MN-MATE: Elastic Resource Management of Manycores and a Hybrid Memory Hierarchy for a Cloud Node

KYU HO PARK, WOOMIN HWANG, HYUNCHUL SEOK, CHULMIN KIM,
DONG-JAE SHIN, DONG JIN KIM, MIN KYU MAENG, and SEONG MIN KIM,
Computer Engineering Research Laboratory, KAIST

Recent advent of manycore system increases needs for larger but faster memory hierarchy. Emerging next generation memories such as on-chip DRAM and nonvolatile memory (NVRAM) are promising candidates for replacement of DRAM-only main memory. Combined with the manycore trends, it gives an opportunity to rethink conventional resource management system with a memory hierarchy for a single cloud node. In an attempt to mitigate the energy and memory problems, we propose MN-MATE, an elastic resource management architecture for a single cloud node with manycores, on-chip DRAM, and large size of off-chip DRAM and NVRAM. In MN-MATE, the hypervisor places consolidated VMs and balances memory among them. Based on the monitored information about the allocated memory, a guest OS co-schedules tasks accessing different types of memory with complementary access intensity. Polymorphic management of DRAM hierarchy accelerates average memory access speed inside each guest OS. A guest OS reduces energy consumption with small performance loss based on the NVRAM-aware data placement policy and the hybrid page cache. A new lightweight kernel is developed to reduce the overhead from the guest OS for scientific applications. Experiment results show that our techniques in MN-MATE platform improve system performance and reduce energy consumption.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Performance

Additional Key Words and Phrases: Virtual machine, resource management, scheduling, NVRAM, hybrid main memory

ACM Reference Format:
Kyu Ho Park, Woomin Hwang, Hyunchul Seok, Chulmin Kim, Dong-Jae Shin, Dong Jin Kim, Min Kyu Maeng, and Seong Min Kim. 2015. MN-MATE: Elastic resource management of manycores and a hybrid memory hierarchy for a cloud node. ACM J. Emerg. Technol. Comput. Syst. 12, 1, Article 5 (July 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2701429

1. INTRODUCTION

Recent advent of manycore system in a computing node enables increased concurrency of executions by accommodating multiple guest OSes and multiple tasks inside each guest. More concurrent tasks requiring larger size of memory creates needs for larger and faster memory hierarchy. However, growing disparity of speed between CPU and off-CPU memory, called memory wall, results in decrease of benefit from concurrent execution of tasks. Limited communication bandwidth between CPU and memory also increase the disparity. Furthermore, large size of DRAM consumes much larger energy.

This work was supported by the Ministry of Knowledge Economy, South Korea, under Project No. 10035231 and the ICT R&D program of MSIP/IITP [10038768, The Development of Supercomputing System for the Genome Analysis].
W. Hwang is now with the Affiliated Institute of ETRI, Daejeon, South Korea.
Authors' addresses: KAIST, Daejeon, South Korea; email: kpark@kaist.ac.kr, wmhwang@core.kaist.ac.kr.

DOI: http://dx.doi.org/10.1145/2701429

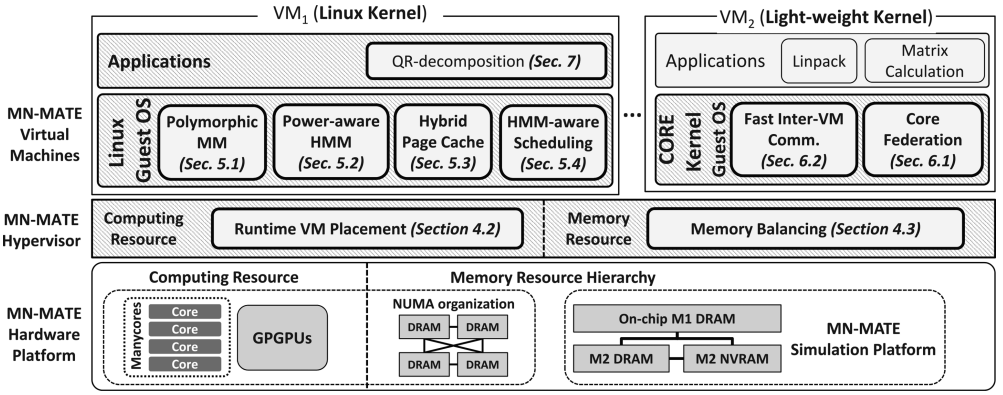


Fig. 1. Overall MN-MATE system hierarchy. HMM indicates a hybrid architecture of the main memory.

Several researches try to mitigate overheads from off-chip DRAM-only main memory system. Putting DRAM inside a CPU chip can reduce access time to the main memory [Kgil et al. 2006; Liu et al. 2005; Loi et al. 2006]. Utilizing NVRAM main memory could reduce the energy consumption [Quershi et al. 2009; Zhao et al. 2007; Lee et al. 2009]. Highly parallelizable computing units like GPGPU accelerate computation speed.

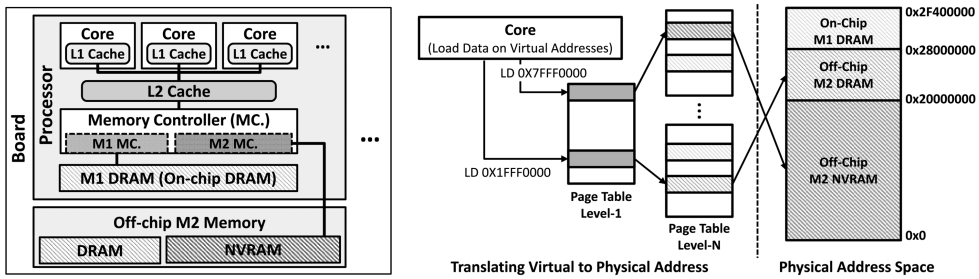
Utilizing resources is another crucial issue when the hardware includes on-chip DRAM, off-chip NVRAM main memory, and GPGPUs for energy-efficiency and high performance. Larger size of memory in a computing node creates different access latencies from memory hierarchy to each core. In such an environment, partitioning and balancing resources among VMs and execution entities is a challenging issue to provide fairness while enhancing performance. NVRAM main memory creates needs for the hypervisor and guest OS to decide how it is distributed to them. Managing hybrid architecture of NVRAM and DRAM and data placing policy on them also should be considered to reduce energy consumption while maintaining application performance.

In an attempt to manage resources, we propose MN-MATE, a novel resource management system which balances resource allocations among VMs and manages them inside each VM. Figure 1 illustrates an overview of our MN-MATE system. In the hypervisor, we propose a VM placement policy considering NUMA (Non-Uniform Memory Access) architecture of target memory hierarchy. The hypervisor also instantly balances memory usage among VMs according to their changing memory demands.

On top of the various types of memory resources, we propose three management schemes for distributed memory hierarchy. Allocated DRAM memories in the hierarchy are managed polymorphically based on memory/cache-oriented solutions. Hot/cold separation and data migration scheme based on page grouping for hybrid main memory of DRAM and NVRAM reduces energy consumption dramatically. Utilizing NVRAM as a base of page caching also reduces energy consumption. Experiment results show that our resource management schemes shows remarkable performance improvements while reducing system-wide energy consumption.

This study is an extension of our previous work [Park et al. 2010, 2012a, 2012b], in which we focused on a single layer of the VM environment, either a guest OS or the hypervisor. Our objective in this study, however, is to integrate techniques for each guest OS and the hypervisor running on top of a single cloud node.

The remainder of this article is organized as follows. Section 2 illustrates our target architecture for MN-MATE as a cloud node. Section 3 describes problems to solve in MN-MATE target system. Section 4 describes VM management techniques in the hypervisor. Sections 5 and 6 describes management solutions inside each of



(a) Physical topology of hierarchical memories

(b) Memory map of hierarchical memories. Part of M1 and M2 memory work as main memory

Fig. 2. Target hardware architecture of the MN-MATE system.

resource-balanced VM, especially with two types of guest OSes. Section 7 describes two applications run on top of our system. Section 8 shows experiment results of our designs. Section 9 discusses related work. We conclude our work in Section 10.

2. TARGET ARCHITECTURE OF MN-MATE

Technical limitations of increasing CPU clock speed make a CPU chip integrate more cores. Manycore CPU increases computational capability of a system by parallelizing instruction executions. Larger size of main memory is required to accommodate more concurrent tasks with bigger memory footprints. However, limited DRAM scalability [Lee et al. 2009; Meza et al. 2012] and various access latencies according to the location of the main memory [Govil et al. 1999; Bligh et al. 2004; VMware 2005] have been pointed out as a main reason of performance degradation. Low energy efficiency of the DRAM main memory also emerges a big issue [Park et al. 2011b]. In addition, conventional cores are not specialized for parallel executions due to their general purposes.

There are several architectural propositions for memory hierarchy to utilize all computational abilities of the manycore CPUs and GPUs. Recent advances in 3D stacking semiconductor technologies give an opportunity of faster main memory access to cores by stacking some part of main memory on top of cores in a chip. It gives higher bandwidth and low access latencies of on-chip main memory to the cores, while most of the main memory out side the chip play roles of conventional main memory. Next generation NVRAMs (NGNVRAM) such as Phase-change RAM (PRAM), Ferroelectric RAM (FRAM), and Magnetoresistive RAM (MRAM) enlarge main memory capacity based on their higher density. They also become promising main memory candidate because of their low energy consumption from nonvolatility characteristic. They have unique advantages compared with DRAM, long access latency and asymmetric read/write latency make it hard to solely used as main memory of high performance computing system. It gives us an insight that advantages of each medium can complement the other. Hybrid combination of NGNVRAM and DRAM, therefore, can be considered as promising future main memory.

In this article, we target a single hardware node for a cloud computing system. Various resources can be integrated into a state-of-the-art single cloud node to enhance system performance. Among them, we target a combined architecture of manycore and hierarchical hybrid main memory. Figure 2(a) shows our target hardware architecture illustrating several architectural propositions for manycore system. There are several manycore processors containing cores, caches, memory controllers, and on-chip DRAM. The main memory hierarchy is composed of on-chip DRAM, conventional off-chip DRAM, and emerging off-chip NVRAM, where each of them are available via

memory controllers. We build two layers of memory hierarchy, M1 memory with on-chip DRAM and M2 memory with off-chip DRAM and NVRAM. M1 memory located in the CPU chip provides higher bandwidth and lower access latencies. Because M2 memory provides larger memory capacity with small performance loss, each type of memory is complementary to the other. In M2 memory, DRAM and NVRAM are located at the same memory level, like shown in Figure 2(b). Here we use PRAM and STTRAM as representatives of NVRAM for its scalability and low energy consumption.

