M.S. THESIS

Enabling User-Controlled Allocations on Hybrid Memory Systems

하이브리드 환경에서 사용자의 제어가 가능한 메모리할당 기법

July 2014

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Cui Wenfeng

Enabling User-Controlled Allocations on Hybrid Memory Systems

하이브리드 환경에서 사용자의 제어가 가능한 메모리할당 기법

지도교수 염 헌영

이 논문을 공학석사 학위논문으로 제출함

2014 년 7 월

서울대학교 대학원

전기.컴퓨터 공학부

최 문봉

Cui Wenfeng의 공학석사 학위논문을 인준함

2014 년 7 월

위 원 장	김 지홍	(인)
부위원장	염 헌영	(인)
위 원	이 제희	(인)

Abstract

Phase-Change Memory (PCM) has received a lot of attention as a next-generation storage component. Because PCM has higher density and lower power consumption as main memory compared with DRAM, hybrid main memory systems are proposed as new models that comprise PCM and DRAM. However, PCM has lower durability and 6-12 times slower write access time than DRAM. When PCM is used as main memory, the high write latency influences performance enormously. To make up these weaknesses in PCM, previous researches focused on automatic page replacement by which pages could be migrated between PCM and DRAM main memory when programs are running based on statistics of the page access patterns. But statistics based page replacement approaches need pattern monitoring overhead and furthermore, proposed hardware driven approaches are hard to deploy because memory controller has to be redesigned and software approaches are not complete solutions because not all page accesses are visible to software. In this paper, we propose to enable user-controlled static memory allocations on hybrid memory systems. Through static memory allocations, programmers can allocate write intensive variables to DRAM to avoid high write access time and wearing of PCM along with energy savings.

Keywords: Phase-change memory, Memory management, Hybrid memory systems, Memory allocations

Student Number: 2012-23962

Contents

List of Figures

List of Tables

0.1 Introduction

With the emergence of many-core CPUs and big data applications, the memory capacity problem is becoming an ever important issue for many platforms. With the advantages of nearly 4x higher density [Ramos et al], and lower idle power consumption than DRAM, PCM has become one of the most promising next-generation memories. PCM shows high potential as a storage cache layer or main memory when used with other low write latency devices [CMU2008survey] due to its comparable read latency with DRAM and the fact that the memory size is more critical to performance than write latency. Energy consumption is also a major issue in the design of data centers and the energy consumed by main memory is one of the dominant parts. Despite its high write latency, PCM shows high performance per price in [Kim et al] when used as the component for both the storage and the main memory even under the assumption of its price 4 times higher than Flash based SSD(Solid State devices) for enterprises. For mobile devices, energy consumption is critical and PCM shows great potential because of its low energy consumption in idle state.Because operating system kernels do not consider hybrid memory systems, PCM and DRAM frames are treated the same when memory page faults occur. This leads inefficient usage of the main memory since DRAM is more suited for write operations than PCM. We focus on the main memory management issue of hybrid memory systems and propose a design and implementation of a new dynamic memory allocation interface with which users can allocate main memory between PCM and DRAM explicitly. We also enable global variables to decide their physical memory frames between PCM and DRAM by kernel modifications. The new interface involves only software modifications and from the point of Glibc(GNU C library) view, it keeps the memory allocation efficiency of Glibc but adds a new feature of hybrid memory allocations in the library. Besides the modification of Glibc, we modified the process layout in the 64bit Linux system to support the

feature of hybrid memory allocations.

Our simulation-based results show that the ability of user-controlled allocations on hybrid main memory systems allows the runtime reduction of 12-19% in the quick sort, 20-35% in the merge sort, 40-45% in the lbm and 1-23% in the original matrix multiplication. Along with runtime reduction, significant reduction of power consumption comes as 33-35% in quick sort, 35-54% in the merge sort, 66-70% in the lbm and 9-40% in the original matrix multiplication.

0.2 Related Work

0.2.1 Statistics based page replacement

In addition to the advantages as stated, PCM has two critical weaknesses compared with DRAM when used as main memories: its low durability of write operations and high write latency. To cope with these two problems, page replacement algorithms have been proposed [Ramos et al] [Lee et al]. Page replacement algorithm proposed in [Ramos et al] which uses [Zhou et al] shows much performance improvement but it is a hardware driven method and requires memory controller to monitor the page access patterns and migrate pages between DRAM and PCM. CLOCK-DWF algorithm proposed in [Lee et al] executes page replacement according to the write frequency to estimate future write references. But the algorithm needs to monitor every access of read/write operation on pages and makes eviction decisions of which page should be evicted out when the DRAM or PCM area is full. It's possible to get each page access operation by *mark_page_accessed* function in the kernel for those pages used as file caching but not for anonymous page accesses. On most hardware-filled TLB platforms (e.g., Intel processors), each page table entry has an access bit, which is automatically set by hardware when the page is accessed [Zhang et al]. By periodically checking and clearing this access bit, one can estimate the access frequency for each

