in this main memory from processor takes same amount of time.

UMA Architecture

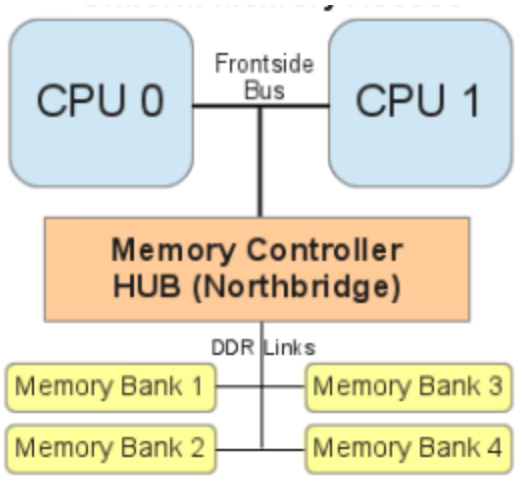



*Figure 2.1 UMA Architecture*

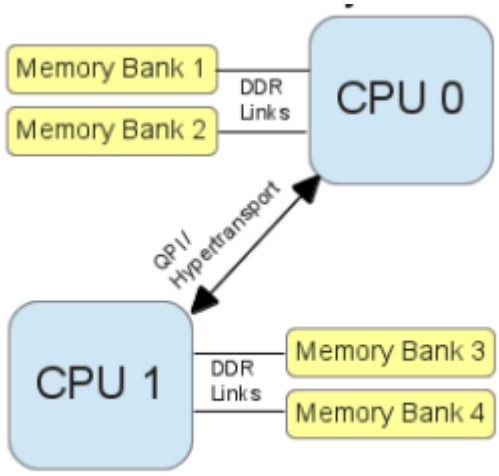



*Figure 2.2 NUMA Architecture*

We can further divide typical memory architecture in Nodes and Zones.

**Node**: A memory which is directly accessible from CPU can be termed as Node. E.g. Multiple main memory modules connected to a single processor, in this case all these modules together will be called as Node.

**Zone:** A node (group of different memory regions) further can be divided into different memory regions may be termed as Zone, which will be dependent on 32 bit or 64 bit Architecture.

**Major Zones in any system are as below:**

DMA: This is lowest 16 MByte of area in the main memory. This area is separated from other memory area as there was some old generation hardware which can only do DMA using this area and are not able to access area beyond it.

DMA32: This is also DMA area but for 64bit machines. With latest technology now there are devices which can do DMA up to 4GByte of memory. This region has limit up to 4GByte of memory.

Normal Zone: Normal zone changes between 32 bit architecture or 64 bit architecture. On a 64 bit machine normal zone is all area beyond 4 GByte of memory while on a 32 bit Architecture this is between 16 MByte to 896 MByte.

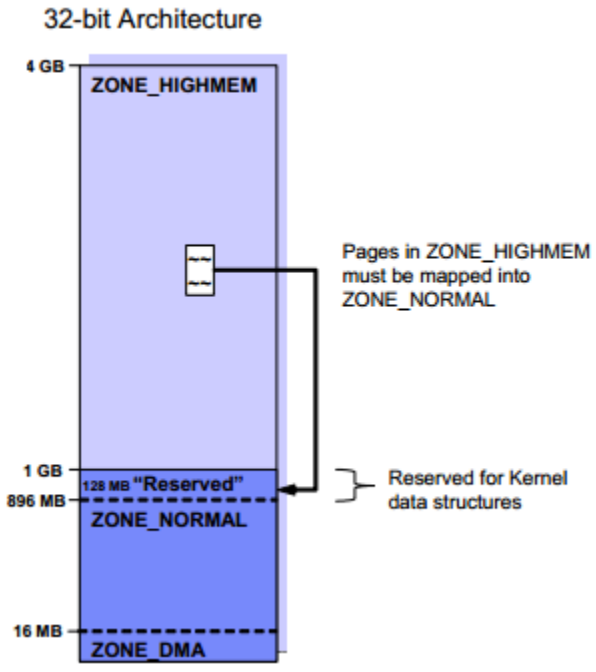



*Figure 2.3 32 Bit Memory Zone*

On a 64 bit system, we will see most of the memory mapped to ZONE DMA32 only if its total value is system is around 4 GByte while in case of 32 bit system most part of it will be in Zone NORMAL or HighMem.

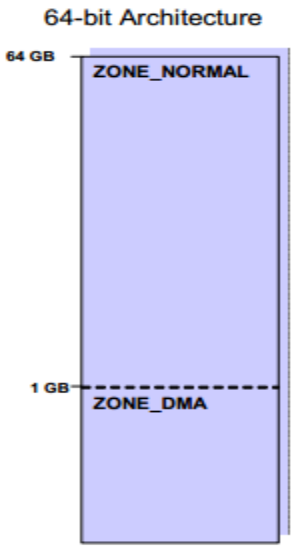



*Figure 2.4 64 Bit Memory Zone*

High Mem: This region is valid only for 32 Machines. This will be to access area of memory above 896 MByte of memory in 32 bit machines.

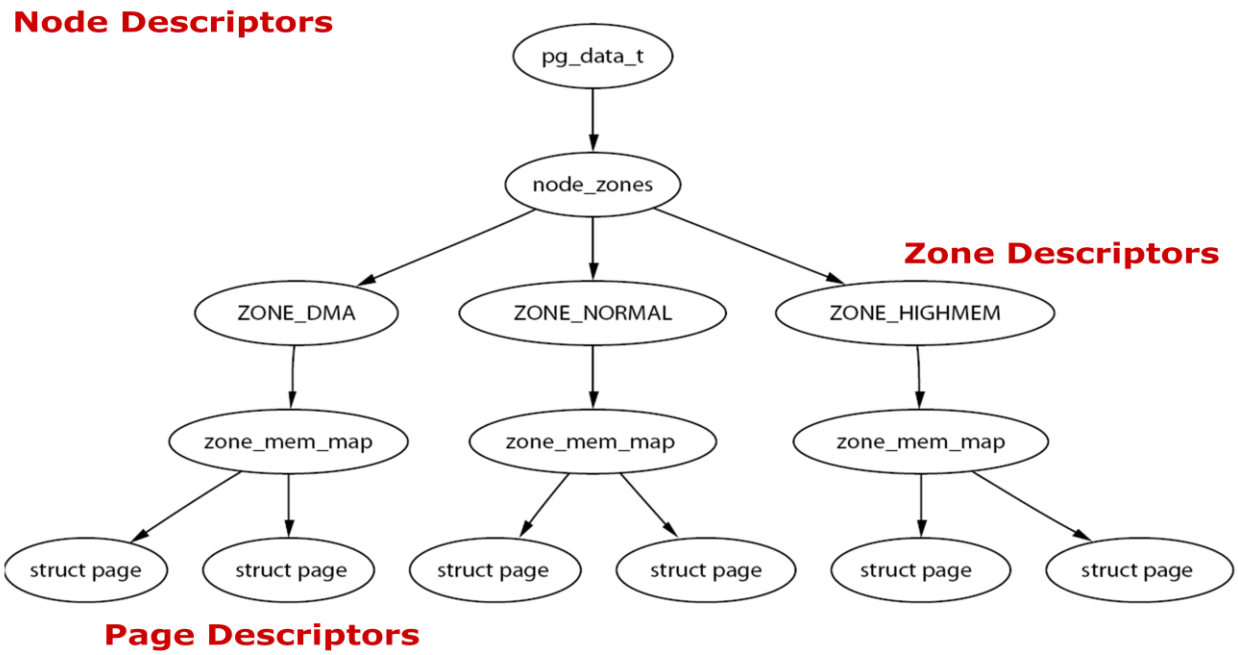



*Figure 2.5 Zoning in the Traditional Linux x86-64 Kernel*

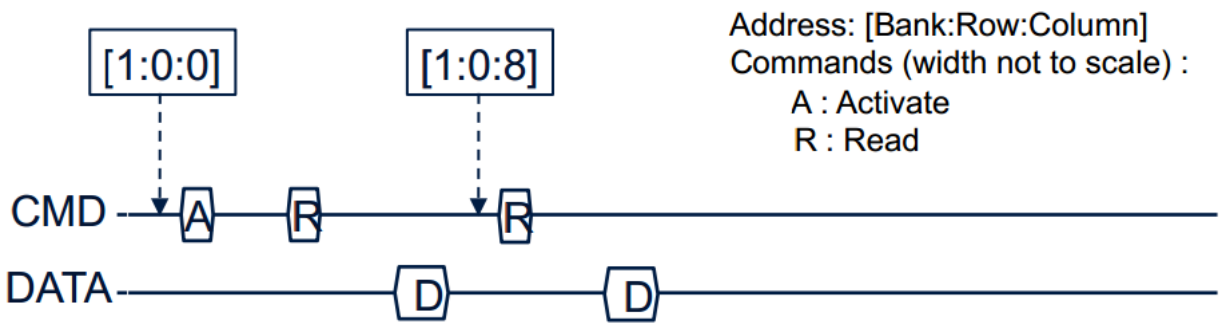
## **2.3** Balancing DRAM Locality and Parallelism in shared Memory CMP Systems:

Balancing DRAM Locality and Parallelism in shared Memory CMP systems Min Kyu Jeong, Doe Hyun Yoon^, Dam Sunwoo*, Michael Sullivan, Ikhwan Lee, and Mattan Erez.

- Spatial locality is lost when independent access streams from many cores are interleaved
- To preserve the locality, we propose to isolate streams to exclusive set of DRAM banks
- Partitioning banks reduces bank-level parallelism available to each thread
- To compensate for lost BLP, we increase effective bank count with sub-ranking
- Our combined approach simultaneously improves performance and efficiency, while maintaining fairness

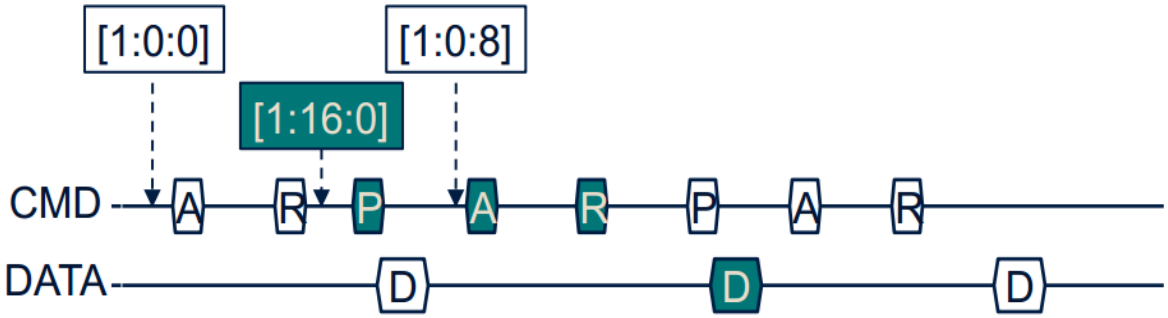
**Spatial Locality in DRAM**

- Many applications exhibit spatial locality
- Modern memory systems are designed to exploit spatial locality to deliver performance cost effectively (e.g. Row-Buffer Hits)



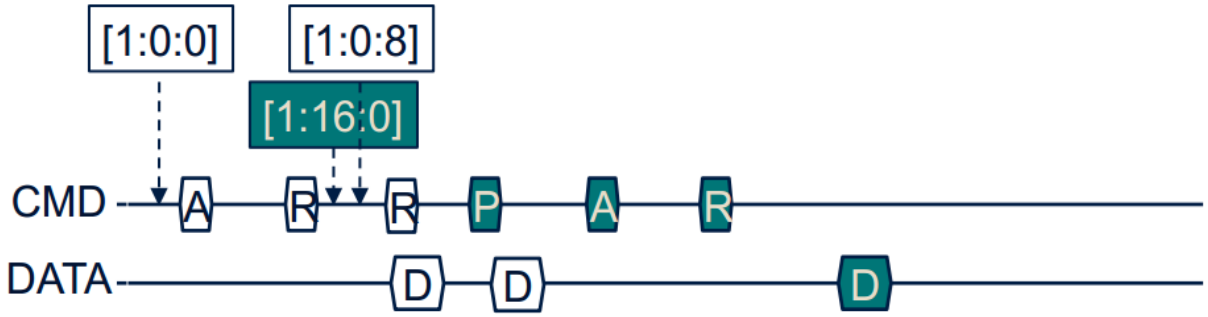
**Loss of Opportunity**

- However, in chip-multiprocessor systems, spatial locality is lost as independent access streams from multiple cores are interleaved
- Result: Lower performance and energy efficiency