3. MOTIVATIONS

Resource management is a consecutive repetition of resource partitioning and balancing to meet requirements. Under the virtualized system, the hypervisor distributes CPU timeslices and memory to each guest VMs. Guest OS manages received resources and run applications on top of them.

The hypervisor have to consider NUMA architecture to distribute CPU timeslices among guest VMs while maintaining fairness among them. Large size of main memory is composed of multiple memory modules, which causes different access latencies. Furthermore, our target system has hierarchical memory architecture which has various kind of access latencies, including NVRAM. In such a situation, there are four considerations. First, a VM may have access its memory remotely when its VCPU and memory are not in the same NUMA node (*rem_acc*). Second, a VM may suffer from contentions on shared resources which its memory access requests pass through. Contention on a shared cache gives higher cache miss ratio to the VMs using it (*l*3*_cont*). Contention on a memory controller and an interconnection gives delayed memory access latency to the VMs using them (*mc_cont* and *ic_cont*, respectively). Depending on the VM placement decision of the hypervisor and the memory access loads of the VMs, each VM will have unpredictable data access performance.

Partitioning and balancing of memory among consolidated VMs are critical for enhancing system-wide memory utilization on a single hardware. Conventionally, memory inertia and the higher penalty from wrong decision make the memory partitioning and balancing more important than scheduling CPU cores. For this, the hypervisor needs to figure out which VM requires memory and which VM has the least useful memory. Then, the hypervisor has to decide the memory type, amount of memory to transfer, page frames to transfer. With previous *ballooning* [Waldspurger 2002] solutions, selection of page frames to transfer is responsible for the owner guest OS. It delays securing free memory in the guest OS due to the reclamation of dirty pages. Such delay depreciates the value of the transferred memory in the recipient VM.

Once the hypervisor distributes resources, each guest OS runs applications on top of them. Each guest OS receives different memories with various memory access latency while receiving timeslices of homogeneous CPU cores. It gives them problems to solve.

Guest OS scheduler have to consider different memory access latency like the hypervisor does to place VMs. In MN-MATE, a guest OS can have three type of main memory, M1 DRAM, M2 DRAM, and M2 NVRAM. Along with the different DRAM access latency, asymmetric read and write latency of NVRAM makes contention in the memory controller more severe. Latency to access NVRAM significantly increases when a task's memory is located in a different domain on a NUMA system. Zhuravlev et al. [2010] and Blagodurov et al. [2010] proposed approaches to utilize memory contention to the task scheduling. But they have no considerations about different NVRAM read and write latencies in their scheduling method. The number of memory transactions they used as criteria does not reflect the amount of memory bandwidth consumption.

In addition to the problems in task scheduling, a guest OS has to choose a method to carefully manage M1 memory, the fastest main memory medium for its higher bandwidth and lower latency. To enlarge SRAM caches, it can be used as an another

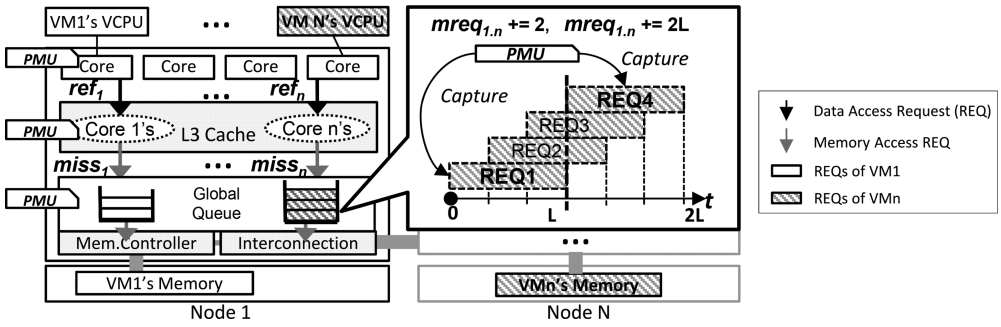


Fig. 3. Detailed explanation of PMU event monitor.

level of LLC (last level cache). Tasks can be accelerated as they get more M1 memory. To enhance system-wide performance, the guest OS need to decide how much memory are managed by each method.

Each guest OS must manage DRAM and NVRAM utilization of M2 memory not to degrade performance while reducing energy consumption. Read from M2 NVRAM is energy-efficient because of the nonvolatility of the medium. However, long write latency and the compensation operations for the limited write endurance degrade performance in exchange for the benefit. Higher write energy consumption may countervail saved energy by the read operations. Because DRAM has faster and same read/write latency with higher energy consumption, the guest OS utilizes both types of memory with the consideration of trade-off between performance and energy consumption.

The management of M2 NVRAM greatly affect system performance if M2 NVRAM page frames are used as page cache. The primary purpose of the page cache is to hide disk access latency. However, conventional algorithms for page cache, such as LRU [Dan and Towsley 1990], LIRS [Jiang and Zhang 2002], and CLOCK-Pro [Jiang et al. 2005], are often suboptimal. Their DRAM-oriented design including symmetric access latency and unlimited endurance make the page cache inefficient, thus make previous algorithms not applicable to the M2 NVRAM-based page cache.

4. DESIGN FOR HYPERVISOR

In the hypervisor, we build a system monitoring module to track current status of running entities. Based on the collected information, we propose a policy for VM placements and balances memory among them.

4.1. Monitoring of System Status

In MN-MATE, several problem-solving approaches use recent status of the system as decision criteria. The hypervisor collects events and performance information of guest VMs and their tasks. For VM relocation, as shown in Table I and Figure 3, it especially utilizes hardware PMUs (Performance Monitoring Units) to count the events occurred for a second in several parts of the MN-MATE hardware platform. In the core level, the hypervisor collect the number of instructions consumed by each VCPU, $inst_i$ in the table. In the cache level, it counts how many times each VCPU attempted to access data in the cache, ref_i, and failed to obtain the data, $miss_i$. In the memory controller and the interconnection, it measures average latency of requests accessing memory which missed the shared cache. For the purpose, the PMU captures a memory access request at a time (e.g., *REQ1* and *REQ4* in the figure) and counts the number of cycles consumed for each captured request. The total number of the cycles divided by the number of captured requests, $mlat_{S,D}/mreq_{S,D}$, become the average latency of the

Table I. Monitored PMU Events for Runtime VM Placement

Event Count	Event (Evt.) Description (Evt. Symbol in AMD 6172 Processor [AMD 2013])	Monitoring Object
ref_i	The number of L3 Cache References, Data Access Requests (NBPMCx4E0)	VCPU of VM_i
$miss_i$	The number of L3 Cache Misses, Memory Access Requests (NBPMCx4E1)	VCPU of VM_i
$inst_i$	The number of retired instructions (PMCx0C0)	VCPU of VM_i
$mreq_{S.D}$	The number of DRAM Read requests captured, one at one time (NBPMCx1E3)	$NodePair_{S.D}$ (Src. S, Dst. D)
$mlat_{S.D}$	Cycles taken for the requests captured for $mreq_{S.D}$ (NBPMCx1E2)	$NodePair_{S.D}$

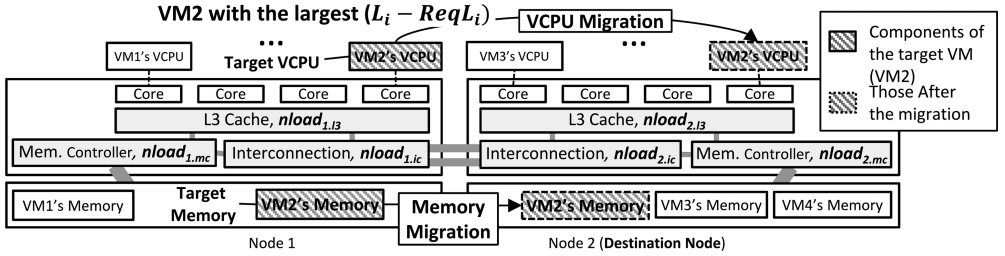


Fig. 4. Runtime migration of target VM on MN-MATE.

memory access requests. For memory balancing among VMs, the hypervisor monitors swap storage usage of each guest OS.

4.2. Runtime Placement of guest VMs

To place guest VMs with the consideration of different memory access latencies, the hypervisor needs several performance information. As the PMUs cannot monitor the value directly, we build a heuristic equation and fill the terms using the counts of the PMUs described in Table I. The equation is expressed as

$$L_i(t) = (1 - cmr_i(t)) \cdot t_{L3} + cmr_i(t) \cdot \left(\frac{t_{i.mem}(t)}{conc_i(t)}\right) = \left(1 - \frac{miss_i(t)}{ref_i(t)}\right) \cdot t_{L3} + \frac{miss_i(t)}{ref_i(t)} \cdot \frac{\left(\frac{mlat_{S.D}(t)}{mreq_{S.D}(t)}\right)}{\left(\frac{miss_i(t)}{mreq_{S.D}}(t)\right)} \quad (1)$$

$$= \left(1 - \frac{miss_i(t)}{ref_i(t)}\right) \cdot t_{L3} + \frac{mlat_{S.D}(t)}{ref_i(t)},$$

where L_i is the measured data access latency of VM_i, t_{L3} is the constant latency of VM_i taken for accessing a L3 shared cache, cmr_i is the cache miss ratio of VM_i, $t_{i.mem}$ is the memory access latency of VM_i on average, and $conc_i$ is the concurrency of the memory access requests. The other terms are described in Table I. The equation is composed of the terms related with the requests hit the L3 shared cache and the requests missed in the cache. For the latter term, as the modern CPUs can issue multiple outstanding memory access requests, we divide the measured memory access latency by the concurrency of the memory access requests.

The primary goal of the hypervisor changing placement of guest VMs is to minimize the effect of the delay. The whole VM placement procedure consists of two parts.

First half is to find VMs which needs to be migrated. The prerequisite for selecting migration candidate is to declare how much data access delay each VM requires, which is denoted by $ReqL_i$ in Figure 4. Each VM then measures the average data access latency using (1). The hypervisor selects a VM with the largest difference of access delay between the required value and the measured value.