anonymous page. However, the high overhead of scanning all the page table entries in the main memory for statistics poses a new problem. NVMALLOC [Wang et al] proposed a method for exploiting non-volatile memory as a secondary memory partition so that applications can explicitly allocate and manipulate memory regions therein. NVMALLOC calls the memory mapped I/O interface every time for each memory request then users are able to manipulate the returned memory mapped area as they do with the main memory. But the approach is not a suitable solution when PCM is used as the main memory because it is critical to performance when memory mapped system call is executed every time for each request. User level dynamic memory allocation interfaces like *malloc* go through Glibc and the library manages the memory allocation and deallocation between the applicatoin layer and the kernel layer. Considering the high overhead of system calls, the library does not request the memory from the kernel for every request. When a request smaller than 64 megabytes comes, the library request 64 megabytes from the kernel and split the allocated memory for the request and the rest part of the 64 megabytes may be used for other requests. When the memory of the request is to be freed, the library does not return it to the kernel imediately considering new memory requests may arrive soon. For requests larger than 64 megabytes, the library request and return the memory through system calls directly. On hybrid memory systems, we need a mechanism to allocate the memory considering not only the requesting memory sizes but also the kind of memory requested.

0.2.2 Process layout for 64bit Linux systems

On the 64bit Linux systems, the process layout looks like Figure 1. Since the virtual addresses are represented using 48 bits on 64bit systems, the size of the whole virtual memory space is 256 terabytes. The Linux kernel divides the virtual memory space equally into the kernel space and the user space. Those temporal variables created during the execution of functions are located in the “stack” area. Heap is the place

where dynamic memory allocations usually take place. The heap area is managed by *malloc*, *realloc*, and *free* in applications, which may use the *brk* or *sbrk* system calls to adjust its size. Spaces allocated by *brk* or *sbrk* system calls are located in the "Heap" area. Spaces allocated by *mmap* are located in the "Memory Mapping Region". The Heap area is shared by all shared libraries and dynamically loaded modules in a process. The compiled code goes to the "Text Segment" and only read operations occur on this segment. Global variables go to the "BSS Segment" or the "Data Segment", initialized global variables are located in the "Data Segment" and uninitialized global variables are located in the "BSS Segment". The reason to separate these two segment is to get a smaller size of ELF(Executable and Linkable Format) file after compilation since uninitialized data can be set as zeros during the run time when accessed.