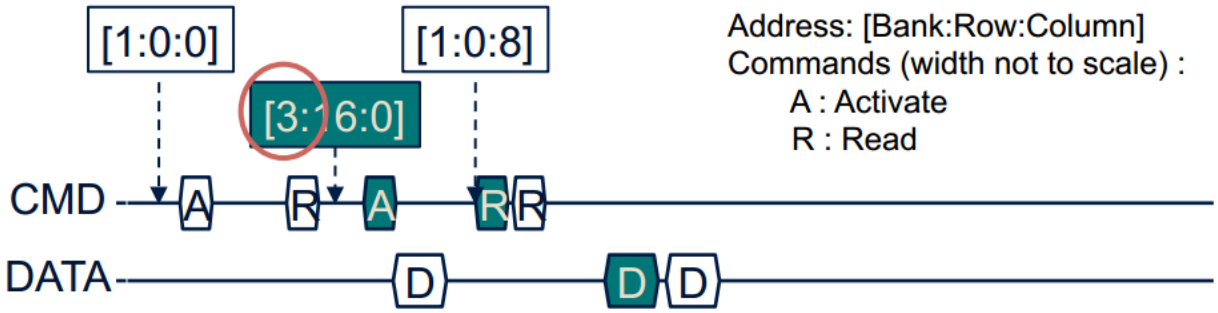

**Prior Work**

- Out-of-order scheduling
  - Reduces the number of back-and-forth row swapping
  - Arrival interval should be short enough
  - Limited by the scheduling queue size
  - Delaying certain streams hurts performance and fairness
- MP fairness-aware scheduling
  - Maximizing bandwidth != system performance
  - Optimize for system fairness and performance
- All still pay the cost of bank conflicts




**Eliminate Inter-Process Bank Conflicts**

- Make different cores to use different DRAM banks
- Modify the physical frame allocation algorithm of an OS




## Virtual to Physical to DRAM Address

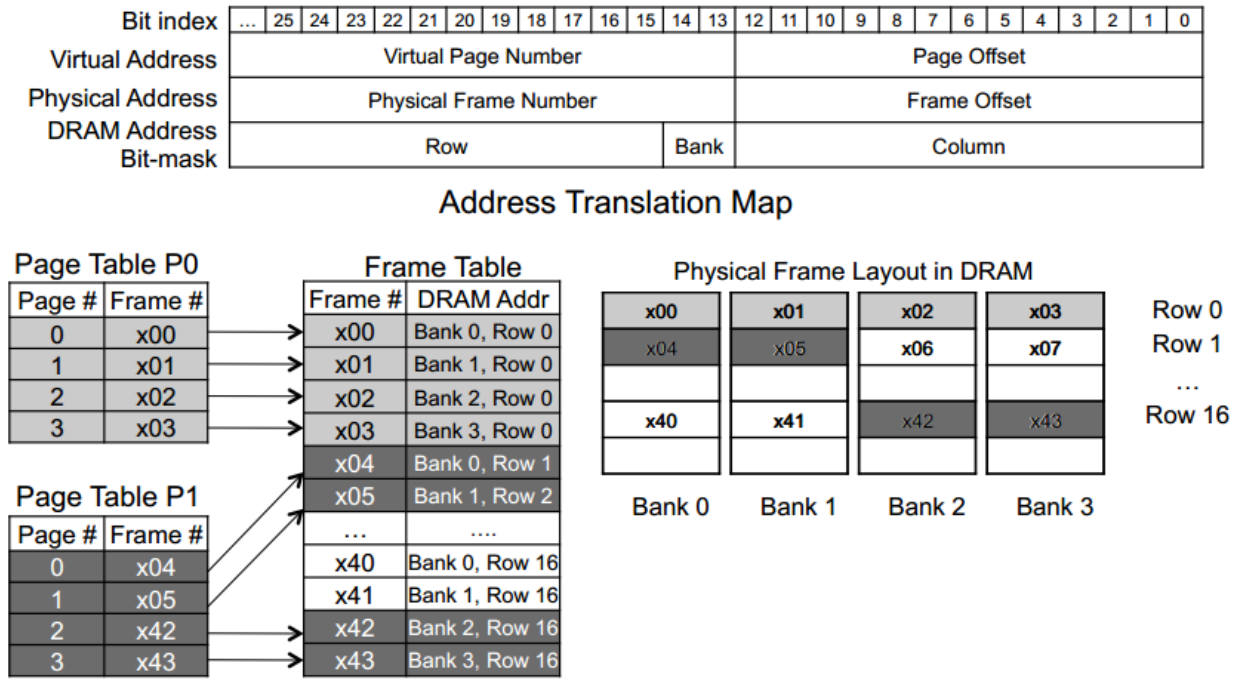



*Figure 2.6 Virtual to physical DRAM*

## Bank-partitioning Frame Allocation

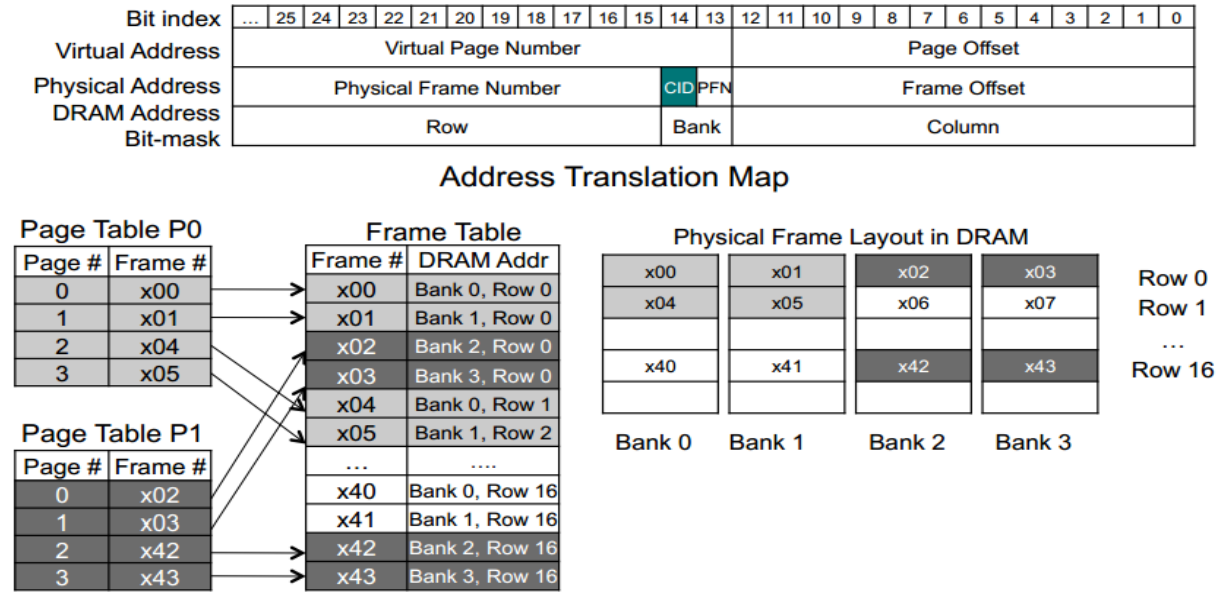



*Figure 2.7 Bank Partition Frame Allocation*

## Bank-Level Parallelism

• Bank-partitioning reduces the number of banks per thread

• Applications with low spatial locality needs many banks to overlap long latency accesses

### Trading off Parallelism and Locality

- Bank Partitioning
  - Isolate streams to preserve locality
  - Good for applications with high spatial locality
- Sub-ranking
  - Controls subsets of rank independently, increases BLP
  - Good for applications with low spatial locality

The two techniques complement each other and improve synergistically

# CONCLUSION

- Combination of bank partitioning and sub-ranking balances locality and parallelism
- It boosts performance and efficiency of the system simultaneously while maintaining fairness
    - 10%, 7%, and 5% throughput gain for HIGH, MIX, and LOW
    - 10%, 9%, and 6% efficiency gain
    - 21.4% DRAM Power reduction on average
    - 15% fairness gain over bank-partitioning only in MIX

## 2.4 Performance Enhancement of NUMA Multiprocessor Systems:

Vipul Kumar Mishra proposes Performance Enhancement of NUMA Multiprocessor systems with On-Demand Memory Migration in 2013 IEEE conference.

- NUMA Architecture is made based on the property of a memory system where access of local memory has different cost than access to Remote memory.
- Process are migrated from one Processor to another in-order to load balance the work on system
- When migrating process there will be reduction in performance and increase in memory access as there will be access across the node.
- Moving all the pages of process along with it is highly inefficient as this will result in lot of migration efforts
- There is proposal to do the migration of memory pages with on-demand request. Whenever some page is requested by Process, it is moved from current node to Node where process has moved.

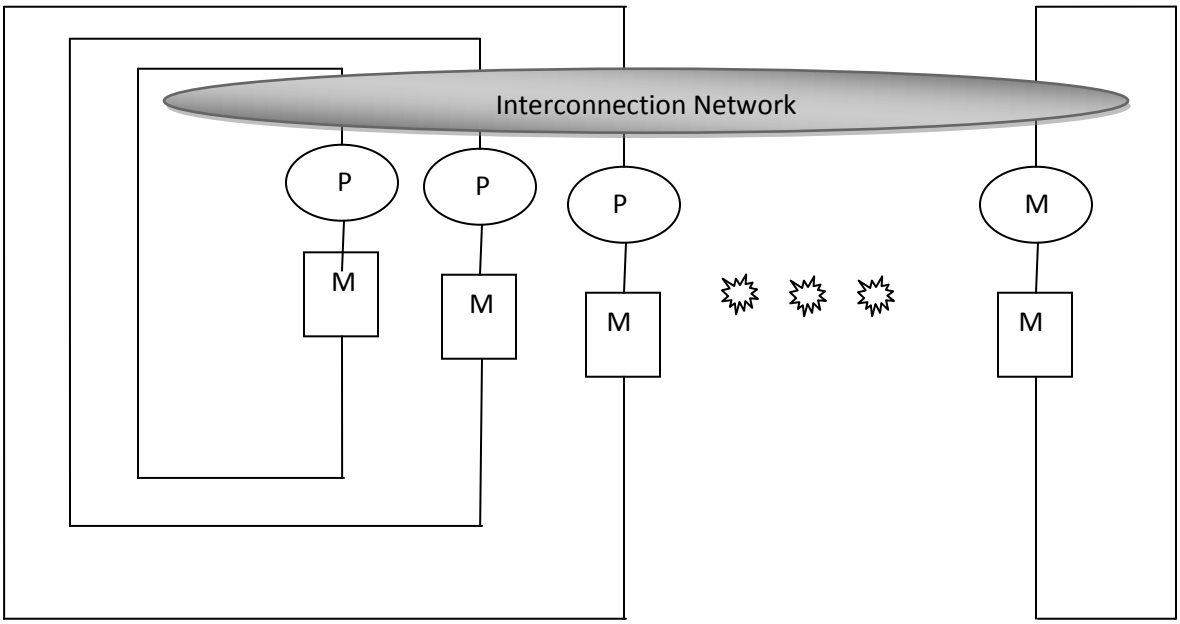



*Figure 2.8* NUMA Architecture

**Related work**

Many improvement algorithms have been proposed to work on process and page migrations for NUMA Architecture. These method has initially focus on static memory placement but this requires prior knowledge of processor memory interconnection architecture, manually binding the threads and allocating memory local to each thread. While in Dynamic schemes data is migrated from where it resides to where it is accessed. There were some proposals to migrate pages using predictive algorithms and aging algorithms.

**Proposed Memory migration policy**

Most important work for performance enhancement of a NUMA architecture based system is to have the most efficient memory migration policy. We know in LINUX operating system, there is already a policy for Copy-On-Write, this is implemented by removing write-access but from the PTE's(Page Table Entry). This means whenever some application will try to write on this page, there will be page fault generated and this page will be copied for our process with write bit set.

Author has proposed similar scheme for Memory migration pages. Both write and read flags from Page Table Entry are removed. There is additional flag being set which is called as non-Touch flag.

Whenever some process will try to access its page, system will check the next-touch flag. If this flag is set, we will copy the pages from current memory node to the process own memory node. After this next touch flag is removed and read/write access is enabled again.

| No of Processes | No of memory migration | Memory migration Along with process | Memory migration on-Demand |
|---|---|---|---|
| 25 | 326 | 326 | 326 |
| 50 | 625 | 630 | 620 |
| 100 | 1207 | 1180 | 1146 |
| 150 | 1650 | 1603 | 1530 |
| 200 | 2230 | 2198 | 2029 |
| 250 | 3012 | 2912 | 2510 |
| 300 | 3852 | 3686 | 3159 |

**Performance Evaluation:**

Performance Evaluation set is done with using system with 8 Nodes, 2 processors and memory access level of 6. It has been observed that if memory migration is done with on-demand policy proposed instead of moving all pages along with process. There is performance improvement of around 16-19%. This intern helps in power optimization of the system.