Table II. Summary of *Mcredit*-Based Migration (Mig.) Algorithm

Reason Name	Mig. Object	Mig. Destination Node (*mig*) ($nload_X$ in Figure 4)
l3_cont	VCPU	The node with the least MPKI (cache misses per kilo instructions) sum on its shared cache ($nload_{mig.l3}$) and the idle interconnection ($nload_{mig.ic}$)
rem_acc	VCPU	The node where the memory of VM_i is
ic_cont	VCPU	The node with less *rem_acc*& the idle interconnection ($nload_{mig.ic}$)
mc_cont	Memory, VCPU	The node with the least MPKI sum on the memory controller ($nload_{mig.mc}$)

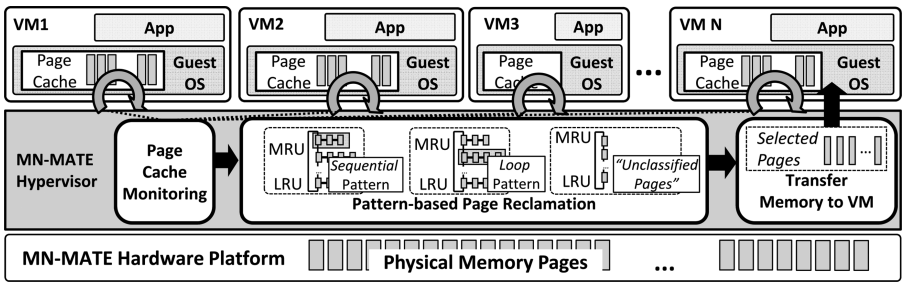


Fig. 5. Overall architecture of instant memory balancing scheme.

The second half is to find out the main reason of the delay and selects proper migration strategy. For each candidate reason, the hypervisor estimates L_i under no effect of the reason by replacing each term in (1) to an ideal value. To exclude *l3_cont*, we replace cmr_i of (1) to the heuristically estimated cache miss ratio as if VM_i owns the shared cache. To exclude *rem_acc*, we subtract the constant remote access delay from the term for the memory access requests. For *ic_cont* and *mc_cont*, we set the term for memory access requests to the ideal memory access latency value. If the hypervisor find that L_i without a certain reason is smaller than the others, the hypervisor tries to migrate the target VM following the policies in Table II.

4.3. Memory Balancing among Consolidated VMs

The hypervisor have to manage M2 memory demands of each guest VM, even after placing them. To adapt changing memory demands of consolidated VMs, the hypervisor balances M2 memory among them. The hypervisor accelerates the balancing procedure based on reference pattern classification. In this subsection, 'memory' indicates the M2 main memory in the MN-MATE memory hierarchy, unless otherwise noted.

The MN-MATE hypervisor selects least-valuable page frames to transfer by estimating reclamation cost of page frames according to the memory type and reference pattern. Then the hypervisor reclaims selected page frames and donates them to memory-thirsty VMs. Figure 5 illustrates an overall memory balancing architecture.

To estimate least-valuable page frames among candidates, the hypervisor uses their reference pattern. In Figure 5, each guest OS passes on information about events on pages within its own page cache. The events include which page is inserted into, evicted from, and reused within the cache. In addition, memory accesses from the VMs to those pages are intercepted by the hypervisor. Combined with the above information, the hypervisor classifies candidate page frames into three pattern categories and manages them to find least-valuable page frames. Here the hypervisor considers clean page frames that belong to a page cache as candidates due to its volatility.

Figure 6 illustrates how the hypervisor monitors memory access patterns. A guest OS informs the hypervisor arguments of read/write-related system calls. We categorize

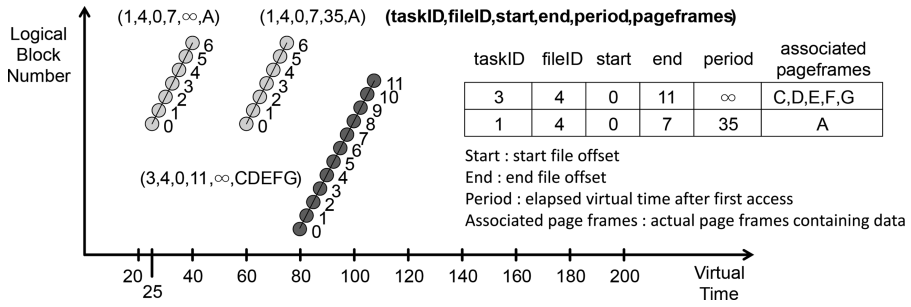


Fig. 6. Examples of sequential, loop reference pattern sequence and a sequence management table.

page frames into three types: sequential, loop, and unclassified references. Page frames are categorized as sequential and grouped as a sequence if a task accesses them sequentially. If a sequence is accessed sequentially again with a period, it is categorized as loop pattern. Otherwise, page frames are categorized as unclassified. We use MRU for sequences and LRU for the unclassified page frames.

The next operation is to decide which VM requires memory. With the collected swap storage usage of each VM, the hypervisor determines memory demands of all the VMs.

The instant memory transfer is a consecutive procedure of page frame reclamation followed by donation to memory-thirst VMs. When a page is requested for memory balance, for example, a least-valuable page frame is selected as the victim page on the basis of its reference pattern. The page frame is transparently reclaimed shortly before the hypervisor tries to schedule the beneficiary VM. Sequentially referenced pages are reclaimed prior to others. Loop referenced pages are next candidates of reclamation for balancing. Depending on the reason for the request, a different allocation mechanism is then applied to the beneficiary VM. If the reason is a paging-in event of a reclaimed page in the hypervisor, a victim page is directly allocated. However, if the request is the result of memory balancing, the hypervisor allocates the page to a guest OS through ballooning. Later, if a victim VM has stolen more memory than the threshold level, the hypervisor requests explicit memory borrowing from the VM.

In our page frame management, we concentrate on clean page frames that belong to a page cache because of the volatility of the clean pages. Generally, guest operating systems, such as Linux, attempt to use any available memory for their own caching purposes. As a result, only a small amount of memory is left free; others contain contents stored in the permanent storage. Because the cached nondirty content can be rebuilt by doing a read operation from its storage location, the guest can tolerate the loss of the page content. Thus, if the hypervisor tries to reclaim those pages, there is no need to swap out page content to its own swap device and dual-swapping can be avoided. Hence, we restrict our monitoring to page frames that are used as a page cache and choose clean pages as victims. This process requires no swap storage area for the hypervisor, no additional management cost, and no data flush overhead.

5. DESIGN FOR LINUX GUEST OS

In this section we describe three memory management techniques to manage various memories in the memory hierarchy distributed by the hypervisor. We propose polymorphic memory for DRAM hierarchy including M1 and M2 DRAMs. For hybrid architecture of M2 NVRAM and M2 DRAM, we proposed two schemes run inside each guest OS. One is a power-aware memory management scheme in the guest OS for user-level memories. The other is a page cache management algorithm to manage a part of kernel memory in the energy-efficient manner.

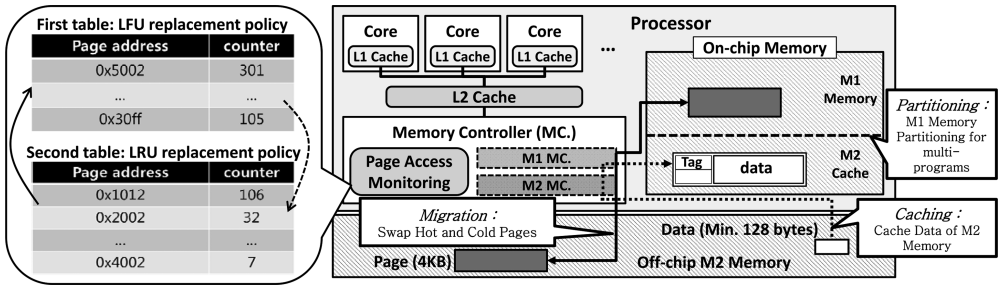


Fig. 7. Polymorphic memory architecture and its operations.

5.1. Polymorphic Memory

Polymorphic memory management scheme is a hybrid approach of the main memory management scheme and the cache management scheme to utilize on-chip M1 DRAM. We divide M1 DRAM into two parts: one is used as a main memory, called *M1 memory*. Another is used as a cache for M2 DRAM, called *M2 cache*. The entire M2 DRAM called 'M2 memory' is used as a main memory with M1 memory. Figure 7 describes the target memory architecture and its operations. Basic operations consist of three parts: 1) Monitoring patterns of page accesses in the memory controller and hot-page migration; 2) Partitioning of M1 DRAM to mitigate contention problems between multiple processes; 3) Utilizing the M2 cache as a new LLC for the data in M2 memory.

The polymorphic memory management is a periodic procedure of hardware-assisted page monitoring followed by OS intervention for migration and partitioning. During a period, the hardware page access monitoring module collects information about page accesses. At the end of each period, the OS decides a new mapping of pages with the collected information. The key assumption of this approach is that the page access pattern during one period is similar to the next pattern during the next period. The OS partitions M1 memory for processes and moves the frequently accessed pages in M2 memory into M1 memory. Then, all processes resume their works.

To monitor the page access patterns, the memory controller maintains two tables that include entries of a page frame address and a counter, as shown in Figure 7. The first table is designed to contain entries of the most frequently accessed pages. It is managed with an LFU-like replacement policy to represent frequency. It always contains pages with larger count values than pages on the second table. The second table use LRU policy to represent recency. When a new page is accessed, its corresponding entry is inserted into the MRU position of the second table with a count value of 1. If the page is accessed, and its corresponding entry is already on the first or second table, the count value of the corresponding entry is increased by 1. When the corresponding entry is on the second table, we compare the count value after increasing by 1. If the count value is greater than or equal to the minimum count value on the first table, the entry of the accessed page is moved to the first table and the entry with the minimum count value is moved to the MRU position of the second table.

The second is M1 memory partitioning to mitigate contention problems between many processes. In the case of multiple programs, all programs will attempt to get a greater portion of M1 memory for their fast execution because M1 memory shows better performance than M2 memory. Because M1 size is limited, it will cause memory contention. Our approach to solve the contention problem is to partition M1 memory by controlling the M1 size for each process. By using variance in M1 hit ratios of each process, we categorize the processes and determine M1 size for each process.