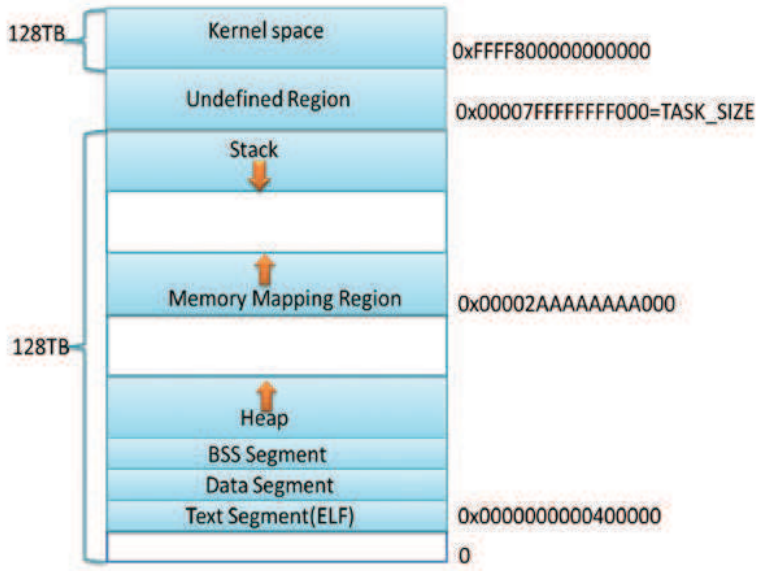


Figure 1 The process layout on 64bit Linux operating systems

0.2.3 Malloc memory allocation interface in Glibc

One of the typical dynamic memory allocations interface is *malloc(int size)* where "size" is the requested memory size. As we can see in Figure 2, *malloc* involves three layers. When users call *malloc*, requests are passed to Glibc and then *sbrk* and *mmap* systems are called by Glibc. Arenas exist in the Glibc library and they are responsible for allocating memories for requests from *malloc*. In the original versions, there is only one main arena. For each memory allocation request, processes will lock the main arena and free the lock after allocations. As the SMP(Symmetric multiprocessing) architecture become popular, the contention among threads degrades the performance of the *malloc* function. Hence, the *malloc* is improved to be adjusted to multithreading environment by adding non-main arenas for the lock contention reduction on the main arena. Non-main arenas are added to a linked list with the main arena. Each arena uses one lock to ensure synchronization among threads. Each process has only one main arena but may have many non-main arenas. The number of non-main arenas increases as the contention among threads increases. When a thread attempts to request a memory space, it looks over if has an arena among thread-local variables; if there is an arena, it tries to lock this arena and if it fails, this thread scans the linked list of arenas to find an arena which is not locked; if all of the arenas are locked, it will create a new non-main arena to reduce contentions among threads. When a thread attempts to free a memory space, it has to wait until the arena which manages the memory space is unlocked. Main arena accesses the "Heap segment" and the "mmap" area. In other words, main arena can request memory by *sbrk* and *mmap* but non-main arenas can only access the 'mmap' area. Users will get continuous virtual memory space if they don't call *sbrk* and *brk* since the main arena will request virtual memory using *sbrk* only. Non-main arenas use *mmap* to get 64 megabytes memory from the kernel in 64bit systems and then slice this large amount of memory as many small number

of allocation units for requests. Since system calls are expensive compared with user space function calls, *malloc* requests a large size memory at one time to reduce the cost of system calls. For the *free* operations for those allocated spaces, Glibc library does not return those spaces to the kernel immediately since other requests probably come afterwards.

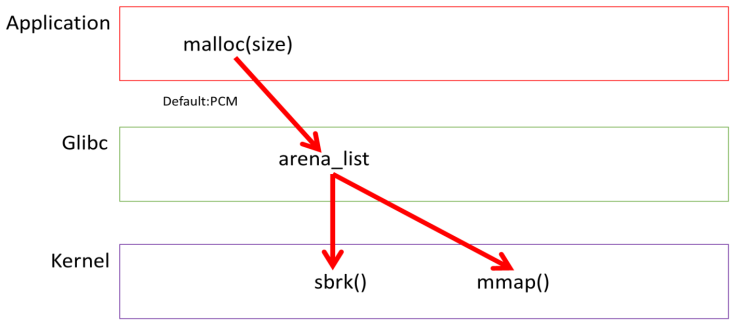


Figure 2 Malloc function path in Glibc 2.12

0.2.4 Physical memory managements in Linux kernels

The physical memory address space is divided into three zones in Linux system: DMA, DMA32 and Normal zone, which serve different functions accordingly. “DMA” zone provides the memory space for those devices in which only 16 megabyte address range is accessible because of the limitation of address lines. Similarly, ’DMA32’ zone provides the memory space for those devices in which 4 gigabyte address range is accessible. The ’Normal’ zone manages the rest of the memory space(equal or greater than 4 gigabyte address space) and provides the memory space for other cases. Each zone maintains a independent buddy allocator management system so making PCM as a separate zone is an obvious way to distinguish DRAM and PCM. By creating a PCM zone, we are able to handle allocated frames when page faults occur.

0.3 Motivations

As was discussed, hardware approaches have difficulties in deployment and software approaches of automatic page migrations have difficulties in monitoring the page access patterns of anonymous pages and the overhead of scanning page table entries. We focus on static memory allocation only by software approaches assuming that programmers have knowledge of the write intensive variables. Because the *malloc* interface in the Glibc is designed with deliberation of performance, we try to implement *hmalloc*(hybrid memory allocations) with low overhead.

0.4 Design and Implementations

0.4.1 Modifications in Glibc

The Glibc maintains a list of arenas, each managing its own allocated memory spaces. Each memory space is represented as a chunk in Glibc. Many chunks with similar sizes are usually grouped together in a linked list as a bin. Each arena maintains 128 bins considering the performance under different requesting sizes. If a requested size is greater than the maximum size the largest bin can maintain, the *mmap* system call is invoked and the *munmap* is called when the requested space is freed. Obviously, small requested spaces are cached by bins of an arena. When a thread tries to request a memory space, it remembers the pointer of the arena as a thread local variable and will try to use it again in the next request. To distinguish PCM requests and DRAM requests in *hmalloc(size, DRAM or PCM)*, we create another list(harena_list in Figure 3) of arenas which is responsible for caching PCM requests and make the existing list(arena_list in Figure 3) responsible for DRAM requests. Of course, each thread will have 2 thread local variables. One is the pointer to the arena responsible for DRAM requests and the other is the pointer to the harena(the pointer of its arena for PCM allocations). Arenas in the arena_list are not modified so they still call *sbrk* or *mmap*

to extend the virtual memory spaces. But arenas in the harena_list can only call the *mmap* system call. There is only one parameter difference between the *mmap* called by arenas and harenas to differentiate a request between for DRAM and PCM.