# Chapter 3

# PROPOSED MEMORY ARCHITECURE

Memory Manager is most important function any operating system and it is core component for its operation. It has great impact on performance of a system along with power consumption.

We understood basics of Memory Manager in Linux operating system in chapter 2. From this design we have the understanding of Node, Zones, and Buddy System.

In this section we will understand the factor which has impact on power consumption while designing a memory manager.

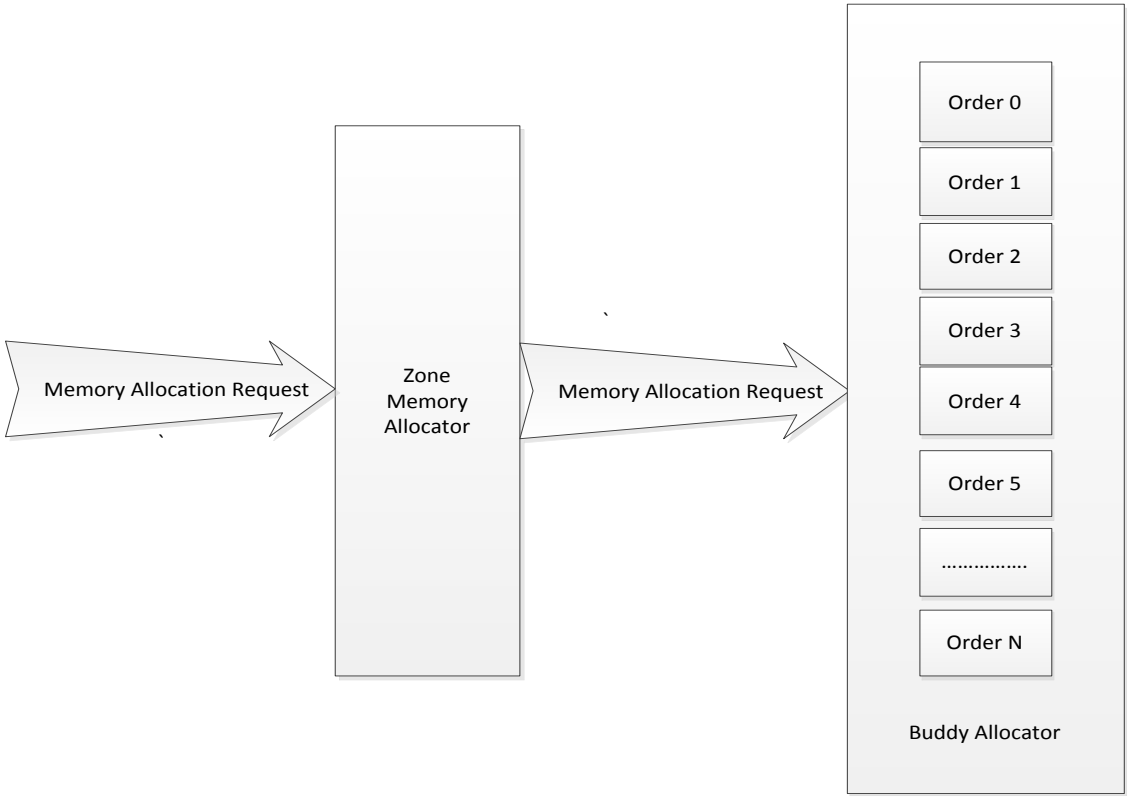



*Figure 3.1 Design of Memory Manager*

As we understand from working of memory manager, every node is divided into zones. Node is divided into zones because of the physical limitation of memory. There are certain pages which cannot be used by kernel for all activities and Zoning is done based on that. Memory allocation component of memory manager which are active during allocation includes Slab Allocator, Buddy Allocator. Slub allocator work over buddy allocator and hence it is not affecting physical management of pages directly.

Memory allocation is done from zones based on buddy system. Buddy system is memory allocation algorithm which divides the memory in blocks of different order and any request of memory is satisfied from block with most nearest matching size.

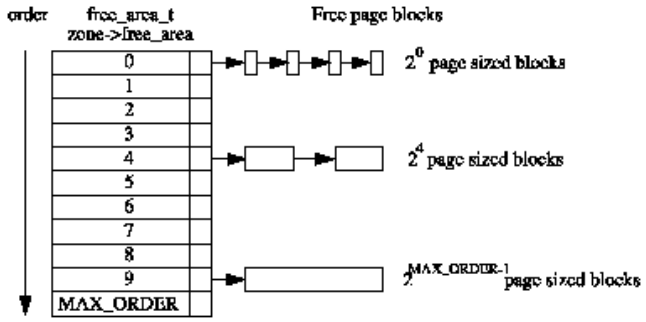



*Figure 3.2 Zone to Buddy Page order Mapping*

As we understand from above Picture, buddy system is spread across the Zone; Memory allocator will try to allocate memory from most matching order block without consideration of memory location of the block.

If physically zone is spread across multiple blocks, memory manager will prefer to allocate the memory to most matching order even if this results in allocations getting scatter across the memory.

This is major problem we can observe in any system with very large memory. Let's consider a system with 2 GByte of memory. In a 32 bit system zone allocation will be Zone DMA – 16 MByte, Zone NORMAL – 880 MByte, Zone HighMem – 1152 MByte.

In our test we have taken a system with 2 GByte memories -- DDR3 RAM. It is having 4 512 MByte memory together.

Now our Zone Normal is split across 2 Memory chips. Initial memory required after boot up of OS at idle is 400 MByte. As memory allocator is trying to find the most matching order of memory from the buddy system, Allocation is split across two memory chip. This means that even when the memory required by our system is fitting to one memory chip we are using the second memory chip.

In DDR3 memory we are having the capability to get in to low power mode but in the above case allocation is on multiple chip and both will keep consuming the power.

In order to work on this problem of allocation spreading to multiple memory blocks or chip we need to work on the architecture of buddy system. Every zone should have the information of memory blocks or chip boundaries and this information should act as input to buddy system in deciding the block from which it has to do the allocation and when should it look for a higher order block instead of taking a matching order block but physically located at position not preferred.

As part of further study on memory allocation spread, we used the system for some time and memory allocation is increased to more than 1.2 GByte of RAM in the system. Now we was able to observe 3 memory chip are getting used. After this we have done the RAM cleaning and closed all the applications. This has reduced the RAM usage to 450 MByte. When we observe at the memory chip usage it is still 3 memory chip modules.

This shows us the second part of problem to work on, with prolonged use memory spread will be across the blocks and it need to be corrected to least possible chip regions. We also need to consider the impact on performance when during memory movement and hence this operation is best to be done when system is in idle state and users work should not get impacted.

## 3.1 Design Of Proposed Architecture

Our goal in this work is to reduce power consumption caused by memory access and making memory management efficient for this purpose.

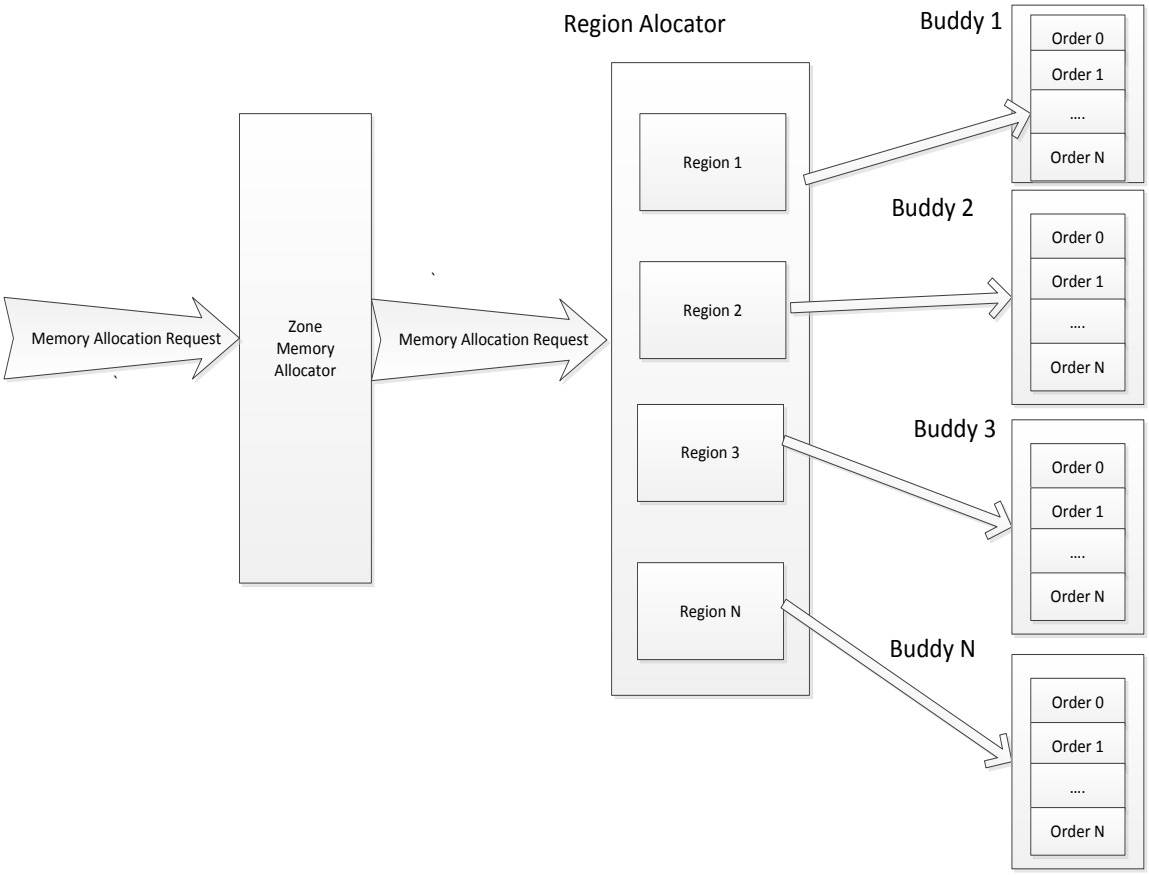



*Figure 3-3 Architecture of Proposed Memory Allocator*

The memory within a node can be divided into regions of memory that can be independently power-managed. That is, chunks of memory can be transitioned to low-power states based on the frequency of references to that region.

For example, if a memory chunk is not referenced for a given threshold amount of time, the hardware (memory controller) can decide to put that piece of memory into a content-preserving low-power state. And of course, on the next reference to that chunk of memory, it will be transitioned back to full-power for read/write operations

Memory Manager can take advantage of this feature by managing the available memory with an eye towards power-savings - i.e., by keeping the memory allocations/references consolidated to a minimum no. of such power-manageable memory regions.

As shown in figure 3.3, initially all allocation request are given to zone memory allocator, from this it will be requested to buddy allocator and allocated memory is given. In our purposed design, our allocations are passed to Region allocator which is division of multiple regions in zone. These regions have their own buddy system and they will be doing allocation of memory for any possible memory order until there is no memory available in the region. When memory is fully allocated from one region than only allocation request can be given to next region.

**Buddy Allocation design**

In order to influence page allocation decisions we need to be able to distinguish page blocks belonging to different zone memory regions within the zones' (buddy) free lists. So, within every free list in a zone, provide pointers to describe the boundaries of zone memory regions and counters to track the number of free page blocks within each region.

Due to the region-wise ordering of the pages in the buddy allocator's free lists, whenever we want to delete a free page block from a free list we need to be able to tell the buddy allocator exactly which migrate type it belongs to. For that purpose, we can use the page's free page migrate type So, while splitting up higher order pages into smaller ones as part of buddy operations, keep the new head pages updated with the correct free page migrate type information.

While merging buddy free pages of a given order to make a higher order page, the buddy allocator might coalesce pages belonging to two different migrate type of that order. So our design explicitly find out the migrate type info of the buddy page and use it while merging the buddies. Also, set the free page migrate type of the buddy to the new one.

The sorted-buddy design for memory power management depends on keeping the buddy free lists region-sorted. And this sorting operation has been pushed to the free logic, keeping the alloc path fast. However, we would like to also keep the free path as fast as

possible, since it holds the zone->lock, which will indirectly affect alloc also. So replace the existing sorting logic used in the free-path, with a new special-case sorting algorithm in order to optimize the free path further. This algorithm uses a bitmap-based radix tree to help speed up the sorting. One of the other main advantages of this design is that it can support large amounts of RAM (up to 2 TB and beyond) quite effortlessly.