Lastly, we use the M2 cache as a cache between a conventional LLC and M2 memory. Despite the efficiency of the monitoring algorithm, its table size is limited. As a result,

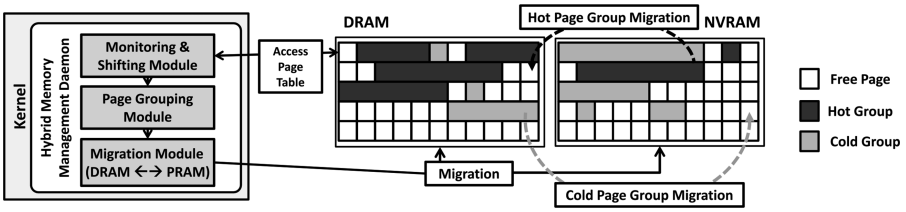


Fig. 8. Overall architecture of hybrid M2 NVRAM and M2 DRAM main memory management system.

misprediction can lowers the hit ratio and the performance. To reduce the performance loss by misprediction, we propose a method to use the M2 cache as a hybrid manner. From our experimental results shown in Figure 17, 16MB of 128MB of M1 (on-chip memory) is proper to the size of M2 cache in order to get the best performance.

5.2. Power-Aware Hybrid Main Memory

To manage main memory composed of M1 DRAM, M2 DRAM, and M2 NVRAM in a energy-efficient manner, we propose a dynamic power-aware page placement strategy by the guest OS for user-level memories. We group hot and cold pages based on our monitoring mechanism and then move them when necessary. In our target architecture, a guest OS can have three types of main memory. As a preliminary design, however, we assume that the target main memory consists of M2 NVRAM and M2 DRAM at the same level of memory hierarchy.

Figure 8 illustrates our power-aware management scheme. Because NVRAM is energy efficient when updated infrequently, our basic strategy is to locate frequently updating pages to DRAM and to locate others to NVRAM for energy efficiency. A kernel daemon performs monitoring, grouping, and migrating pages periodically.

Monitoring the frequency of page changes is important for the guest OS to decide locations of pages under different memory characteristics. Basically, frequently changing pages are better located in DRAM because of the lower latency and small write energy consumption. We introduce a new metric, *hotness*, to measure how frequently each page is changed in a period of time. To differentiate hotness of the pages, we monitor dirty bit in the page table entry (PTE) of all pages. Figure 9 shows how we monitor the hotness of the pages. Utilizing unused 59-62 bits of each PTE, the monitoring module collects recent write history of the page. The 6th bits, denoted with 'D', means dirty bit which is changed to '1' by processor when a write operation occurs. If dirty bit is 1, it shifts to the first location. Similarly, in the case of 0, it shifts to the first location. Hotness is the sum of weight where a bit is set.

Next step is to bind pages to a group to move them together. We use another metric, physical distance, indicating the difference of PFN that each page is actually located. It is applicable because it is based on the characteristics that the Linux memory allocator, Buddy System, keeps memory blocks allocated together contiguous. While existing techniques usually use placement policy based on page-level granularity, we focus on the relation between physically near pages. We found that grouped placement policy can improve the efficiency because these pages are usually accessed at the same time. Every scanning time the module calculates physical difference of adjacent PTEs' page frames in descending order. If the physical difference of a PTE is less than the predefined threshold, the PTE's page frame is grouped to a previous one. A PTE with greater physical difference than the threshold indicates a beginning of a new group.

The final step is to migrate page groups to the appropriate locations. The migration module first decide the hotness of each group, then migrate them based on the decision. The hotness of a group is calculated by the average of all PTEs' hotness. If the average

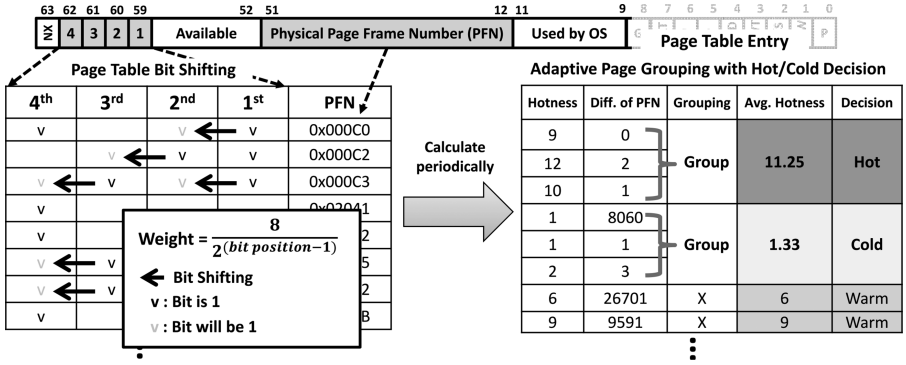


Fig. 9. An example diagram of the adaptive page grouping algorithm. Bit shifting is used for storing dirty bit information. Page Group is based on physical distance because physically near pages have similar access counts.

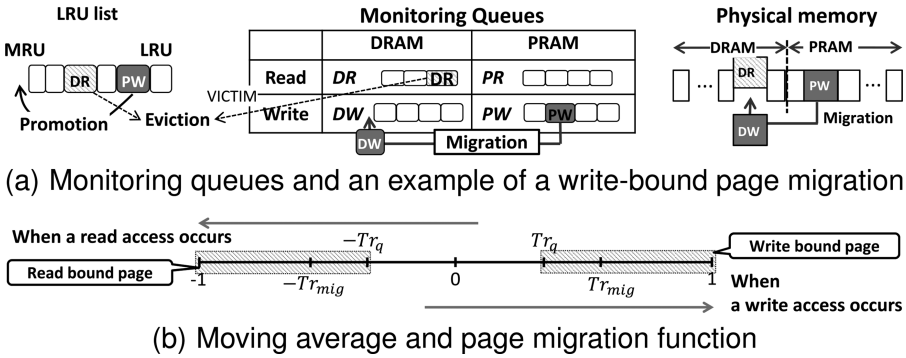


Fig. 10. An overview of the hybrid page cache and its operations.

hotness of a group is greater than the predefined hot_threshold, it become a hot group. If it is smaller than the cold_threshold, it become a cold group. If not included in both cases, it become a warm group.

The migration module migrates a page group if a hot group is located in NVRAM and a cold group is located in DRAM. We used four basic migration principles: 1) Allocate DRAM to every memory allocation request; 2) Move hot groups to DRAM and cold groups to NVRAM; 3) No changed to the warm groups; 4) Allocate DRAM to every kernel memory request.

5.3. Hybrid Page Cache

Hybrid main memory of M2 NVRAM and M2 DRAM also gives each guest OS a need to manage data placements for its own page cache. To reduce energy consumption while maintaining memory access performance, we propose a data placement scheme based on predicted page access pattern, which aims at locating write-bound pages to DRAM.

A guest OS predicts page access patterns with four monitoring queues: a DRAM read queue, a DRAM write queue, a NVRAM read queue, and a NVRAM write queue. Figure 10(a) illustrates basic operations with the four queues. If a page is accessed, it is added to both the LRU list for cache management and one of the four queues according to its access type and the target memory type.

To determine how close a page is to write-bound or read-bound, the guest OS calculates a weight value. It is calculated by a moving average $W_{cur} = \alpha W_{prev} + (1 - \alpha)RT$

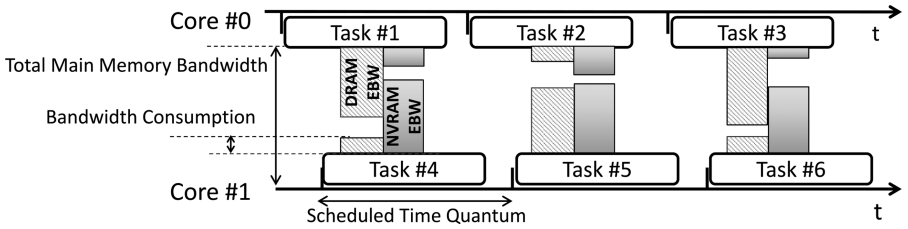


Fig. 11. Basic idea of the task scheduling for hybrid main memory. The scheduler selects a task consuming complementary EBW to different type of memory.

with weight $\alpha \in [0, 1]$, where RT means the requested type of the page. W_{cur} increases when write requests occur while decreases at every read request.

Figure 10(b) shows how the guest OS determines movements between read queue and write queue. We use two threshold values, Tr_{mig} and Tr_q, based on experiments. Tr_{mig} indicates whether a page is required to be migrated. Tr_q decides movement between read queue and write queue.

We have a two-way migration strategy: write-bound page to DRAM and read-bound page to NVRAM. For example, if a write touches a page in the NVRAM write queue and its weight value is greater than Tr_{mig}, the guest OS migrates the page to the DRAM write queue. Similarly, a page in the DRAM read queue can be migrated to NVRAM's if the page's W_{cur} is smaller than $-Tr_{mig}$ when read.

5.4. Task Scheduling for Hybrid Main Memory

Hybrid main memory of M2 NVRAM and M2 DRAM incurs conflicts in memory accesses among tasks using same type of memory. Each guest OS, therefore, requires a new task scheduling policy diffusing memory accesses through time to prevent performance degradation. Our scheduling scheme consists of two phases: 1) per-task memory access monitoring; 2) selection of a next candidate task to be scheduled based on collected memory access statistics during the previous phase. Figure 11 illustrates the basic idea of the hybrid main memory-aware task scheduling.

In the first phase, a guest OS collects per-task memory access counts whenever a task consumes all scheduled time slices or is preempted by other task. Collected values include the number of DRAM accesses, the number of NVRAM reads, and the number of NVRAM writes. Generally, the DRAM bandwidth usage of a task is measured as the number of memory transactions during a unit time. However, NVRAM has different access latencies from DRAM, which an access request takes longer time to be responded. Conventional method has no capability to recognize bandwidth fluctuation of a task according to the distribution ratio of read and write. We therefore calculate a new metric, Effective Memory Bandwidth (EBW) to translate NVRAM's bandwidth usage into the number of DRAM transactions using Equation (2). Here $EBW(T)$ indicates the effective bandwidth of a task T, EBW_{DorNV} indicates effective bandwidth for DRAM or NVRAM. BW_D represents conventional bandwidth for DRAM. NUM_{Req} is the number of memory access transactions. T_D indicates memory access latency of DRAM. N_R and N_W is the number of read/write transactions to the NVRAM. γ and δ denotes a relative access latency of NVRAM read and write compared with the DRAM access, respectively.

$$\begin{aligned} EBW_D &= BW_D \simeq NUM_{Req} \times T_D \\ EBW_{NV} &\simeq \gamma \times NUM_R \times T_D + \delta \times NUM_W \times T_D. \end{aligned} \tag{2}$$

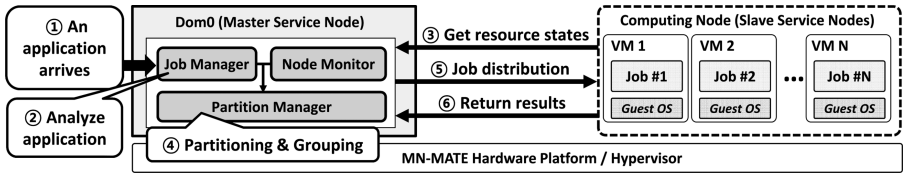


Fig. 12. Overview of the Core Federation.