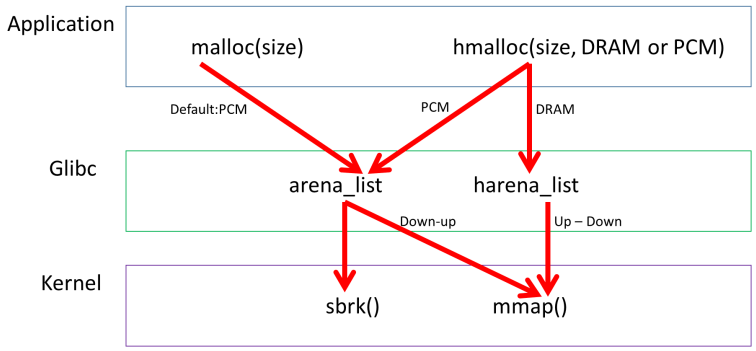


Figure 3 New memory allocation interfaces path in Glibc

0.4.2 New design of process layout and handling page faults

A page fault is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in the physical memory. When a page fault occurs, the information we know in the kernel is limited since it is raised by the hardware and it is hard to pass extra parameters to the *do_page_fault* function. The only useful information we can obtain is the virtual address where the page fault occurs and it is stored in the *CR2* register. Hence, we propose to use different virtual address space for DRAM area and PCM area. We make it possible to decide DRAM or PCM frame we should allocate for a page fault only by the page fault virtual address. This idea drives us to modify the default process layout as shown in Figure 4. Basically, we have enough virtual memory space as large as 256 terabytes which is much larger than the physical memory a machine can equipped with. So we divide the whole virtual space into two parts. The cut-off value is 0x558862ef0000

and it is the median of the beginning values of two “Memory Mapping Region”s. This value is trivial since each “Memory Mapping Region” has a terabyte level space, we assume each region is enough for containing all DRAM requests or PCM requests without striding the median value. The upper part includes the kernel space and its range is from cut-off value to 256 terabytes. The lower part has a range from 0 to cut-off value. The default “Memory Mapping Region” lies in the lower part and raises from bottom up. We create another “Memory Mapping Region” which lies in the upper part and descends from top to bottom leaving enough virtual memory space for the “Stack” area. The function *mmap* is called by arenas use the top-down“Memory Mapping Region” and those called by harenas use the bottom-up “Memory Mapping Region” region. Because events in the kernel space are critical to the system performance, we allocate DRAM frames when page faults occur in the upper part and PCM frames for the lower part. By allocating diffrent frames for the two “Memory Mapping Region”s and making the Glibc library calls different *mmap* functions, we implement a user-controlled interface(textithmalloc) for dynamic memory allocations. But as we look in Figure 4, those segments lie in the lower part such as “Text Segment”, “BSS Segment” and “Data segment”. Allocating PCM frames to the “Text Segment” segment is not critical for the performance since only read operations occur in the “Text Segment”. But write operations possibly occur in the ’BSS Segment’ and the ’Data segment’. When the size of these segments become larger and get more write operations, the performance degradation from these segments can become non-negligible. The GNU compiler provides a feature that we can use *__attribute__(section('segment name'))* to embellish global variables. To make use of this feature, we define another segment name called ’DRAM AREA’ and place write intensive variables in this segment. In the kernel, when a page fault occurs, first we check if the address is in the ’DRAM AREA’ segment; if yes we allocate DRAM frames; if not we check if the address is in the upper or lower part. The function which loads ELF files is the *load_elf_binary* in

the kernel. When an ELF file is loaded, we remember the range of "DRAM_AREA" of the file and make the *mm_struct* of the process remember the range. When a page fault occurs, we can get the "DRAM_AREA" value from the *mm_struct* of current process and use it to check where the page fault address belong to.