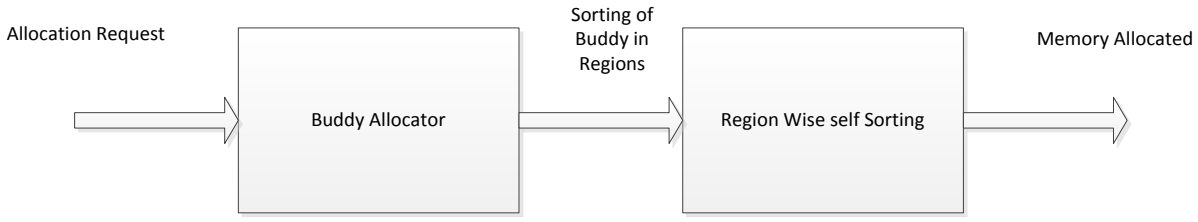



*Figure 3.4 Region buddy self sorting*

While allocating pages from buddy free lists, there could be situations in which we have a ready free page of the required order in a higher numbered memory region, and there also exists a free page of a higher page order in a lower numbered memory region. To make the consolidation logic more aggressive, try to split up the higher order buddy page of a lower numbered region and allocate it, rather than allocating pages from a higher numbered region. This ensures that we spill over to a new region only when we truly don't have enough contiguous memory in any lower numbered region to satisfy that allocation request.

## 3.2 Algorithm for memory allocation

Memory allocation is divided in two parts, in first part is for allocation of initial memory; second part is for compaction of memory when some memory is freed and it results in memory spread to multiple regions.

### 3.2.1 Algorithm for Initial memory allocator

In the Flow diagram below, there is steps in which initial memory allocation is made from Region Allocator, This is explained with all details below.

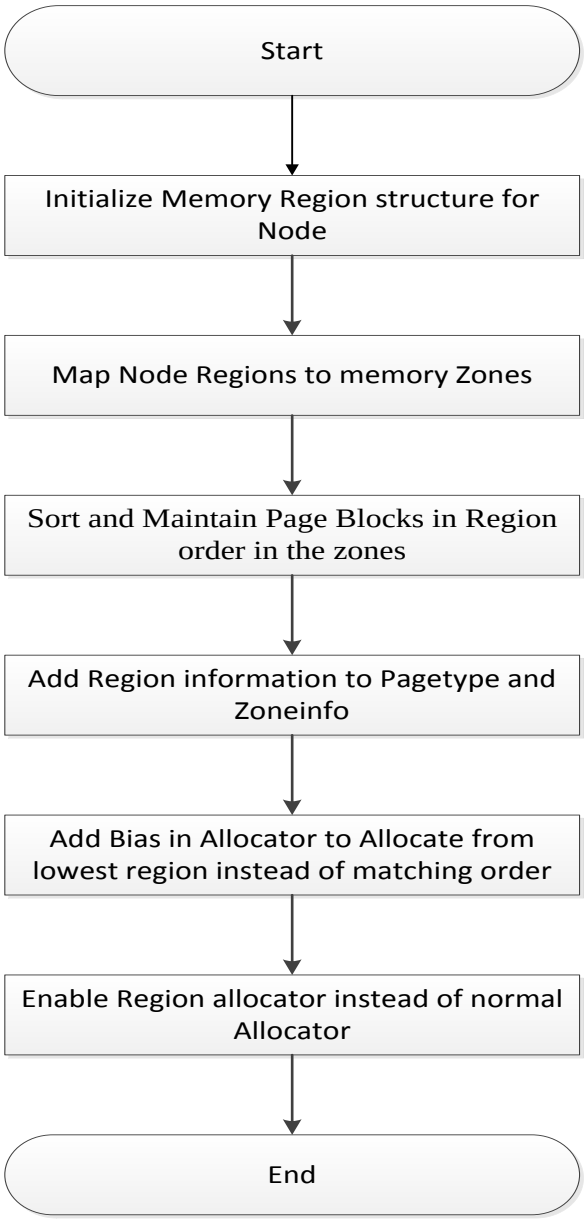



*Figure 3.5 Flow chart for Initial memory allocator*

The first step of initial memory allocator is to teach the MM about the boundaries of these regions - and to capture that info, we introduce a new data-structure called "Memory Regions".

Node is sub-divide into zones based on some well-known constraints. So where do we fit in memory regions in this hierarchy. Instead of artificially trying to fit it into the hierarchy one way or the other, we choose to simply capture the region boundaries in a parallel data-structure, since most likely the region boundaries won't naturally fit inside the zone boundaries or vice-versa.

Once we capture the region boundaries in the memory regions data-structure, we can influence MM decisions at various places, such as page allocation, reclamation etc, in order to perform power-aware memory management.

Most of the MM algorithms (like page allocation etc) work within a zone; hence such a zone-level mapping of the absolute region boundaries will be very useful in influencing the MM decisions at those places

As we can see in the figure 3.3, Node and Zones are default design of a memory allocator. They are the basic building blocks for a memory management system while we added new parts in the end which is further dividing zones in region of memory.

Today, the MM subsystem uses the buddy 'Page Allocator' to manage memory at a 'page' granularity. But this allocator has no notion of the physical topology of the underlying memory hardware, and hence it is hard to influence memory allocation decisions keeping the platform constraints in mind.

So we need to augment the page-allocator with a new entity to manage memory (at a much larger granularity) keeping the underlying platform characteristics and the memory hardware topology in mind.

To that end, introduce a "Memory Region Allocator" as a backend to the existing "Page Allocator". Splitting the memory allocator into a Page-Allocator at front-end and a Region-Allocator at backend.

The flow of memory allocations/frees between entities requesting memory (applications/kernel) and the MM subsystem:

Pages regions Applications <========> Page <========> Memory Region

Since the region allocator is supposed to function as a backend to the page allocator, we implement it on a per-zone basis (since the page-allocator is also per-zone).

The sorted-buddy page allocator keeps the buddy free lists sorted region-wise, and tries to pick lower numbered regions while allocating pages. The idea is to allocate regions in the increasing order of region number. Propagate the same bias to the region allocator as well. That is, make it favor lower numbered regions while allocating regions to the page allocator. To do this efficiently, add a bitmap to represent the regions in the region allocator, and use bitmap operations to manage these regions and to pick the lowest numbered free region efficiently.

Now that we have built up an infrastructure that forms a "Memory Region Allocator", connect it with the page allocator. To entities requesting memory, the page allocator will function as a front-end, whereas the region allocator will act as a back-end to the page allocator. Implement the flow of free pages from the page allocator to the region allocator. When the buddy free lists notice that they have all the free pages forming a memory region, they give it back to the region allocator. When __rmqueue_smallest () comes out empty handed, try to get free pages from the region allocator. If that fails, only then fallback to an allocation from a different migrate type. This helps significantly in avoiding mixing of allocations of different migrate types in a single region. Thus it helps in keeping entire memory regions homogeneous with respect to the type of allocations.

The free page migrate type is used to determine which free list a given page should be added to, upon getting freed. To ensure that the page goes to the right free list, set the free page migrate type of all the pages of a region, when allocating free pages from the region

allocator. This helps ensure that upon freeing the pages or during buddy expansion, the pages are added back to the free lists of the migrate type for which the pages were originally requested from the region allocator.

We would like to maintain memory regions such that all memory pertaining to given a memory region serves allocations of a single migrate type. We don't want to permanently mix allocations of different migrate types within the same region. So, when allocating a region from the region allocator to the page allocator, set the page block migrate type of all that memory to the migrate type for which the page allocator requested memory. Note that this still allows temporary sharing of pages between different migrate types; it just ensures that there is no permanent mixing of migrate types within a given memory region. An important advantage to be noted here is that the region allocator doesn't have to manage memory in a granularity lesser than a memory region, in any situation. This is because the free page migrate type and the fallback mechanism allows temporary sharing of free memory between different migrate types when the system is short on memory, but eventually all the memory gets freed to the original migrate type (because we set the page block migrate type of all the free pages appropriately when allocating regions). This greatly simplifies the design of the region allocator, since it doesn't have to keep track of memory in smaller chunks than a memory region.

Currently, whenever the page allocator notices that it has all the free pages of a given memory region, it attempts to return it back to the region allocator. This strategy is needlessly aggressive and can cause a lot of back and forth between the page-allocator and the region-allocator. More importantly, it can potentially completely wreck the benefits of having a region allocator in the first place - if the buddy allocator immediately returns free pages of memory regions to the region allocator, it goes back to the generic pool of pages. So, in future, depending on when the next allocation request arrives for this particular migrate type, the region allocator might not have any free regions to hand out, and hence we might end up falling back to free pages of other migrate types. Instead, if the page allocator retains a few regions as a cache for every migrate type, we will have higher chances of avoiding fallbacks to other migrate types.

So, don't return all free memory regions (in the page allocator) to the region allocator. Keep at least one region as a cache, for future use.

## Algorithm for initial memory allocator

Step 1: Start

Step 2: Declare structure for Regions in memory as part of Node structure

*Struct node_mem_region*

Step 3: Initialize Region of memory during Node Initialization

***for each*** *page in the Node* ***do***

*set Start and End Page Frame for the Region*

***Done***

Step 4: Define Mapping of memory area of Region and Zones within a Node

***for each*** *Zone in the Node* ***do***

*set Start and End Page Frame for the Zone*

***for each*** *Region in the Node* ***do***

*Update zone region start and end Page Frame*

***Done***

***Done***

Step 5: Sort and Maintain Page Blocks in Region order in the zones

Add_to_Freelist_of_pages()

{

Region_id = get_region_id( page )

Add_page_block_to_Region_List(Region_id)

}

Step 6: Print all the Region information in memory statics in Pagetypeinfo, Zoneinfo to accurately display the memory changes

Flag_show_print()

{

**For** each region in zone **do**

**For** Each order in region **do**

Print Region free blocks

**done**

**done**

}

Step 7: Add the bias in allocator to allocate memory from lowest region instead of most matching buddy order.

__rmqueue_smallest

{

**If true** Select_most_matching_page(Region, order) **than**

Return page;

**Else** Select_page_in_same_region_but_higher_order **than**

Return Page;

**end**

```
}
```

Step 8: Connect Region Allocator with Page allocator to allocate memory from Regions

```
zone_init_free_lists_late
{
        Initalize_zone_region_allocator(); // Instead of normal zone allocator
}
```

### 3.2.2 Algorithm for Memory Compaction

To enhance memory power-savings, we need to be able to completely evacuate lightly allocated regions, and move those used pages to lower regions, which would help consolidate all the allocations to a minimum no. of regions. This can be done using some of the memory compaction and reclaim algorithms.

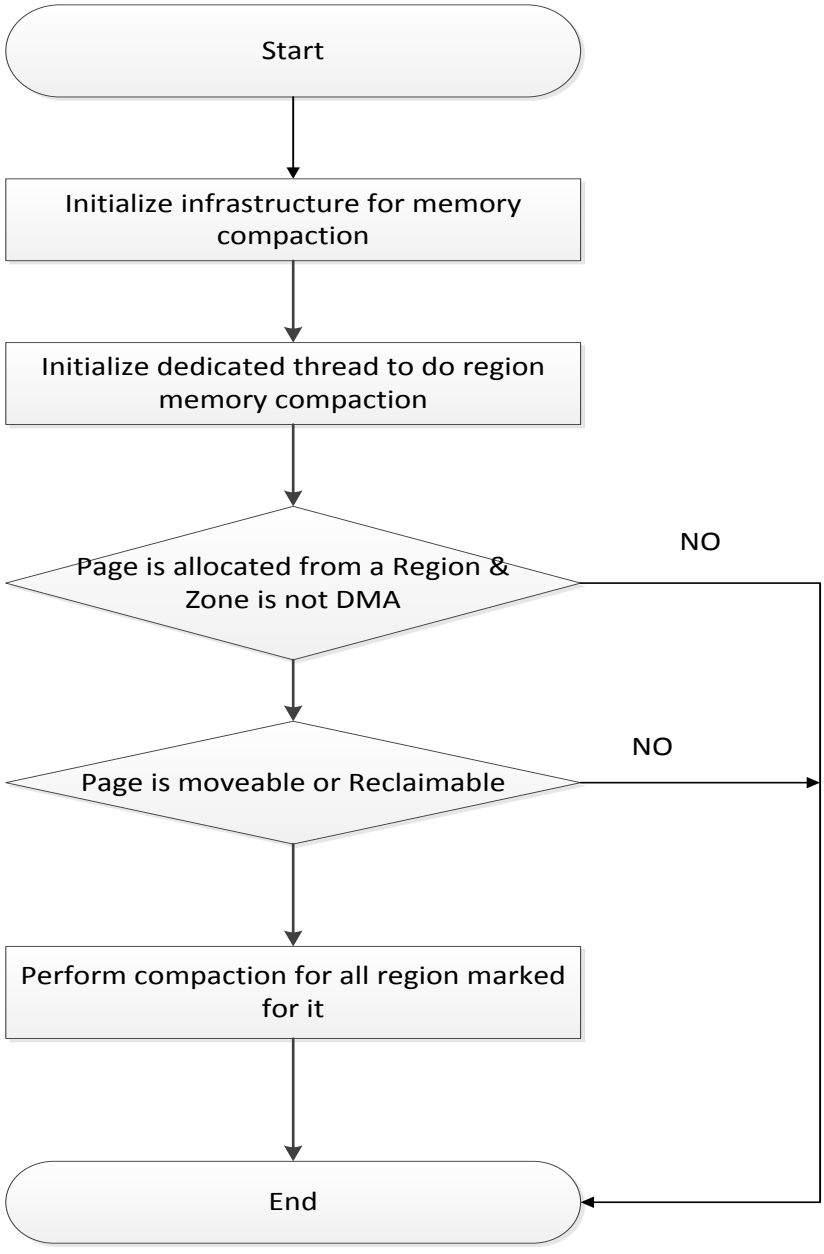



*Figure 3.6 Flow Diagram for Memory compaction algorithm*

Develop such an infrastructure to evacuate memory regions completely. The traditional compaction algorithm uses a PFN walker to get free pages for compaction. But this would be way too costly for us. So we do a pfn walk only to isolate the used pages, but to get free pages, we just depend on the fast buddy allocator itself.