In the second phase, the guest OS selects a task based on a EBW, a converted NVRAM bandwidth consumption, which has the same unit as the bandwidth consumption of DRAM. The detailed selection procedure is like follows.

(1) All tasks are arranged in order of EBW and are classified into two categories based on its bandwidth consumption. Latency-sensitive tasks consume less amount of memory bandwidth. Bandwidth-sensitive tasks spend more time to use both types of memory. Let $Task_i$ indicates a task having ith lower EBW, then we can calculate $TotalEBW = \sum_{i=1}^{n} EBW(Task_i)$. Schedulable tasks are classified into two categories. K tasks satisfying $\sum_{i=1}^{k} EBW(Task_i) < \alpha TotalEBW$ are classified into latency-sensitive tasks. Others are classified into bandwidth-sensitive tasks. Here α is a parameter $0 \leq \alpha \leq 1$, where lower α indicates a stronger threshold. We generally use $\alpha = 0.1$. Bandwidth-sensitive tasks belong to two lists; each list sorts tasks into increasing order of consumed EBW to each memory type.
(2) The scheduler chooses a latency-sensitive task prior to bandwidth-sensitive tasks. If there are multiple latency-sensitive tasks, a task consumed lower EBW has higher priority. The selection order of next task from bandwidth-sensitive tasks are like follows: 1) a task with largest EBW_D; 2) a task with smallest EBW_{NV}; 3) a task with largest EBW_{NV}; 4) a task with smallest EBW_D.

6. DESIGN FOR LIGHTWEIGHT GUEST OS: CORE KERNEL

One of the target workloads of the MN-MATE system is communication-intensive scientific calculations such as QR-factorization or 3D-perspective rendering. In MN-MATE, these workloads are distributed among computing entities, that is, VMs, to parallelize job executions. It generates application-specific needs for explicit resource reallocation. Although the MN-MATE hypervisor has functionalities to balance resources among VMs, it is beyond the responsibility of the hypervisor. Communication-intensive characteristic of the partitioned job requires faster data transfer speed.

In this section, we explains our lightweight kernel named CORE Kernel. The CORE kernel performs application-aware job distribution with the accelerated inter-VM communication scheme. It also has simplified memory management module, I/O parts, and the small number of kernel threads and daemons.

6.1. Core Federation

Partitioning a job and distributing to computing entities are basic functionalities of the lightweight kernel. To perform application-specific job partitioning and allocation, we utilized two strategies: 1) resource-aware partitioning like available VCPUs and memory capacity; 2) application-specific partitioning to maximize parallel executions considering application characteristics and input data.

Figure 12 illustrates two components of the core federation system and their operation procedures. There are two types of service nodes: master and slave. Master service node is the control tower of the core federation system, which consists of three modules. Job Manager analyzes the arrived application properties including input data distribution and the amount of calculation. Node Monitor collects current states of the

resource distribution and past balancing histories of slave service nodes. It also monitors performance information of the slaves. Partition Manager splits job into several job fragments and allocates them to nonbusy slave nodes.

The core federation is a consecutive procedure of the following operations. When an application arrives at the job manager, the manager gets the amount of calculation and the number of required threads. With the collected information from slave nodes, the partition manager determines the number of job fragments and the number of threads to executes them. The manager can start more VMs if needed. The manager can also request the hypervisor to balance resources among VMs if they are in a resource-unbalanced states. The hypervisor responds with the resource management schemes described in Section 4. The partition manager triggers executions of threads and returns collected results when all slaves complete executions.

6.2. Transparent Inter-VM Communication between CORE Kernel VMs

Because they exchange large amount of data, inter-VM communication speed is critical to the calculation performance. However, complex communication path and unnecessary packaging stack decreases communication speed. Several researches such as XenSocket [Zhang et al. 2007] or XWAY [Kim et al. 2008] proposed inter-VM communication solutions. Though they may have several advantages, they are not applicable to our target architecture. Different access latencies from hierarchical structure of on-chip M1 DRAM and M2 NVRAM and M2 DRAM degrades communication speed, thus degrading performance all over the VMs. There is no consideration about contention in the memory node where the communication buffers are located.

We proposed an inter-VM communication scheme to accelerate communication among VMs. The CORE kernel performs two operations: contention monitoring and buffer location decision. First, the guest OS periodically acquires the number of cache misses of each physical core from the hardware monitoring module described in Section 4.1. The guest OS then maintain a latency expectation table for each sender and receiver node combinations. It includes past histories of the memory copy latency from the source node to the buffer in the destination node.

Based on the collected past histories of memory copy latency, the dom0 performs a buffer selection procedure for each VM pair. When communication paths are set up, a VM pins buffers to each memory node. The number of buffers on a node is the same number of VMs. Whenever two VMs try to communicate, the dom0 allocates a buffer using locations of those VMs and the monitored memory copy latency.

We apply a buffer pipelining technique for efficient buffer usage. In the scheme, a buffer is divided into several blocks. A sender can write another block-sized data though the receiver does not read previously written data. Conversely, A receiver can read existing data regardless of the sender's operation. This technique is effective when VMs exchange large data via memory sharing and direct memory copy. Overall data transfer time depends only on the memory-to-memory copy time.

7. APPLICATION: QR DECOMPOSITION

We chose QR decomposition as our application, which is a decomposition of single matrix into a multiplication of two matrices, $A = QR$. In the equation, a given matrix A is decomposed into two matrices, Q and R. Here, Q matrix is an orthogonal matrix, which satisfies $Q^T = Q^{-1}$. R matrix is upper triangular matrix, which has nonzero elements only at the upper right part of the matrix, and zeros for other part. This decomposition is to solve linear equations and to generate Eigenvectors with following equations: $Ax = b$, $QRx = b$, $Q^T QRx = Rx = Q^T b$. Tiled QR decomposition [Buttari et al. 2009; Bouwmeester et al. 2011] divides a matrix into rectangle tiles and distributes them to processors or computing devices to parallelize it.

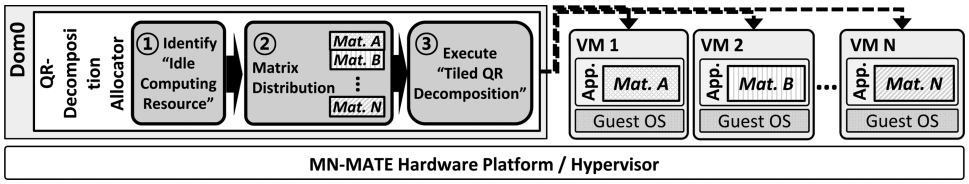


Fig. 13. An operation flow for tiled QR decomposition allocator.

In our system, which can utilize both manycore CPU and multi GPU, the QR decomposition can be accelerated easily using massive parallel cores. Because parallelism of GPU is better than CPU's, just calculating almost tiled QR steps on the GPU can make speed faster. If there are multiple users to request QR decomposition, however, because each GPU kernels are not preemptive, the wait time to use GPU for each user will be increased. Also, if workloads mostly want to use GPUs, utilization of CPU cores will not be enough, compare with GPUs.

Figure 13 shows a flow of our QR decomposition allocator module. On the first step, it monitors currently utilizable devices, speed for each request, and the input matrix size. Second, it will decide tile size to minimize cache refresh, decide how many and which devices should be participated, and allocate tiles into decided computing devices. Here, tiles with larger workload will be allocated faster device, and tiles with memory dependency will be allocated together. Of course, if input matrix size is large, multiple devices can be allocated together. Finally, after execution on each device, the result matrices will be returned to corresponding user.

8. EVALUATION

In each evaluation, hardware specification is as following unless otherwise noted. Manycores and M2 DRAM related experiments were done on HP DL585 G7 with 64 physical CPU cores and 128 GB DRAM. Because we cannot find manycore CPU with on-chip DRAM and DIMM-compatible NVRAM memory, we simulated the on-chip M1 DRAM and off-chip M2 NVRAM. NVRAM parameters are shown in Table III.

8.1. Performance Effect of Memory Balancing

To evaluate the effect of memory transfer speed to the one-time memory transfer from victim VMs to a beneficiary VM, we measured the elapsed time to reclaim the designated number of page frames. By default, all the VMs were initially allocated 256MB of memory and configured with 512MB as their highest possible memory allocation.

Figure 14 shows the elapsed time of one-time memory reclamation from the number of victim VMs. As the number of page frames reclaimed at one time increases, the elapsed time increases fast. The number of victim VMs influences the page frame reclamation speed. The more victim VMs, the longer the elapsed time to reclaim designated number of page frames. Faster transfer of page frames enlarges the amount of free memory when the guest OS requires them. It reduces the number of time-consuming memory reclamation trials if there is not enough free memory. As a result, the guest OS can respond faster to the memory allocation request of user application.

8.2. Performance of Runtime VM Migration

To evaluate our runtime VM migration algorithm targeting data access performance QoS per VM, we used the AMD Opteron NUMA machine with 4 nodes each of which

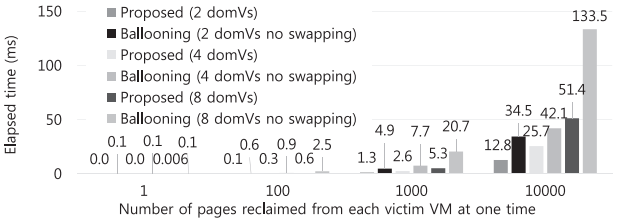


Fig. 14. Effect of balancing methods on one-time memory transfer among VMs.