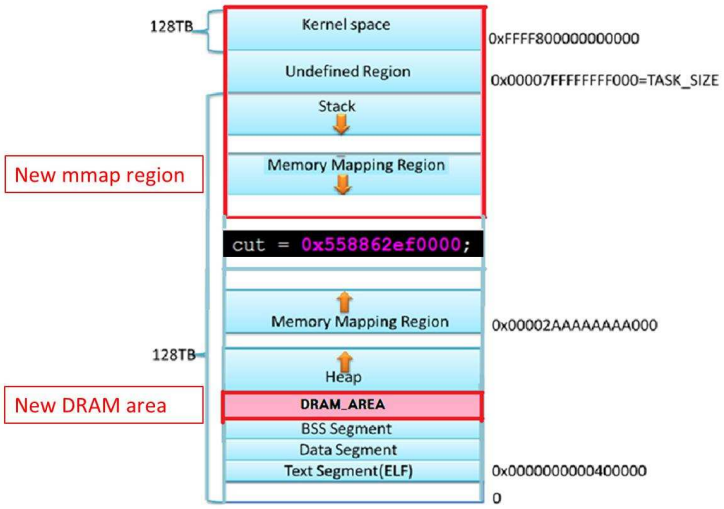


Figure 4 New process layout in 64bit systems

0.5 Evaluation

0.5.1 Experimental environment

We used simulation based evaluations to evaluate the performance of our new memory allocation management scheme. We used MARSS-x86 cycle accurate simulator for system simulation[Patel et al]. We also used Dramsim2 to simulate the main memory system in the MARSS-x86[Rosenfeld et al]. By using these two simulators, we could get cycle accurate physical memory traces, then replay these traces in another self-implemented simulator. We configure the memory latency as short as possible in the Dramsim2 and simulate actual latency of DRAM and PCM in the trace simula-

tion. We filter out memory references that are accessed directly from the L1 and L2 cache memories and gather only the memory references observed at the main memory system. We get diffrent trace distributions between DRAM and PCM by using the *malloc* and *hmalloc*. Then we replay the two diffrent traces in the trace simulation to get the runtime and energy consumption results. Figure 5 shows the system settings and DRAM/PCM delay configurations. We referred [Lee et al] and [Ramos et al] for the experimental enviroment.

System settings	Value
CPU	Single core, 2.7Ghz
L1 cache size	128k
L2 cache size	2M
Cache policy	write back
DRAM size	2GB
PCM size	2GB
page size	4k

Memory Devices	PCM	DRAM
tRCD delay	37ns	10ns
tRP delay	100ns	10ns
tRRDact delay	3ns	4ns
tRRDpre delay	18ns	4ns
Refresh time	n/a	64ns
tRFC / tREFI	n/a	74ns / 3.906μs
Rank size	2048MB	512MB
Bank size	256MB	64MB

Current	PCM	DRAM
Row Buffer Read	220mA	220mA
Row Buffer Write	220mA	220mA
Avg Array R/W	242mA	110mA
Active Standby	62mA	62mA
Precharge Powerdown	40mA	40mA
Refresh	n/a	240mA

Figure 5 System settings and delays of DRAM/PCM

0.5.2 Quick Sort and Merge Sort performance

We evaluate two sorting algorithms as in Figure 6 and Figure 7 using benchmarks in [Quicksort] and [mergesort]. We modified the memory allocating function call from *malloc* to *hmalloc* by which the space for storing the sorted arrays are allocated. We also modified the buffer allocation function to the *hmalloc* in the mergesort benchmark code. X axis represents the number of elements in the sorting algorithms and y axis represents the runtime or energy. The Figure 6(c) and Figure 7(c) show the statistics of R/W operations occur in DRAM(zone0) and PCM(zone1). We compared our default kernel&Glibc and customized kernel&Glibc. We only modified memory allocation interfaces from *malloc* to *hmalloc* in the application code. As results show, in our customized kernel&Glibc the runtime of quicksort reduces by 12-29% along with energy reduction by 33-42% compared with the default kernel&Glibc. The runtime of the mergesort reduces by 16-35% along with energy reduction by 34-54%. PCM has a much much slower write latency than DRAM, so we can get more performance improvement from the mergesort since the mergesort has a higher ratio of write operations than the quicksort.

0.5.3 The lbm performance

We evaluated the "lbm" workload in the SPEC benchmarks as an example of write intensive workloads and the result is shown in Figure 8. We modified memory allocation interfaces from *malloc* to *hmalloc* where the space of cells were allocated. This workload takes too long time to get the whole trace so we limit the read/write operations to 2 million. The X axis represents the cell size of the "lbm" workload. The write ratio in this workload is relatively stable and we get runtime reduction of 40-45% along with energy reduction by 66-70% compared with the default kernel&Glibc in our customized kernel&Glibc.

0.5.4 Data structure overhead of glibc and kernels

We compare the performances of *hmalloc* and *malloc* in Figure 9. We measure the *malloc* performance from a default Glibc2.12 and *hmalloc* in our own modified Glibc2.12. We designed four kinds of tests as below.

(1) 'small size': allocate 1e6 times of small range: [1, 512] bytes, then free it all as Figure 9(a).