We are careful to abort the compaction run when the buddy allocator starts giving free pages in this region itself or higher regions

To further increase the opportunities for memory power savings, we can perform targeted compaction to evacuate lightly-filled memory regions. For this purpose, introduce a dedicated per-node kthread to perform the targeted compaction work. Our "kmempowerd" kthread uses the generic kthread-worker framework to do most of the usual work all kthread needs to do.

On top of that, this kthread has the following infrastructure in place, to perform the region evacuation. A work item is instantiated for every zone. Accessible to this work item is a spin-lock protected bitmask, which helps us indicate which regions have to be evacuated. The bits set in the bitmask represent the zone-memory-region number within that zone that would benefit from evacuation.

The operation of the "kmempowerd" kthread is quite straight-forward: it makes a local copy of the bitmask (which represents the work it is supposed to do), and performs targeted region evacuation for each of the regions represented in that bitmask. When it's done, it updates the original bitmask by clearing those bits, to indicate that the requested work was completed.

While the kthread is going about doing its duty, the original bitmask can be updated to indicate the arrival of more work. So once the kthread finishes one round of processing, it re-examines the original bitmask to see if any new work had arrived in the meantime, and does the corresponding work if required. This process continues until the original bitmask becomes empty.

Now that we have a dedicated kthread in place to perform targeted region evacuation, add and export a mechanism to queue work to the kthread. Adding work to kmempowerd is

very simple: just set the bits corresponding to the region numbers that we want to evacuate, and queue the work item to the thread.

**Algorithm for Memory Compaction**

Step 1: Start

Step 2: Add infrastructure to evacuate memory regions using compaction

Step 3: Add a kthread to perform targeted compaction for memory power man

**For** each region marked for evacuation **do**

Evacuate the region of memory

Clear Evacuation bit from mask

**Done**

Step 4: Trigger for call of Evacuation thread based on the criteria of region compaction

If page is allocated from a region

if page zone is not DMA

if page is Moveable or Reclaimable

Trigger thread of compaction

Done

Done

Done

# Chapter 4

# EXPERIMENT SETUP AND RESULT ANALYSIS

Our goal is to show that region based memory allocation mechanism is capable of reducing power consumption with minimal performance overheads. Our goal is to save power consumption by prioritizing accesses to the low power regions via region based allocation and defragmentation of memory to lowest regions. In order to show the true benefits of region access prioritization (zoning), we simulated an ideal case, initially memory was allocated to the applications and system so that it is within the first region of memory allocation.

**Configuration of System under Test:**

Intel Core 2 Duo + 2 GByte of DDR2 Main Memory [ 1 GByte + 1 GByte ] + 160 GByte HDD + 1280x800 Resolution.
Operating system: Ubuntu 14.04
Linux kernel: 3.12.0

Below are the steps to setup the system:

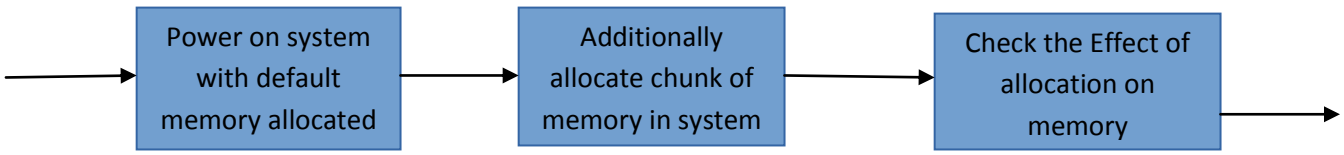



*Figure 4.1 Setup of system for test*

This design of verification helps us to test the system in real time stress. First part of algorithm help us in making the allocation always from the lowest region, With first experiment we will be able to test and confirm if the allocation of memory happens to

prefer lowest region or it is scatter across the regions.

## System under test without any optimization

1. Memory Allocation from a system on First boot

| Zone | Region ID | Total Memory | Available Memory |
|---|---|---|---|
| DMA | Region 0 | 3998 | 2378 |
| **DMA32** | **Region 0** | **28673** | **1024** |
| DMA32 | Region 1 | 32768 | 0 |
| DMA32 | Region 2 | 32768 | 0 |
| DMA32 | Region 3 | 32768 | 0 |
| **DMA32** | **Region 4** | **32768** | **776** |
| **DMA32** | **Region 5** | **32768** | **6288** |
| **DMA32** | **Region 6** | **32768** | **245** |
| **DMA32** | **Region 7** | **32768** | **22843** |
| DMA32 | Region 8 | 32768 | 32768 |
| DMA32 | Region 9 | 32768 | 32768 |
| DMA32 | Region 10 | 32768 | 32768 |
| DMA32 | Region 11 | 32768 | 32768 |
| DMA32 | Region 12 | 32768 | 32768 |
| DMA32 | Region 13 | 32768 | 32768 |
| DMA32 | Region 14 | 32768 | 32768 |
| DMA32 | Region 15 | 30316 | 17408 |

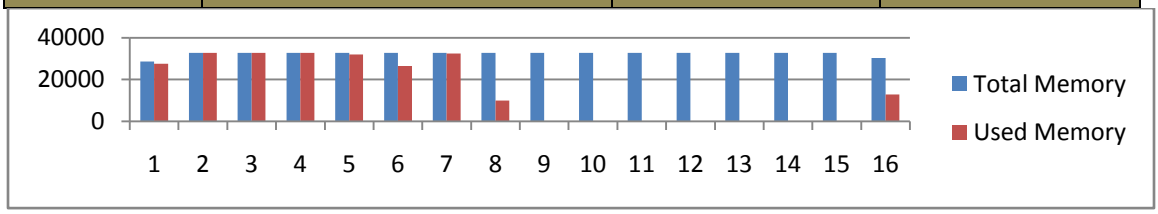



*Figure 4.2 Memory Allocation on First boot*

**Inference from the results**

- **This clearly shows on the first boot itself we are able to see memory is scatter across multiple regions.**
- **We can see the regions specially region 0,4,5,6,7. Memory is allocated in each region but not fully taken.**

2. Memory Allocation from a system for addition 256 MByte

| Zone | Region ID | Total Memory | Available Memory |
|---|---|---|---|
| DMA | Region 0 | 3998 | 1956 |
| **DMA32** | **Region 0** | **28673** | **1024** |
| **DMA32** | **Region 1** | **32768** | **0** |
| **DMA32** | **Region 2** | **32768** | **1** |
| **DMA32** | **Region 3** | **32768** | **0** |
| **DMA32** | **Region 4** | **32768** | **778** |
| **DMA32** | **Region 5** | **32768** | **6050** |
| **DMA32** | **Region 6** | **32768** | **147** |
| **DMA32** | **Region 7** | **32768** | **4579** |
| **DMA32** | **Region 8** | **32768** | **0** |
| **DMA32** | **Region 9** | **32768** | **0** |
| **DMA32** | **Region 10** | **32768** | **0** |
| DMA32 | Region 11 | 32768 | 18408 |
| DMA32 | Region 12 | 32768 | 32768 |
| DMA32 | Region 13 | 32768 | 32768 |
| DMA32 | Region 14 | 32768 | 32768 |
| DMA32 | Region 15 | 30316 | 17408 |

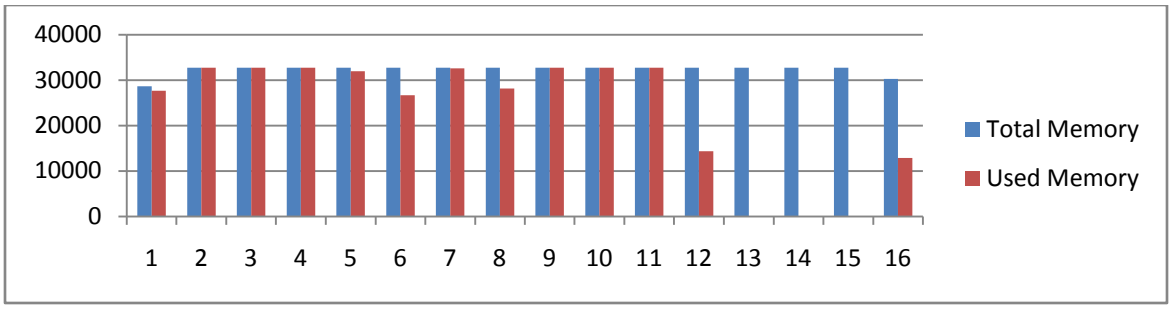



Figure 4.3 Memory Allocation for additional 256 MByte

**Inference from the results**

- **On additional allocation of memory, it is scatter across multiple region of memory.**
- **We can see the regions specially region 0-10. Memory is allocated in each region but not fully taken.**

## System under test with optimization of Memory Management

3. Memory Allocation from a system on First boot

| Zone | Region ID | Total Memory | Available Memory |
|---|---|---|---|
| DMA | Region 0 | 3998 | 2357 |
| DMA32 | Region 0 | 28673 | 1024 |
| **DMA32** | **Region 1** | **32768** | **0** |
| **DMA32** | **Region 2** | **32768** | **0** |
| **DMA32** | **Region 3** | **32768** | **0** |
| **DMA32** | **Region 4** | **32768** | **0** |
| **DMA32** | **Region 5** | **32768** | **0** |
| **DMA32** | **Region 6** | **32768** | **12377** |
| **DMA32** | **Region 7** | **32768** | **32768** |
| DMA32 | Region 8 | 32768 | 32768 |
| DMA32 | Region 9 | 32768 | 32768 |
| DMA32 | Region 10 | 32768 | 32768 |
| DMA32 | Region 11 | 32768 | 32768 |
| DMA32 | Region 12 | 32768 | 32768 |
| DMA32 | Region 13 | 32768 | 32768 |
| DMA32 | Region 14 | 32768 | 32768 |
| DMA32 | Region 15 | 30316 | 20019 |

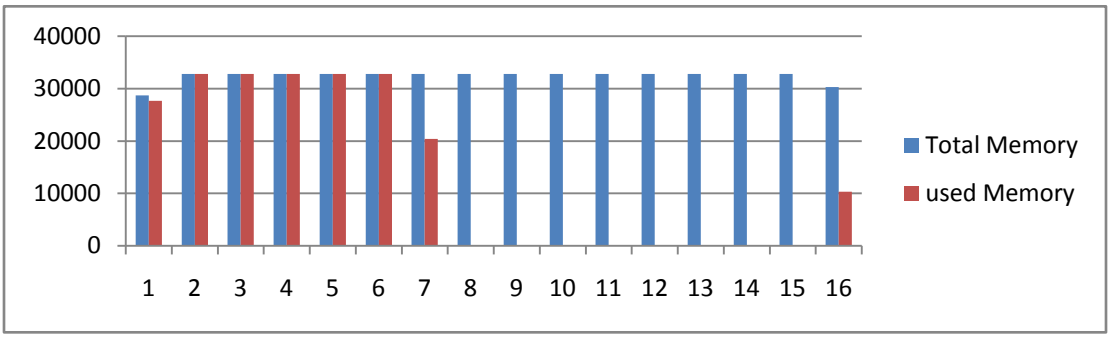



Figure 4.4 Memory Allocation for addition 256 MByte

**Inference from the results**

- **This clearly shows on the first boot itself we are able to see memory is optimized to be limited to least regions.**
- **We can see the regions specially region 0-6.**