Table III. Memory Specification

Power characteristics		
Parameter	DRAM	PRAM
Read Energy	3.2uJ	3.2uJ
Write Energy	3.2uJ	16uJW
Idle Power	1.3W/GB	0.05W/GB

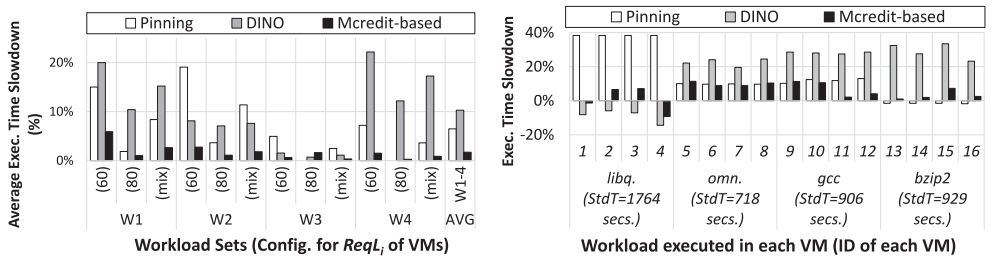
Timing characteristics		
Parameter	DRAM	PRAM
Read	15ns	28ns
Write	22ns	150ns

Table IV. List of Workload Sets for Evaluation

Set #	Workload of each 4-VM group initially placed in Node				Config. for Required Data Access Latency, $ReqL_i$ (C = Cycles)
	1 (VM1-4)	2 (VM5-8)	3 (VM9-12)	4 (VM13-16)	
W1	libq.	omn.	gcc	bzip2	1. W∗(60 or 80)'s $ReqL_i$ = 60 or 80 C, respectively.
W2	libq.	libq.	bzip2	bzip2	2. W∗(mix)'s $ReqL_i$
W3	omn.	omn.	bzip2	bzip2	= (60 C for the first two VMs in a 4-VM group.
W4	gcc	gcc	bzip2	bzip2	80 C for the rest VMs.)

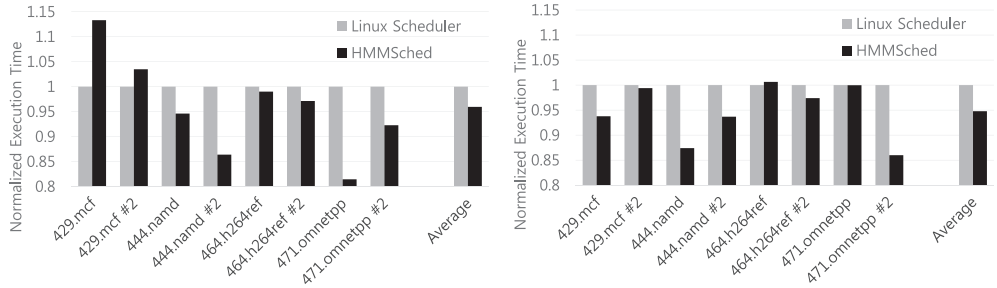
is configured with 4 cores, a shared cache, a memory controller, and 1GB DIMM. The algorithm consumes a negligible amount of CPU cycles and system memory bandwidth (1% of a cores and 0.3% of the entire system memory bandwidth, respectively). In all experiments, we simultaneously ran 16 VMs executing various SPEC CPU 2006 workloads shown in Table IV. To see whether the QoS of a VM is kept or not, we compare the execution time of the workload running on the VM with the expected execution time of the workload (which was obtained preliminarily). For comparison, we did the same experiments also under Pinning (with no migration) and DINO (with the migration for system-wide memory access performance).

Figure 15(a) describes the average execution time slowdown under various workload sets shown in Table IV and various algorithms (Pinning, DINO, and ours). The value greater than zero means that the QoS of 16 VMs is not kept well on average. W∗(60) and W∗(80) indicate that a VM in each configuration expects 60 and 80 CPU cycles for its average memory access latency, respectively. In case of W∗(mix), a half of 16 VMs expect 60 CPU cycles while the rest of them expect 80 CPU cycles for it. In almost all configurations, ours shows the lowest value of the average slowdown over those of Pinning and DINO. Under Pinning, the VMs running the workload with heavy memory access loads (e.g., libquantum) can compete with each other for a memory controller. This situation cannot be resolved as Pinning does not do any VM migration at all. Also, under DINO, a VM running the workload with light memory access loads (e.g., bzip2) will be migrated to share the same memory access path with a VM running the workload with heavy memory access loads. While DINO believes that this migration policy increases system-wide memory access performance, it makes the former VM suffer from the delay due to the heavy contention. In the other hand, ours dynamically and timely migrates VMs when it notices those VMs not receiving expected memory access performance. Figure 15(b) describes the slowdown values of individual VMs under W1(60) configuration. As we mentioned, Pinning shows high slowdown values for the VMs executing libquantum and DINO shows high slowdown values for the VMs with bzip2. In maximum, ours only has 11% of the slowdown while Pinning and DINO shows 38% and 33.4%, respectively.



(a) Average Slowdown of the VMs (Slowdown of Satisfied VMs=0) (b) per-VM Execution Time Slowdown of W1(60)

Fig. 15. Data Access Performance QoS under various algorithms.



(a) Half for CPU-intensive tasks and others for memory-intensive tasks.

(b) More memory-intensive tasks

Fig. 16. Execution times of selected benchmarks with the proposed scheduler compared with default Linux scheduler. Access latency of DRAM : NVRAM (read) : NVRAM (write) is set to (a) 1:1:3, (b) 1:3:8.

8.3. Performance of Hybrid Main Memory-aware Task Scheduling

We first evaluated the performance of the scheduler. We ran the SPEC CPU2006 benchmark [SPEC 2012] on Intel i7-960 (3.2GHz) and 6GB of DRAM, with Linux 2.6.38.2. Here, we assumed that both types of memory are controlled by a controller. Because NVRAM is not yet available for main memory and commercial processors have no per-task performance monitoring functionality, we used a Pin tool [Luk et al. 2005] to simulate both features. We attached a Pin tool and generated more consecutive memory access requests to simulate different latencies of read and write to NVRAM main memory. We also utilized the tool as an extended powerful performance monitoring unit to collect per-task memory access frequencies. It collects the number of reads and the number of writes to both types of memory for each task separately.

Figure 16 shows preliminary experimental results of the scheduler. Here, we used two access latency ratios of DRAM, NVRAM read, and NVRAM write, which is similar to the state-of-the-art STT-RAM and PRAM specification. With the first latency configuration, only NVRAM write is three times longer than others. The second latency configuration of DRAM : NVRAM (read) : NVRAM (write) is set to 1:3:8. With both latency configurations, latency-sensitive and CPU-intensive tasks execute faster because of their higher priority. Though several tasks are slightly hurt their performance, bandwidth-sensitive tasks can get some performance gain in spite of the use of same memory controller.

Table V.
Workloads set for polymorphic memory evaluation with multi-programs. Selected benchmarks are from SPEC2006 benchmark suite

#	Workloads	#	Workloads	#	Workloads
1	mcf,omnetpp,astar,soplex	12	bzip2,libquantum,omnetpp,milc	23	bzip2,sjeng,omnetpp,soplex
2	mcf,omnetpp,milc,soplex	13	libquantum,omnetpp,milc,lbm	24	bzip2,gobmk,omnetpp,lbm
3	omnetpp,astar,milc,soplex	14	mcf,sjeng,h264ref,omnetpp	25	gobmk,h264ref,omnetpp,lbm
4	bzip2,gcc,libquantum,lbm	15	mcf,sjeng,omnetpp,namd	26	h264ref,omnetpp,namd,povray
5	gcc,hmmer,libquantum,lbm	16	sjeng,omnetpp,milc,namd	27	bzip2,sjeng,h264ref,povray
6	bzip2,hmmer,libquantum,lbm	17	mcf,sjeng,milc,name	28	gcc,mcf,hmmer,lbm
7	gobmk,sjeng,h264ref,namd	18	sjeng,libquantum,namd,lbm	29	mcf,omnetpp,milc,lbm
8	gobmk,h264ref,namd,povray	19	bzip2,sjeng,libquantum,namd	30	omnetpp,astar,milc,namd
9	sjeng,h264ref,namd,povray	20	libquantum,h264ref,namd,lbm	31	bzip2,gcc,namd,lbm
10	bzip2,mcf,omnetpp,lbm	21	bzip2,gcc,hmmer,lbm		
11	mcf,libquantum,milc,lbm	22	mcf,h264ref,omnetpp,lbm		

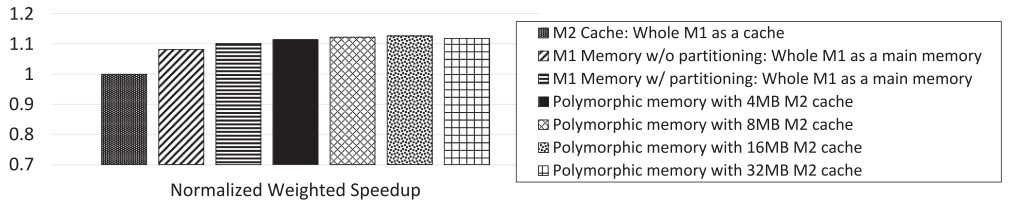


Fig. 17. Average performance results of multi-program workloads with various polymorphic memory configurations.

8.4. Performance of Polymorphic Memory

We built a simulator based on Pin [Luk et al. 2005] instrumentation tool to evaluate performance effect of the polymorphic management technique. To evaluate various multi-program cases, we configured 31 benchmark sets with the workloads in SPEC CPU2006. Table V summarizes 31 sets of workloads. Each set was classified by the access frequency and the page coverage.

We measured the performance of the management policy of M1 memory by using $WeightedSpeedup = \sum_i \frac{IPC_{shared,i}}{IPC_{alone,i}}$ [Snavely and Tullsen 2000]. All experiments are conducted for 5 billion cycles, including 1 billion cycles of warming up.

Figure 17 shows the average values of normalized weighted speedup in multi workloads. The results show that the performance of M1 memory which means that whole on-chip memory is used as a main memory is better than that of M2 cache which means that whole on-chip memory is used as a cache. By using partitioning technique, the performance can be improved by 2%. In addition, the polymorphic memory can improve the performance compared to M1 memory cases. The results show that the performance of polymorphic with 16MB M2 cache is the best and on average, the performance improvement is 12.72% compared to the case of M2 cache.

8.5. Performance of Hybrid Page Cache

Next we evaluated the performance of hybrid page cache. We ran *financial1* workloads on a trace-driven simulator and the OLTP traces [UMass TraceRepository 2007]. We selected the values of parameter α, Tr_{mig} and Tr_q as 0.5, 0.5, and 0.35, respectively.

For the amounts of DRAM and PRAM, because PRAM density is expected to be four times higher than DRAM [Park et al. 2010b; Wu et al. 2009], we mainly allocated four times larger amount of memory to PRAM in this experiment.