(2) 'large size': allocate 1e6 times of memory of large sizes, range:[1, 524287] bytes, then free it all as Figure 9(b).

(3) 'hybrid-non-free loop': allocate 1e4 times of memory of different sizes, range:[1, 524287] bytes, free all of it, repeat this loop 10 times, then free all spaces finally as Figure 9(c).

(4) 'hybrid-half-free loop': allocate 1e4 times of memory of different sizes, range:[1, 524287] bytes, free half of it (e.g. the even allocations), repeat this loop 10 times, then free all spaces finally as Figure 9(d).

As we know, we keep the original *malloc* path, add another list of arenas and only modify one parameter of the *mmap* function. The results shows that the *hmalloc* performs even better than the *malloc* in many cases('-' means runtime reduction of *hmalloc* compared with *malloc*) so we implement the *hmalloc* with low overhead.

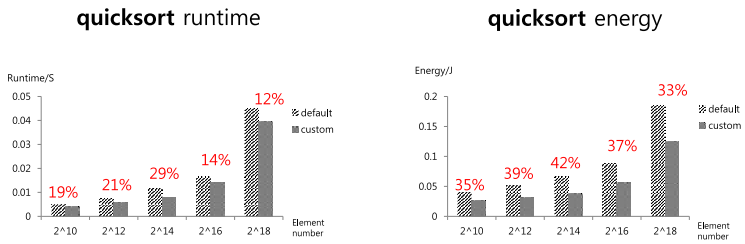
0.6 Limitations

Our approach makes it possible to place variables in dynamically allocated area and process segments into separate DRAM and PCM areas. However, it is still not possible to handle variables in the stack on hybrid memory systems. One possible approach named distributed stack involves compiler modification and have to solve the

virtual memory space layer problem since physical main memories are not visible to CPUs on modern computers unlike embedded systems [Avissar et al]. Today's modern computers even mobile devices have a level of gigabytes of the main memories but the default stack size of programs in the Linux kernels is only 8 megabytes. Moreover, the last level cache size has reached megabyte level so variable accesses in the stack will hit the cache especially those iterators in loops so these variables in the stack are not critical to the performance of processes. So we remain it as a future work when necessary.

0.7 Conclusions and Future Work

We have proposed static memory allocation methodologies where no page migrations will happen assuming programmers know which part of variables are write intensive. Dominating memory consuming variables are made up of two kinds, one is the global data whose memory ranges are fixed after compilation, another is dynamicly allocated variables whose values are determined during the runtime of programs. We make use of the *__attribute__* feature with some kernel modification to make global variables' addresses optional between PCM and DRAM main memories. For dynamicly allocated memory variables, we implement a hybrid memory allocation interface called *hmalloc* by kernel and Glibc modifications. Compared with the interface *malloc*, *hmalloc* needs not only the requesting memory size but also the flag indicating PCM or DRAM, where the allocated memory space should be located in. Because the process layout in 64bit systems allow us to use a huge virtual memory space, by segmenting the process layout carefully, we can define the mapping rules according to the ranges of each area in the layout. We implemented a new memory allocation interface in PCM+DRAM hybrid memory systems with low overhead only by software approaches.



(a) (b)

element: 2^10	total count - default	total count - custom
zone0 read	7655	33054
zone0 write	598	2000
zone1 read	48656	23459
zone1 write	2434	770

element: 2^12	total count - default	total count - custom
zone0 read	8657	40619
zone0 write	840	7274
zone1 read	55162	23614
zone1 write	7026	873

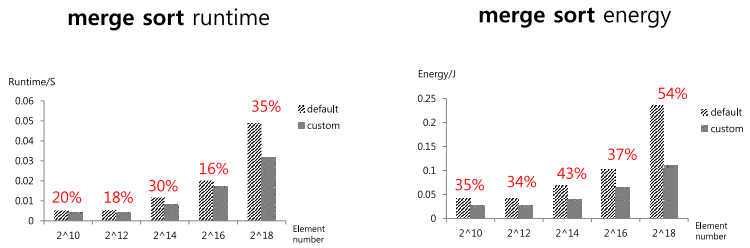
element: 2^14	total count - default	total count - custom
zone0 read	9717	47500
zone0 write	803	10191
zone1 read	60623	24016
zone1 write	9987	842

element: 2^16	total count - default	total count - custom
zone0 read	10244	59875
zone0 write	1126	19709
zone1 read	71004	24418
zone1 write	18169	848

element: 2^18	total count - default	total count - custom
zone0 read	11687	82611
zone0 write	2022	42763
zone1 read	93692	24730
zone1 write	39656	849