4. Memory Allocation from a system for addition 256 MByte

| Zone | Region ID | Total Memory | Available Memory |
|---|---|---|---|
| DMA | Region 0 | 3998 | 1958 |
| DMA32 | Region 0 | 28673 | 1024 |
| **DMA32** | **Region 1** | **32768** | **0** |
| **DMA32** | **Region 2** | **32768** | **0** |
| **DMA32** | **Region 3** | **32768** | **0** |
| **DMA32** | **Region 4** | **32768** | **0** |
| **DMA32** | **Region 5** | **32768** | **0** |
| **DMA32** | **Region 6** | **32768** | **0** |
| **DMA32** | **Region 7** | **32768** | **0** |
| **DMA32** | **Region 8** | **32768** | **0** |
| **DMA32** | **Region 9** | **32768** | **0** |
| **DMA32** | **Region 10** | **32768** | **10441** |
| DMA32 | Region 11 | 32768 | 32768 |
| DMA32 | Region 12 | 32768 | 32768 |
| DMA32 | Region 13 | 32768 | 32768 |
| DMA32 | Region 14 | 32768 | 32768 |
| DMA32 | Region 15 | 30316 | 20019 |

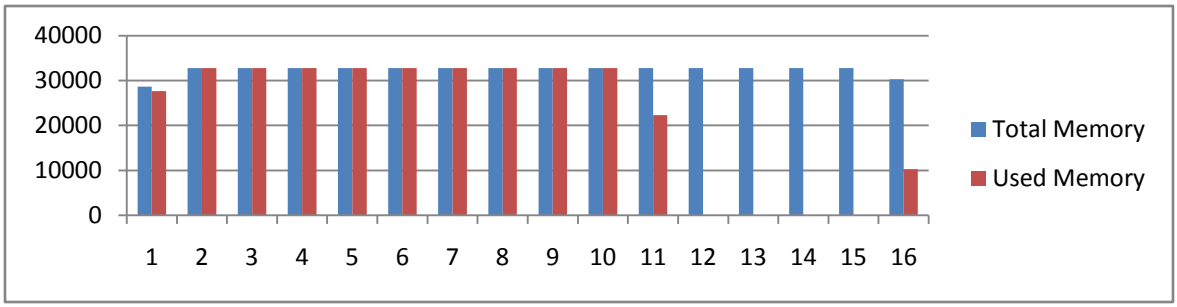



Figure 4.5 Memory Allocation for addition 256 MByte

**Inference from the results**

- **On additional allocation of memory, it is still being taken from the least number of regions.**
- **We can see the regions specially region 0-10. Memory is allocated in each region and fully utilized.**

**Second Experiment**

First experiment was to show that memory usage and allocation is always done from the lowest region in case of power optimized memory allocation.

In the final check, we need to see the memory usage of a system which is in use for quite some time. This means how our algorithm can make the power reduction if system is having memory allocation and de-allocation ongoing.

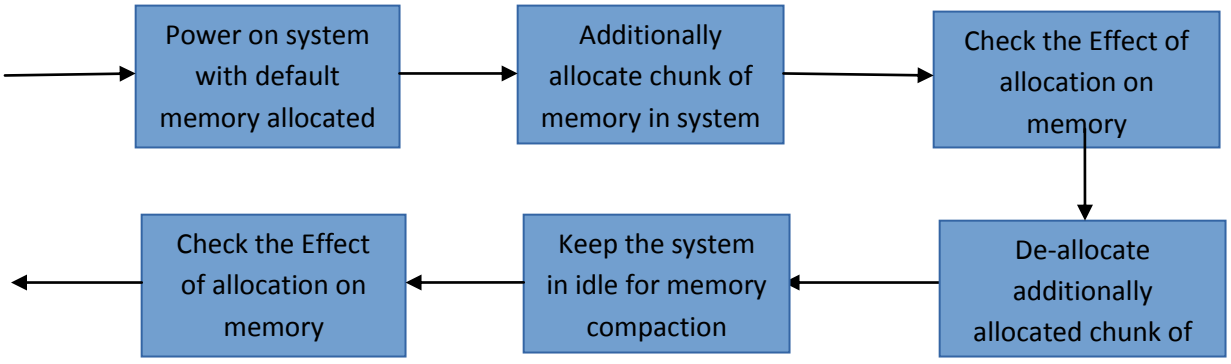



Figure 4.6 *Memory Allocation to test compaction*

## System under test without any optimization

5. Memory De-allocation from a system for additional 256 MByte memory

| Zone | Region ID | Total Memory | Available Memory |
|---|---|---|---|
| DMA | Region 0 | 3998 | 2151 |
| DMA32 | Region 0 | 28673 | 1024 |
| **DMA32** | **Region 1** | **32768** | **0** |
| **DMA32** | **Region 2** | **32768** | **2** |
| **DMA32** | **Region 3** | **32768** | **0** |
| **DMA32** | **Region 4** | **32768** | **770** |
| **DMA32** | **Region 5** | **32768** | **5888** |
| **DMA32** | **Region 6** | **32768** | **162** |
| **DMA32** | **Region 7** | **32768** | **12956** |
| **DMA32** | **Region 8** | **32768** | **16388** |
| **DMA32** | **Region 9** | **32768** | **16395** |
| **DMA32** | **Region 10** | **32768** | **16380** |
| **DMA32** | **Region 11** | **32768** | **25967** |
| DMA32 | Region 12 | 32768 | 32768 |
| DMA32 | Region 13 | 32768 | 32768 |
| DMA32 | Region 14 | 32768 | 32768 |
| DMA32 | Region 15 | 30316 | 17408 |

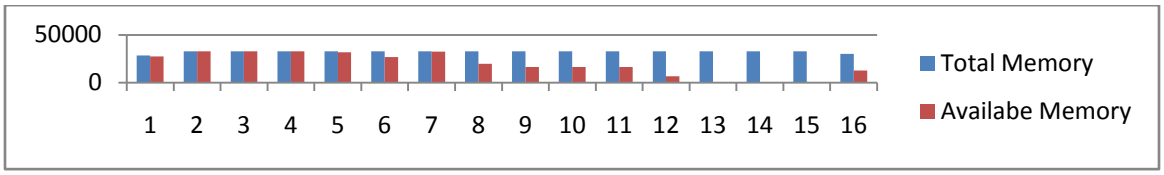



Figure 4.7 Memory De-allocation for additional 256 MByte

### Inference from the results

- **On de-allocating additional allocation of memory, it is scatter from all the regions.**
- **This clearly indicates that default memory allocation algorithm does not care for the fragmentation of memory and confining it to least regions.**

## System under test with optimization of Memory Management

6. Memory De-allocation from a system for additional 256 MByte memory

| Zone | Region ID | Total Memory | Available Memory |
|---|---|---|---|
| DMA | Region 0 | 3998 | 2067 |
| DMA32 | Region 0 | 28673 | 1024 |
| **DMA32** | **Region 1** | **32768** | **0** |
| **DMA32** | **Region 2** | **32768** | **0** |
| **DMA32** | **Region 3** | **32768** | **0** |
| **DMA32** | **Region 4** | **32768** | **0** |
| **DMA32** | **Region 5** | **32768** | **0** |
| **DMA32** | **Region 6** | **32768** | **0** |
| **DMA32** | **Region 7** | **32768** | **2794** |
| **DMA32** | **Region 8** | **32768** | **7423** |
| DMA32 | Region 9 | 32768 | 32768 |
| DMA32 | Region 10 | 32768 | 32768 |
| DMA32 | Region 11 | 32768 | 32768 |
| DMA32 | Region 12 | 32768 | 32768 |
| DMA32 | Region 13 | 32768 | 32768 |
| DMA32 | Region 14 | 32768 | 32768 |
| DMA32 | Region 15 | 30316 | 20019 |

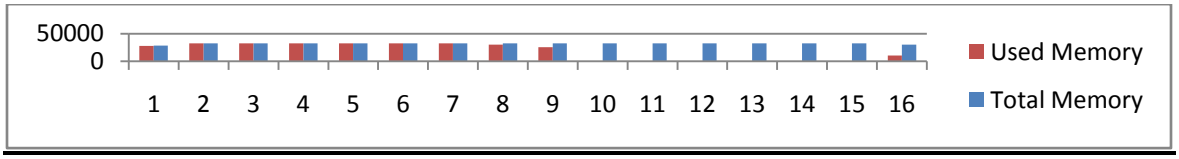



Figure 4.8 Memory De-allocation for additional 256 MByte

### Inference from the results

- **On de-allocating additional allocation of memory, it is compacted to least no. of regions.**
- **This clearly indicates that our memory allocation algorithm care for the fragmentation of memory and confining it to least regions.**

**Configuration of System under Test:**

Intel Core I 5 + 6 GByte of DDR3 Main Memory [4 GByte + 2 GByte] + 512 GByte HDD + 1280x800 Resolution.

Operating system: Ubuntu 12.04

Linux kernel: 3.12.0

## System under test without any optimization

7. Memory Allocation from a system on First boot

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 112898 |
| DMA32 | Region 1 | 130560 | 512 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 18092 |
| NORMAL | Region 0 | 131072 | 115742 |
| NORMAL | Region 1 | 131072 | 0 |
| NORMAL | Region 2 | 131072 | 0 |
| NORMAL | Region 3 | 131072 | 0 |
| NORMAL | Region 4 | 131072 | 0 |
| NORMAL | Region 5 | 130559 | 29183 |

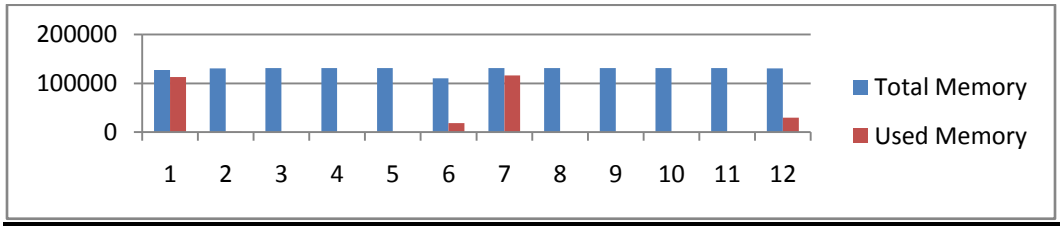



Figure 4.9 Memory Allocation on First boot

**Inference from the results**

- **This clearly shows on the first boot itself we are able to see memory is scatter across multiple regions.**

8. Memory Allocation from a system for addition 256 MByte

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 124996 |
| DMA32 | Region 1 | 130560 | 52765 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 18092 |
| NORMAL | Region 0 | 131072 | 129006 |
| NORMAL | Region 1 | 131072 | 53895 |
| NORMAL | Region 2 | 131072 | 0 |
| NORMAL | Region 3 | 131072 | 0 |
| NORMAL | Region 4 | 131072 | 0 |
| NORMAL | Region 5 | 130559 | 29183 |

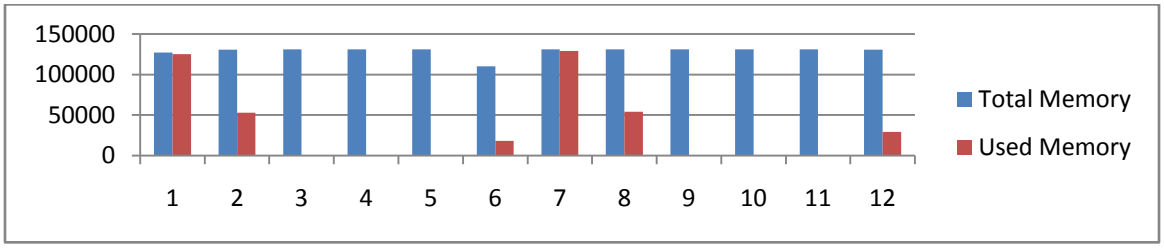



Figure 4.10 Memory Allocation for addition 256 MByte

**Inference from the results**

- **On additional allocation of memory, it is scatter across multiple region of memory.**
- **We can see the regions specially region 0-10. Memory is allocated in each region but not fully taken.**