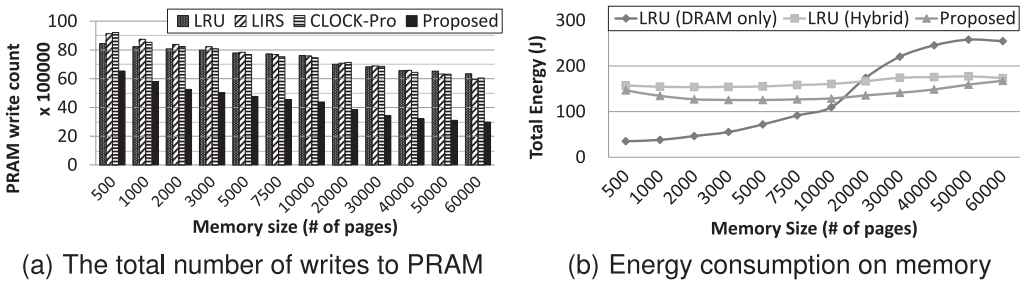


Fig. 18. Performance and energy consumption comparison for the hybrid page cache with *financial1* workload. The total number of writes to PRAM directly affects page cache performance.

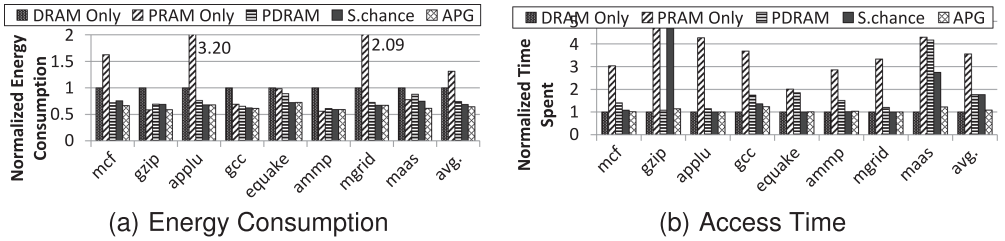


Fig. 19. Performance and energy consumption comparison for the power-aware hybrid main memory.

Figure 18(a) shows the total number of write accesses on PRAM, which is strictly related to the total latency of the page cache and the lifetime of PRAM. When using our algorithm, we can see that the total number of write accesses is reduced compared to that for the conventional page caching algorithm. We can reduce the total write access count by a maximum of 52.9%. Figure 18(b) shows the total energy consumption. Here, our algorithm can reduce the energy consumption by 19.9%. Therefore, we can enhance the average page cache performance and reduce the endurance problem in the hybrid main memory.

8.6. Performance and Energy Efficiency of Power-aware Hybrid Main Memory

Figure 19(a) shows energy consumption of the memory system. Here, the energy consumption includes static power (standby power, refresh power), dynamic power (active power, read power, write power), and migration power. In the cases of *mcf*, *applu*, and *mgrid*, PRAM only system spent more energy than DRAM only system. Because PRAM has lower static power but higher write power and write latency, workloads which have high write operations per instructions spend more time for writing, and this additional time cancels out profit of low static power. For 5GB hybrid memory system, which is comprised of 1GB of DRAM and 4GB of PRAM, saves more energy than 2GB memory system (1GB of DRAM and 1GB of PRAM), because low static power of PRAM take up the more portion of total energy consumption.

Figure 19(b) shows the memory access time for the read, write, and migration. The most impactive variable which determines the performance of memory system is the number of write operations on PRAM. Only DRAM system has the smallest execution time while only PRAM system has the largest execution time. In case of *gzip* using s.chance algorithm, it spent long time because of ping-pong migration of second chance algorithm. APG system had a delay incremented by 8% compared to only DRAM system on average, 38% decreased compared to PDRAM or s.chance algorithm.

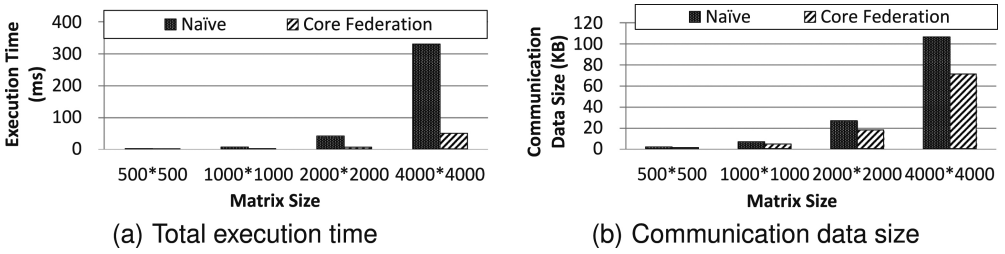


Fig. 20. Performance effect and its analysis for core federation with a matrix multiplication workload. Smaller amount of data transfer significantly reduces workload execution time.

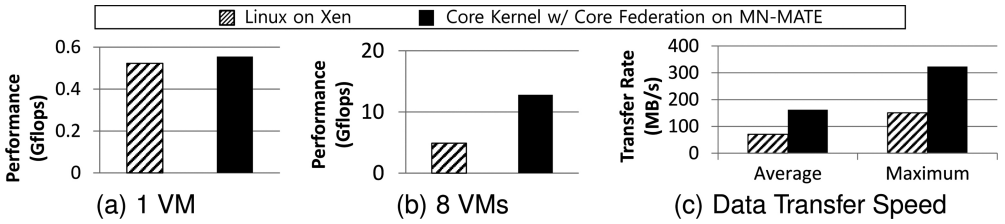


Fig. 21. Performance effect of Core kernel with core federation and inter-VM communication schemes.

Our hybrid main memory management system reduced energy consumption by 36% while minimizing performance degradation. We can apply our system immediately as a software patch when PRAM is released, because all these schemes are implemented in Linux guest OS without additional hardware.

8.7. Performance Effect of Core Federation on Core Kernel

We evaluated the performance of core federation in terms of communication data size by comparing with a naive approach. We ran a matrix multiplication workload generating input matrices for size 500 by 500 to 4000 by 4000 with 30% density randomly. We implemented naive approach as MPI-based partitioning with blocked dense matrix multiplication [Nimako et al. 2012] for comparison. Here, each slave VM had 8 VCPUs.

Figure 20(a) shows the total execution time of matrix multiplication. We can see that the core federation improves performance compared to naive approach for every matrix size. With balancing of the execution time of partitioned thread on core federation, we could get 81.4% times faster execution time compared to naive approach. Figure 20(b) shows the total communication data size during the matrix multiplication. Since the Core Federation parallelizes the matrix by considering the density of matrix and transfers only nonzero data, it has smaller communication data. We could get 33.1% reduced communication data size compared to naive approach.

In addition, we performed an additional experiment to show the effectiveness of the core federation and the inter-VM communication schemes on Core Kernel. Figure shows results with HPL benchmark, where both schemes are turned on. With the benefit of the lightweight kernel, Core kernel showed better performance even with a single VM. Because of the enhanced data communication speed shown in Figure 21(c), we could get improved application performance compared with the application on standard Xen-virtualized Linux guest OS.

9. RELATED WORK

There were several approaches for efficient management of the manycore server to overcome the technical limitations of CPU and DRAM.

As a way to increase the memory bandwidth, [Woo et al. 2010] employs a 3D-stacked memory architecture which is placed in the CPU processor. They propose a new 3D-stacked memory architecture with a vertical L2 fetch/write-back network using a large array of through-silicon-vias (TSVs). It shows higher bandwidth and lower latency than those of conventional DRAM memory. [Kgil et al. 2006; Liu et al. 2005; Loi et al. 2006] also talked about on-chip memory. Currently, the size of on-chip memory can be increased up to 16GB in the stack [Woo et al. 2010].

Hybrid main memory architecture of DRAM and NVRAM is a promising architecture to enhance energy efficiency while preserving accessing performance and providing larger memory capacity. Qureshi et al. [2009] proposed a hybrid architecture of DRAM and PRAM where both media are located at the same level.

Several software methods are proposed to manage resources like manycores and various memory media in the memory hierarchy. In the hypervisor, several researches have been proposed to optimize memory access performance in the NUMA organization of the main memory. While the earlier works [Govil et al. 1999; Bligh et al. 2004; VMware 2005] only focused on distance between the VCPUs and the memory of VMs, recent researches [Majo and Gross 2011; Blagodurov et al. 2011] additionally considered shared resource contentions. DINO [Blagodurov et al. 2011] suggested a solution to balance memory access loads across NUMA nodes by migrating the VCPUs and memory of VMs. While these works with NUMA-awareness have concentrated on the system-wide performance, our VM placement solution is orthogonal to them as we tried to guarantee a certain level of memory access performance to each VM especially under heavy contention among VMs.

For the memory balancing among VMs in the hypervisor, [Waldspurger 2002; Lu and Shen 2007; Magenheimer 2008, 2009; Zhao and Wang 2009; Schwidefsky et al. 2006] already proposed several memory balancing methods. Zhao and Wang [2009] is the most recent work in the memory balancing area. It decides the proper memory size of each VM through working set size estimation. In Zhao and Wang [2009], the victim VMs inflate the balloon driver to release memory, and the beneficiary VMs subsequently deflate the balloon driver to make the guest OS control the acquired memory. It utilizes its own memory reclaiming policy with regard to the victim page selection of the guest OS. However, the policy also causes a swap out or data flush of dirty pages to the corresponding storage location in accordance with the policy of the guest OS. Feedback directed ballooning [Magenheimer 2008] also mentioned a similar solution with the same weak points. We expect that our balancing method is more efficient and it will also be effective with NVRAM.

From the perspective of the guest OS, several researches [Jiang et al. 2010; Woo et al. 2010; Iyer 2003; Zhang et al. 2004] introduced on-chip DRAM as a last level cache. It shows a good performance gain with memory-intensive applications, but Zhao et al. [2007] pointed out a scalability problem. With the large size up to 16GB [Woo et al. 2010], the overhead of a tag array will grow to several MBs. Previous works tried to reduce the overhead of the tag array by increasing the cache line size. However, this increases memory bandwidth and decreases on-chip memory utilization. Furthermore, in order to access cached data, a cache tag must be accessed first and then data can be accessed. It doubles the latency for accessing data [Dong et al. 2010].

For the energy efficiency of the main memory structure, [Park et al. 2011b] suggested a power saving technique reducing the refresh energy of DRAM. Rank-based Page Placement (RaPP) [Ramos et al. 2011] is proposed as a memory management mechanism designed for hybrid main memory, using rank-based migration techniques to provide lower energy-delay. Regardless of the implementation method, hotness of the past may misplace pages with dynamically changing access patterns. We tried to reduce the misplacement like ping pong case by grouping pages. We compared our

solution with the two previous works utilizing page-level placement. In our previous works, we propose several management algorithms to utilize hybrid main memory with DRAM and NVRAM [Park et al. 2010a, 2011a; Shin et al. 2012]. However, they are for desktop-sized workloads.