(c)

Figure 6 Runtime, Energy and R/W statistics evaluation of quicksort



(a) (b)

element: 2^10	total count - default	total count - custom
zone0 read	8347	33189
zone0 write	681	1984
zone1 read	48798	23520
zone1 write	2205	880

element: 2^12	total count - default	total count - custom
zone0 read	8191	34605
zone0 write	710	2354
zone1 read	48791	23649
zone1 write	2308	791

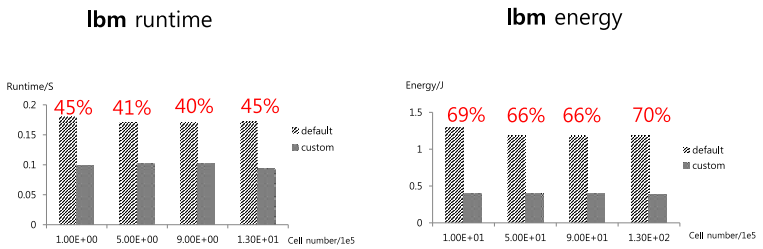
element: 2^14	total count - default	total count - custom
zone0 read	9474	49227
zone0 write	772	11278
zone1 read	61152	24116
zone1 write	11255	973

element: 2^16	total count - default	total count - custom
zone0 read	10989	68568
zone0 write	1418	28308
zone1 read	74679	24421
zone1 write	21130	755

element: 2^18	total count - default	total count - custom
zone0 read	13574	120093
zone0 write	2815	70171
zone1 read	142221	24791
zone1 write	67767	853

(c)

Figure 7 Runtime, Energy and R/W statistics evaluation mergesort



(a) (b)

Cell number: 1e5	total count - default	total count - custom
zone0 read	20351	3074216
zone0 write	12487	1722314
zone1 read	2716207	14842
zone1 write	1505304	942

Cell number: 5e5	total count - default	total count - custom
zone0 read	38368	2202775
zone0 write	31238	2146946
zone1 read	2121853	12368
zone1 write	2065232	795

Cell number: 9e5	total count - default	total count - custom
zone0 read	24293	1917920
zone0 write	17896	1862727
zone1 read	1196122	12388
zone1 write	1139892	802

Cell number: 13e5	total count - default	total count - custom
zone0 read	115522	7129387
zone0 write	103023	7067711
zone1 read	7158842	23887
zone1 write	7051565	791

(c)

Figure 8 Runtime, Energy and R/W statistics evaluation of lbm

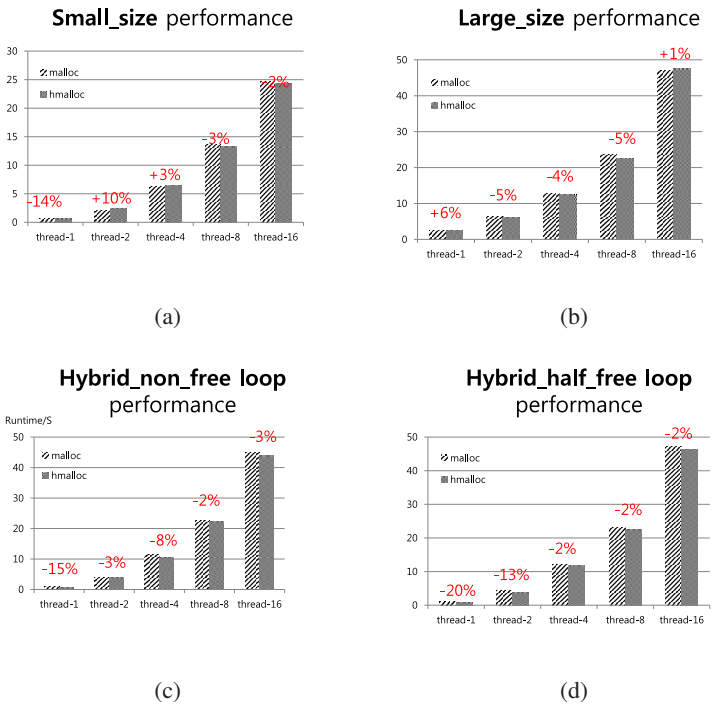


Figure 9 Glibc and kernel overhead

Bibliography

[Lee et al] Lee, Benjamin C and Ipek, Engin and Mutlu, Onur and Burger, Doug. Architecting phase change memory as a scalable dram alternative. In ACM SIGARCH Computer Architecture News, 2009.

[matrix] http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[Quicksort] Stanford. Quicksort benchmark. https://chromium.googlesource.com/native_client/pnacl-llvm-testsuite/+/master-backup/SingleSource/Benchmarks/Stanford/Quicksort.c.