## System under test with optimization of Memory Management

9. Memory Allocation from a system on First boot

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 116206 |
| DMA32 | Region 1 | 130560 | 3369 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 17415 |
| NORMAL | Region 0 | 131072 | 117128 |
| NORMAL | Region 1 | 131072 | 3906 |
| NORMAL | Region 2 | 131072 | 0 |
| NORMAL | Region 3 | 131072 | 0 |
| NORMAL | Region 4 | 131072 | 0 |
| NORMAL | Region 5 | 130559 | 27048 |

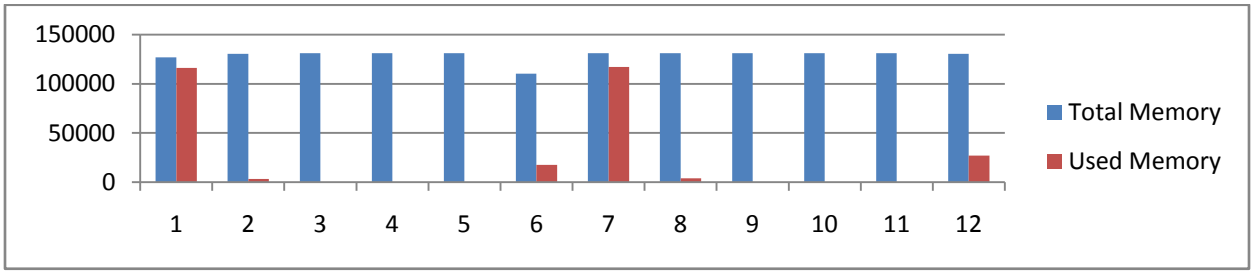



Figure 4.11 Memory Allocation on first boot

**Inference from the results**

- **This clearly shows on the first boot itself we are able to see memory is optimized to be limited to least regions.**
- **We can see the regions specially region 0-6. Memory is allocated in each region before moving to next one.**

10. Memory Allocation from a system for addition 256 MByte

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 126976 |
| DMA32 | Region 1 | 130560 | 57967 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 17415 |
| NORMAL | Region 0 | 131072 | 131072 |
| NORMAL | Region 1 | 131072 | 56041 |
| NORMAL | Region 2 | 131072 | 0 |
| NORMAL | Region 3 | 131072 | 0 |
| NORMAL | Region 4 | 131072 | 0 |
| NORMAL | Region 5 | 130559 | 27048 |

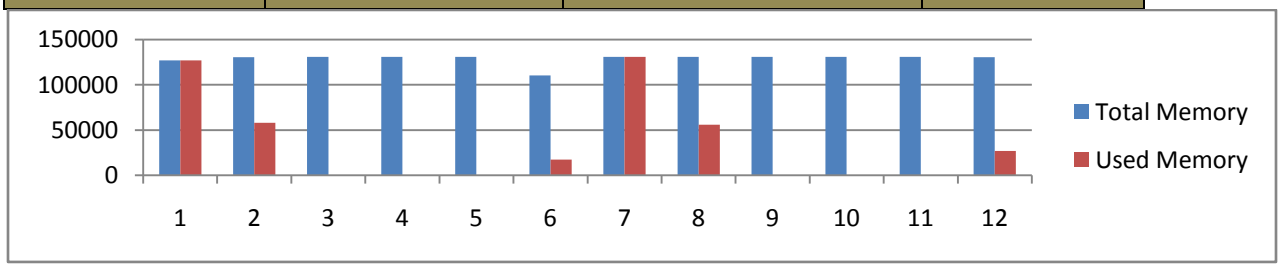



Figure 4.12 Memory Allocation for addition 256 MByte

**Inference from the results**

- **On additional allocation of memory, it is still being taken from the least number of regions.**
- **We can see the regions specially region 0-10. Memory is allocated in each region and fully utilized.**

## Second Experiment

## System under test without any optimization

11. Memory De-allocation from a system for additional 256 MByte memory

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 119003 |
| DMA32 | Region 1 | 130560 | 26887 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 18092 |
| NORMAL | Region 0 | 131072 | 122868 |
| NORMAL | Region 1 | 131072 | 26648 |
| NORMAL | Region 2 | 131072 | 0 |
| NORMAL | Region 3 | 131072 | 0 |
| NORMAL | Region 4 | 131072 | 0 |
| NORMAL | Region 5 | 130559 | 29182 |

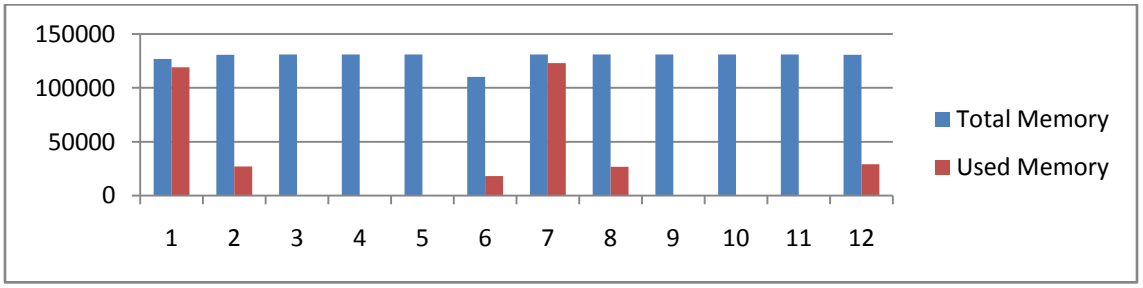



*Figure 4.13* Memory De-allocation for additional 256 MByte

- **On de-allocating additional allocation of memory, it is scatter from all the regions.**
- **This clearly indicates that default memory allocation algorithm does not care for the fragmentation of memory and confining it to least regions.**

## System under test with optimization of Memory Management

12. Memory De-allocation from a system for additional 256 MByte memory

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 126976 |
| DMA32 | Region 1 | 130560 | 25383 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 17415 |
| NORMAL | Region 0 | 131072 | 131072 |
| NORMAL | Region 1 | 131072 | 22270 |
| NORMAL | Region 2 | 131072 | 0 |
| NORMAL | Region 3 | 131072 | 0 |
| NORMAL | Region 4 | 131072 | 0 |
| NORMAL | Region 5 | 130559 | 27048 |

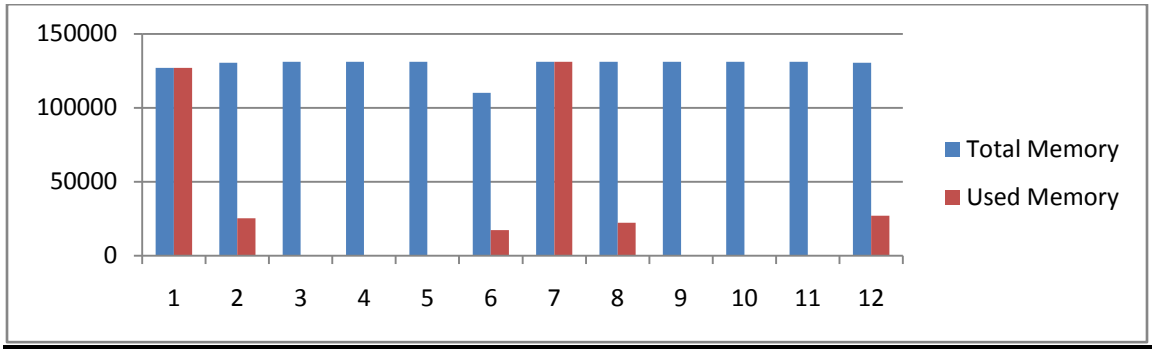



Figure 4.14 Memory De-allocation for additional 256 MByte

**Inference from the results**

- **On de-allocating additional allocation of memory, it is compacted to least no. of regions.**
- **This clearly indicates that our memory allocation algorithm care for the fragmentation of memory and confining it to least regions.**

**Configuration of System under Test:**

Intel Core I 5 + 4 GByte of DDR3 Main Memory [4 GByte] + 512 GByte HDD +

1280x800 Resolution.

Operating system: Ubuntu 13.04

Linux kernel: 3.12.0

## System under test without any optimization

13. Memory Allocation from a system on First boot

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 112898 |
| DMA32 | Region 1 | 130560 | 512 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 18092 |
| NORMAL | Region 0 | 131072 | 115742 |
| NORMAL | Region 1 | 130559 | 29183 |

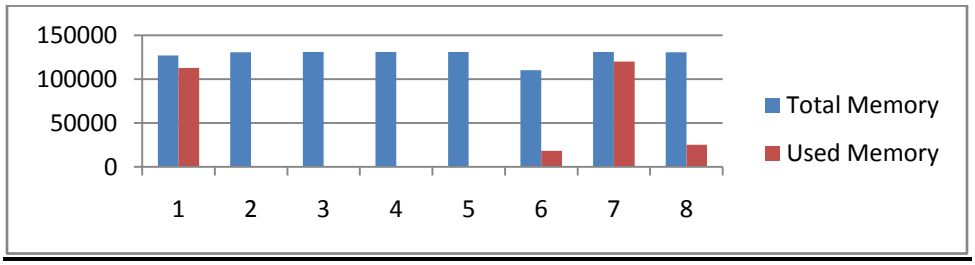



Figure 4.15 Memory De-allocation for First boot

**Inference from the results**

- **This clearly shows on the first boot itself we are able to see memory is scatter across multiple regions.**

14. Memory Allocation from a system for addition 256 MByte

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 124996 |
| DMA32 | Region 1 | 130560 | 52765 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 18092 |
| NORMAL | Region 0 | 131072 | 129006 |
| NORMAL | Region 1 | 130559 | 83078 |

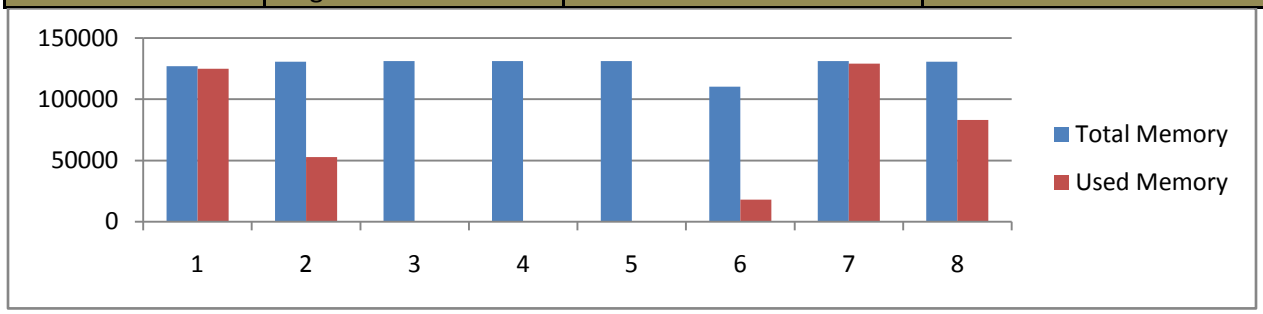



Figure 4.16 Memory Allocation for additional 256 MByte

**Inference from the results**

- **On additional allocation of memory, it is scatter across multiple region of memory.**
- **We can see the regions specially region 0-10. Memory is allocated in each region but not fully taken.**

## System under test with optimization of Memory Management

15. Memory Allocation from a system on First boot

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 116206 |
| DMA32 | Region 1 | 130560 | 3369 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 17415 |
| NORMAL | Region 0 | 131072 | 121034 |
| NORMAL | Region 1 | 130559 | 27048 |

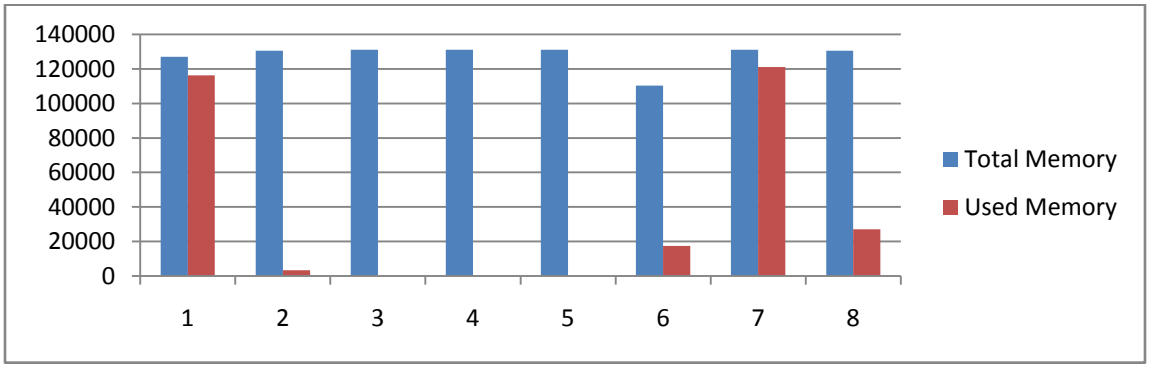



Figure 4.17 Memory Allocation on First boot

**Inference from the results**

- **This clearly shows on the first boot itself we are able to see memory is optimized to be limited to least regions.**
- **We can see the regions specially region 0-6. Memory is allocated in each region before moving to next one.**