[mergesort] http://groups.csail.mit.edu/cag/raw/benchmark/suites/mergesort/.

[Rosenfeld et al] Rosenfeld, Paul and Cooper-Balis, Elliott and Jacob, Bruce. Dramsim2: A cycle accurate memory system simulator. In IEEE Computer Architecture Letters Computer Architecture News, 2011.

[Patel et al] Patel, Avadh and Afram, Furat and Ghose, Kanad. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In 1st International Qemu Users Forum, 2011.

[Qureshi et al] Qureshi, Moinuddin K and Srinivasan, Vijayalakshmi and Rivers, Jude A. Scalable high performance main memory system using phase-change memory technology. In ACM SIGARCH Computer Architecture News, 2009.

[Mogul et al] Mogul, Jeffrey C and Argollo, Eduardo and Shah, Mehul A and Faraboschi, Paolo. Operating System Support for NVM+ DRAM Hybrid Main Memory. In HotOS, 2009.

[Avissar et al] Avissar, Oren and Barua, Rajeev and Stewart, Dave. Heterogeneous memory management for embedded systems. In ACM Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems, 2001.

[Zhang et al] Zhang, Xiao and Dwarkadas, Sandhya and Shen, Kai. Towards practical page coloring-based multicore cache management. In ACM Proceedings of the 4th ACM European conference on Computer systems, 2001.

[Luk et al] Luk, Chi-Keung and Cohn, Robert and Muth, Robert and Patil, Harish and Klauser, Artur and Lowney, Geoff and Wallace, Steven and Reddi, Vijay Janapa and Hazelwood, Kim. Pin: building customized program analysis tools with dynamic instrumentation. In ACM Sigplan Notices, 2005.

[Wang et al] Wang, Chao and Vazhkudai, Sudharshan S and Ma, Xiaosong and Meng, Fei and Kim, Youngjae and Engelmann, Christian. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, 2012

[CMU2008survey] Carnegie Mellon. Phase Change Memory. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[micron2012mobile] Micron Corporation. Micron Announces Availability of Phase Change Memory for Mobile Devices. http://files.shareholder.com/downloads/ABEA-

45YXOQ/0x0x583454/d3618e67-4160- 4740-92b5-9d47531ec598/MU_News_2012_7_17_Product_News.pdf.

[Zhou et al] Zhou, Yuanyuan and Philbin, James and Li, Kai. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In USENIX Annual Technical Conference, General Track, 2001

[Beloglazov et al] Beloglazov, Anton and Buyya, Rajkumar. Energy efficient resource management in virtualized cloud data centers. In IEEE Computer Society, 2010.

[Ramos et al] Ramos, Luiz E and Gorbatov, Eugene and Bianchini, Ricardo. Page placement in hybrid memory systems. In Proceedings of the international conference on Supercomputing, 2011.

[Kim et al] Kim, Hyojun and Seshadri, Sangeetha and Dickey, Clement L and Chiu, Lawrence. Evaluating phase change memory for enterprise storage systems: a study of caching and tiering approaches. In Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14), 2014.

[Lee et al] Lee, Soyoon and Bahn, Hyokyung and Noh, S. CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures. In IEEE Computer Society, 2013.

요약

Phase-Change Memory (PCM) 은 차세대 메모리로 주목을 받고 있다. PCM은 DRAM과 비교해서 높은 밀도를 가지고 있고 비휘발성이므로 idle상태에서 에너지가 크지 않다. 하지만, PCM 은 짧은 소자수명과 느린 쓰기속도가 문제가 된다. 이를 해결하기 위하여 현재는 PCM과 DRAM을 혼용하는 하이브리드 메모리 시스템에 대한 연구가 진행되고 있다. 최근의 연구로는 리눅스 커널에서 page 단위의 DRAM과 PCM사이에서 이동을 시키는 기법들을 진행해왔다. 그러나 DRAM과 PCM사이에서 page들을 이동을 시키려면 각각의 page에 대해서 읽기/쓰기가 발생한 통계가 필요하며 이런 통계를 위해서는 또 다른 자료구조들을 만들어야 된다. 하드웨어 기반의 접근법은 성능이 좋게 나오나 배포하는데 어려움이 있고 소프트웨어기반의 접근법은 익명 페이지에 대해서 통계 계산의 어려움이 있다. 본 논문에서는 기존의 메모리할당 인터페이스를 확장하여, 변수들을 DRAM 또는 PCM에 구분하여 할당하는 방식을 연구하였다.

주요어: 서울대학교, 전기.컴퓨터공학부, 졸업논문
학번: 2012-23962