16. Memory Allocation from a system for addition 256 MByte

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 126976 |
| DMA32 | Region 1 | 130560 | 57967 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 73456 |
| NORMAL | Region 0 | 131072 | 131072 |
| NORMAL | Region 1 | 130559 | 27048 |

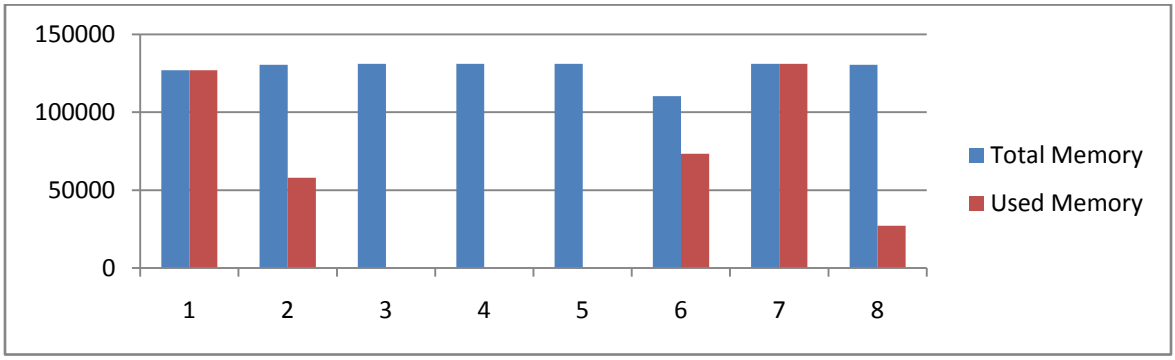



Figure 4.18 Memory Allocation for addition 256 MByte

**Inference from the results**

- **On additional allocation of memory, it is still being taken from the least number of regions.**
- **We can see the regions specially region 0-10. Memory is allocated in each region and fully utilized.**

## Second Experiment

## System under test without any optimization

17. Memory De-allocation from a system for additional 256 MByte memory

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 119003 |
| DMA32 | Region 1 | 130560 | 26887 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 44740 |
| NORMAL | Region 0 | 131072 | 122868 |
| NORMAL | Region 5 | 130559 | 29182 |

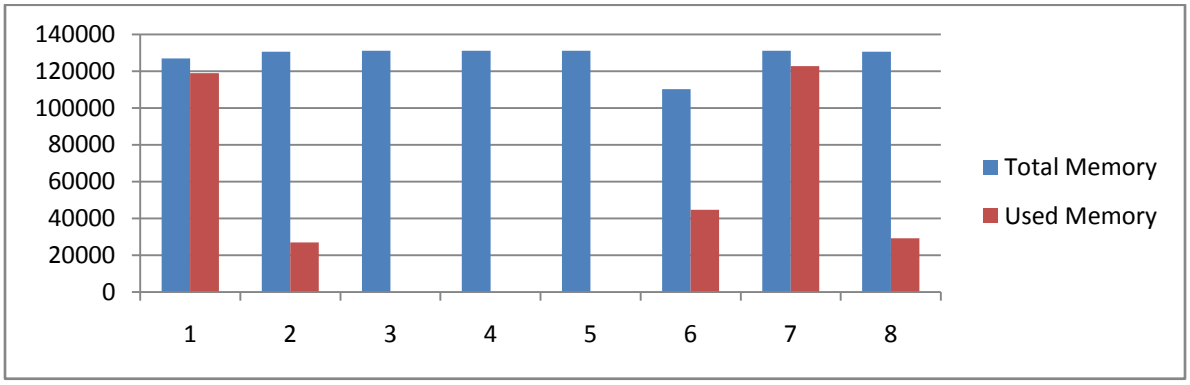



*Figure 4.19* Memory De-Allocation for additional 256 MByte

**Inference from the results**

- **On de-allocating additional allocation of memory, it is scatter from all the regions.**
- **This clearly indicates that default memory allocation algorithm does not care for the fragmentation of memory and confining it to least regions.**

## System under test with optimization of Memory Management

18. Memory De-allocation from a system for additional 256 MByte memory

| Zone | Region ID | Total Memory | Used Memory |
|---|---|---|---|
| DMA | Region 0 | 3997 | 161 |
| DMA32 | Region 0 | 126976 | 126976 |
| DMA32 | Region 1 | 130560 | 25383 |
| DMA32 | Region 2 | 131072 | 0 |
| DMA32 | Region 3 | 131072 | 0 |
| DMA32 | Region 4 | 131072 | 0 |
| DMA32 | Region 5 | 110252 | 39685 |
| NORMAL | Region 0 | 131072 | 131072 |
| NORMAL | Region 1 | 130559 | 27048 |

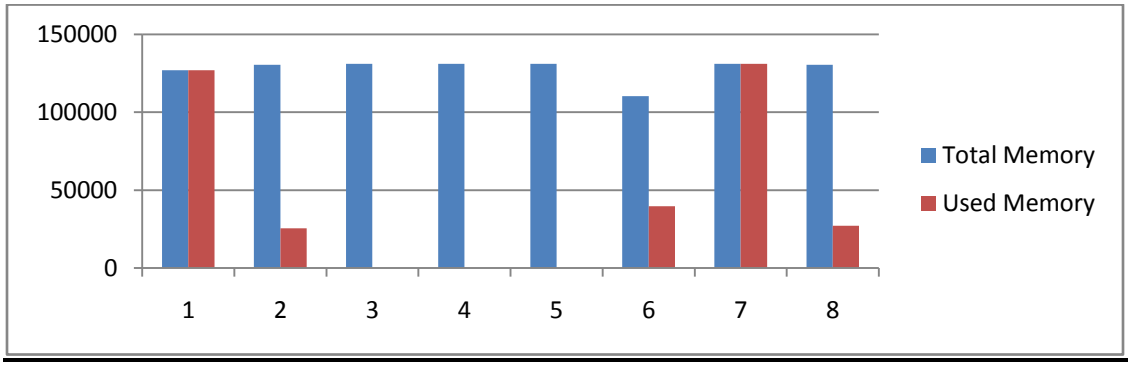



Figure 4.20 Memory De-allocation for additional 256 MByte

**Inference from the results**

- **On de-allocating additional allocation of memory, it is compacted to least no. of regions.**
- **This clearly indicates that our memory allocation algorithm care for the fragmentation of memory and confining it to least regions.**

# Chapter 5

# CONCLUSION & FUTURE WORK

In this project we presented Power management for large scale memory system, a system-level variability-aware solution that adapts to the power variability inherent in a set of DRAM memory modules. With our module we have shown how to restrict usage of memory to the minimum area. This was archived during initial allocation of memory at first. This was to demonstrate the capability of our changes on restricting memory allocation at initial step itself.

With second step, we have shown how to further de-fragment memory usage when system is in use for long time. With long usage every system is expected to de-allocate memory which it was using and this de-allocation occur randomly. Sometime the memory is de-allocated from middle of memory area while at other time same is de-allocated from start of it. We have demonstrated with our changes that we were able to restrict the usage of memory to minimum area again.

Our Main memory is divided into banks. Each of these banks is independently power switchable. In some cases we were able to restrict No. of banks in usage from 6 with normal algorithm to 3 with optimized algorithm. This means improvement in the power usage of the system by almost 30 Percent.

Future work from this project include following, Power optimization from our algorithm are demonstrated on a server and personal computer. These are validated with verification on multiple computers. We need to further demonstrate the algorithm on an embedded system like a mobile device. This will demonstrate the power usage and optimization from our algorithm directly in power domain itself

Also we have worked to keep the de-fragmentation of memory during idle time. This activity takes time, we need to measure the time taken by this activity and also method to optimize this time. Further identification of suitable time may be done by using context awareness utility (widely available in portable devices) as well as in other systems based on past activity logs.

# REFERENCES


1. Balancing DRAM Locality and Parallelism in shared memory CMP systems IEEE 2012, Min Kyu Jeong

2. Performance enhancement of NUMA multiprocessor systems with on-demand memory migration 2013, Mishra, V.K

3. Memory Affinity: Balancing Performance, Power, Thermal and Fairness for Multi-core Systems, IEEE 2013, Gangyong Jia

4. introduces Red-Black Tree Used for Arranging Virtual Memory Area of Linux IEEE 2010, Zhang Haiyang

5. ViPZonE: OS-Level Memory Variability-Driven Physical Address Zoning for Energy Savings Luis Angel D. Bathen, Nikil Dutt, Alex Nicolau

6. Model-based, memory-centric performance and power optimization on NUMA multiprocessors Chunyi Su ; Dong Li ; Nikolopoulos, D.S. ; Cameron, K.W. ; De Supinski, B.R. ; Leon, E.A.

7. A system level memory power optimization technique using multiple supply and threshold voltages Ishihara, T. ; Asada, K.

8. Scratchpad memory-global power optimization Karthika, M. ; Rajasekaran, C. Pattern

9. Performance-power optimization of memory components for complex embedded systems Gebotys, C.H. ; Gebotys, R.J.

10. Memory power models for multilevel power estimation and optimization Schmidt, E. ; von Colln, G. ; Kruse, L. ; Theeuwen, F. ; Nebel, W.

11. Very Large Scale Integration (VLSI) Systems, Device-architecture co-optimization of STT-RAM based memory for low power embedded systems Cong Xu ; Dimin Niu ; Xiaochun Zhu ; Kang, S.H. ; Nowak, M. ; Yuan Xie

12. Digital Object Identifier Low-power multiple-bit upset tolerant memory optimization Seokjoong Kim ; Guthaus, M.R.

13. Particle Swarm Optimization Based Scheme for Low Power March Sequence Generation for Memory Testing Kumar, K.S. ; Kaundinya, S. ; Chattopadhyay, S.

14. Accurate, wideband characterization and optimization of high power LDMOS amplifier memory properties Eron, M. ; Martony, E. ; Fogel, Y. ; Jeckeln, E. ; Hrybenko, M.

15. Analysis and Optimization of Low-Power Passive Equalizers for CPU–Memory Links Ling Zhang ; Wenjian Yu ; Yulei Zhang ; Renshen Wang ; Deutsch, A. ; Katopis, G.A. ; Dreps, D.M. ; Buckwalter, J. ; Kuh, E.S. ; Chung-Kuan Cheng

16. Memory Optimizations For Fast Power-Aware Sparse Computations Malkowski, K. ; Raghavan, P. ; Irwin, M.J.

17. Optimization of programming consumption of silicon nanocrystal memories for low power applications Della Marca, V. ; Masoero, L. ; Molas, G. ; Amouroux, J. ; Petit-Faivre, E. ; Postel-Pellerin, J. ; Lalande, F. ; Jalaguier, E. ; Deleonibus, S. ; De Salvo, B. ; Boivin, P. ; Ogier, J.

18. A novel flash memory cell and design optimization for high density and low power application Huiwei Wu ; Shiqiang Qin ; Yimao Cai ; Poren Tang ; Zhan, Zhan ; Huang, Qianqian ; Ru Huan

19. Memory Access Characterization of Scientific Applications on GPU and Its Implication on Low Power Optimization Wang, Guibin

20. Optimization of memory organization and hierarchy for decreased size and power in video and image processing systems Nachtergaele, L. ; Catthoor, F. ; Balasa, F. ; Franssen, F. ; De Greef, E. ; Samsom, H. ; De Man, H.

21. Power-performance co-optimization of throughput core architecture using resistive memory Goswami, Nilanjan ; Cao, Bingyi ; Li, Tao

22. A global bus power optimization methodology for physical design of memory dominated systems by coupling bus segmentation and activity driven block placement Hua Wang ; Papanikolaou, A. ; Miranda, M. ; Catthoor, F.

23. Power and Area Optimization for Run-Time Reconfiguration System On Programmable Chip Based on Magnetic Random Access Memory Weisheng Zhao ; Belhaire, E. ; Chappert, C. ; Mazoyer, P.

24. Scratchpad memory based power efficient optimization for MPSoC Wei Hu

25. Hardware compilation of application-specific memory-access interconnect Venkataramani, G. ; Bjerregaard, T. ; Chelcea, T. ; Goldstein, S.C.