For hybrid page cache, LIRS was proposed to solve the problems of LRU [Jiang and Zhang 2002] by using Inter-Reference Recency (IRR). It selects pages with large IRRs for replacement. CLOCK can reduce the overheads of the LRU algorithm, which are related to the overhead of moving a page to the MRU position on every page hit [Bansal and Modha 2004]. CLOCK-Pro [Jiang et al. 2005] was also proposed based on CLOCK, which is a simple approximation of the LRU replacement algorithm [Corbato and MAC. 1968; Carr and Hennessy 1981]. It covers the ways how LIRS and CLOCK work. In addition, several caching algorithms such as FBR [Robinson and Devarakonda 1990], LRU-2 [O'Neil et al. 1993], 2Q [Johnson and Shasha 1994], LRFU [Lee et al. 2001], and MQ [Zhou et al. 2001] were proposed to combine recency and frequency. They tried to compensate for the LRU's disadvantages from sequential or cyclic access pattern with larger than cache. However, all these methods cannot consider different physical properties of hybrid memory.

For inter-VM communication operations, XenSocket [Zhang et al. 2007] provides socket interface with simplified communication path. It reduces overhead from eliminating TCP stack. XWAY [Kim et al. 2008] proposed a communication scheme using shared memory. It intercepts communication requests in the kernel so that the data is sent through XWAY switch if destination IP is in local machine. However, both approaches did not consider different memory access latencies from NUMA architecture.

10. CONCLUSION

As manycore system accommodates multiple guest OSes and tasks inside them, resource management become a key issue. More concurrent tasks increases needs for larger and faster memory hierarchy, memory wall decreases benefit from the concurrent executions of tasks, especially in the virtualized environment. To provide enough computing resources for consolidated VMs in a energy-efficient manner, we targets a manycore system with a memory hierarchy of on-chip DRAM, off-chip DRAM and off-chip NVRAM. In this article, we provided the MN-MATE, an energy-efficient, full-fledged resource management system for consolidated VMs in a single hardware node. We proposed a runtime placement policy of consolidated VM and a memory balancing scheme among them in the hypervisor. On top of this, We build two guest OS kernels, Linux and our own lightweight kernel named CORE kernel. With the balanced CPU and memory resources, we proposed a resource-aware task scheduling algorithm, polymorphic management scheme of on-chip/off-chip DRAMs, and data placements and management policies for hybrid main memory of off-chip M2 DRAM and off-chip M2 NVRAM for the Linux guest OS. We also build a simplified kernel to accelerate execution of scientific applications through NUMA-aware buffer selection mechanism. Consequently, MN-MATE significantly enhanced system performance with the new hardware system with an improved energy efficiency. We can see performance improvement with our applications, core federation and QR factorization. The experiment results show that MN-MATE outperforms other competing resource management schemes in terms of application execution time and energy consumption.

REFERENCES

AMD. 2013. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h. Tech. Document.

Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX Association, 187–200.

S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. 2011. A case for NUMA-aware contention management on multicore processors. In *Proceedings of the Usenix Annual Technical Conference*.

Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. 2010. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, 557–558.

Martin J. Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. 2004. Linux on NUMA systems. In *Proceedings of the Linux Symposium*, Vol. 1. 89–102.

H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert. 2011. Tiled QR factorization algorithms. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1, 38–53.

Richard W. Carr and John L. Hennessy. 1981. WSCLOCK&Mdash: A simple and effective algorithm for virtual memory management. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP'81)*. ACM, New York, 87–95. DOI:http://dx.doi.org/10.1145/800216.806596

F. J. Corbato and MIT Cambridge Project MAC. 1968. A Paging Experiment with the Multics System. Defense Technical Information Center.

Asit Dan and Don Towsley. 1990. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 143–152.

Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. 2010. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE, 1–11. DOI:http://dx.doi.org/10.1109/SC.2010.50

Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. 1999. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM, 154–169.

R. Iyer. 2003. Performance implications of chipset caches in web servers. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 176–185.

Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference (ATEC'05)*. USENIX Association, Berkeley, CA, 35–35.

Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 31–42.

Xiaowei Jiang, N. Madan, Li Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture*. 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2010.5416642

Theodore Johnson and Dennis Shasha. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. Morgan Kaufmann, San Francisco, CA, 439–450.

Taeho Kgil, Shaun D'Souza, Ali Saidi, Nathan Binkert, Ronald Dreslinski, Trevor Mudge, Steven Reinhardt, and Krisztian Flautner. 2006. PicoServer: Using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 117–128.

Kangho Kim, Cheiyol Kim, Sung-In Jung, Hyun-Sup Shin, and Jin-Soo Kim. 2008. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'08)*. ACM, New York, 11–20. DOI:http://dx.doi.org/10.1145/1346256.1346259

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, 2–13.

D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 50, 12, 1352–1361. DOI:http://dx.doi.org/10.1109/TC.2001.970573

Christianto C. Liu, Ilya Ganusov, Martin Burtscher, and Sandip Tiwari. 2005. Bridging the processor-memory performance gap with 3D IC technology. *IEEE Des. Test* 22, 6, 556–564.

Gian Luca Loi, Banit Agrawal, Navin Srivastava, Sheng-Chih Lin, Timothy Sherwood, and Kaustav Banerjee. 2006. A thermally-aware performance analysis of vertically integrated (3-d) processor-memory hierarchy. In *Proceedings of the 43rd Annual Design Automation Conference (DAC'06)*. ACM, 991–996.

Pin Lu and Kai Shen. 2007. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, 1–15.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, 190–200. DOI:http://dx.doi.org/10.1145/1065010.1065034

Dan Magenheimer. 2008. *Memory Overcommit... without the Commitment*. Xen Summit.

Dan Magenheimer. 2009. *Transcendent Memory on Xen*. Xen Summit.

Z. Majo and T. R. Gross. 2011. Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the International Symposium on Memory Management*. ACM, 11–20.

Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. 2012. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *IEEE Comput. Archit. Lett.* 11, 2, 61–64. DOI:http://dx.doi.org/10.1109/L-CA.2012.2

G. Nimako, E. J. Otoo, and D. Ohene-Kwofie. 2012. Fast parallel algorithms for blocked dense matrix multiplication on shared memory architectures. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'12)*: Part I. Springer, 443–457. DOI:http://dx.doi.org/10.1007/978-3-642-33078-0_32

Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*. ACM, New York, 297–306. DOI:http://dx.doi.org/10.1145/170035.170081

Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. 2011b. Power management of hybrid DRAM/PRAM-based main memory. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference*. 59–64.

Kyu Ho Park, Sung Kyu Park, Woomin Hwang, Hyunchul Seok, Dong-Jae Shin, and Ki-Woong Park. 2012a. Resource management of manycores with a hierarchical and a hybrid main memory for MN-MATE cloud node. In *Proceedings of the 8th IEEE World Congress on Services*. 301–308. DOI:http://dx.doi.org/10.1109/SERVICES.2012.26

Kyu Ho Park, Sung Kyu Park, Hyunchul Seok, Woomin Hwang, Dong-Jae Shin, Jong Hun Choi, and Ki-Woong Park. 2012b. Efficient memory management of a hierarchical and a hybrid main memory for MN-MATE platform. In *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'12)*. ACM, New York, 83–92. DOI:http://dx.doi.org/10.1145/2141702.2141712

Kyu Ho Park, Youngwoo Park, Woomin Hwang, and Ki-Woong Park. 2010b. MN-Mate: Resource management of manycores with DRAM and nonvolatile memories. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*. 24–34. DOI:http://dx.doi.org/10.1109/HPCC.2010.35

Youngwoo Park, Sung Kyu Park, and Kyu Ho Park. 2010a. Linux kernel support to exploit phase change memory. In *Proceedings of the Linux Symposium*. 217–224.

Youngwoo Park, Dong-Jae Shin, Sung Kyu Park, and Kyu-Ho Park. 2011a. Power-aware memory management for hybrid main memory. In *Proceedings of the 2nd International Conference on Next Generation Information Technology*. 82–85.

Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, 24–33. DOI:http://dx.doi.org/10.1145/1555754.1555760

Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, 85–95. DOI:http://dx.doi.org/10.1145/1995896.1995911

John T. Robinson and Murthy V. Devarakonda. 1990. Data cache management using frequency-based replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*. ACM, New York, 134–142. DOI:http://dx.doi.org/10.1145/98457.98523

Martin Schwidefsky, Hubertus Franke, Ray Mansell, Damian Osisek, Himanshu Raj, and Jonghyuk Choi. 2006. Collaborative memory management in hosted linux systems. In *Proceedings of the Ottawa Linux Symposium*.

Dong-Jae Shin, Sung Kyu Park, Seong Min Kim, and Kyu Ho Park. 2012. Adaptive page grouping for energy efficiency in hybrid PRAM-DRAM main memory. In *Proceedings of the ACM Research in Applied Computation Symposium*. ACM, 395–402.

Allan Snavely and Dean M. Tullsen. 2000. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 234–244. DOI:http://dx.doi.org/10.1145/378993.379244

SPEC. 2012. Spec's benchmark. http://www.spec.org/cpu2006.

UMass TraceRepository. 2007. OLTP Application I/O and Search Engine I/O. http://traces.cs.umass.edu/index.php/Storage/Storage.

VMware. 2005. ESX Server 2 NUMA Support. WhitePaper, http://www.vmware.com/pdf/esx2_NUMA.pdf.

Carl A. Waldspurger. 2002. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*.

Dong Hyuk Woo, Nak Hee Seong, D. L. Lewis, and H.-H. S. Lee. 2010. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture*. 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2010.5416628

Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. 2009. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, 34–45. DOI:http://dx.doi.org/10.1145/1555754.1555761

Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. 2007. XenSocket: A high-throughput interdomain transport for virtual machines. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware (Middleware'07)*. Springer, 184–203. http://dl.acm.org/citation.cfm?id=1516124.1516138

Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2004. Design and optimization of large size and low overhead off-chip caches. *IEEE Trans. Comput.* 53, 7, 843–855. DOI:http://dx.doi.org/10.1109/TC.2004.27

Li Zhao, R. Iyer, R. Illikkal, and D. Newell. 2007. Exploring DRAM cache architectures for CMP server platforms. In *Proceedings of the 25th International Conference onComputer Design (ICCD'07)*. 55–62. DOI:http://dx.doi.org/10.1109/ICCD.2007.4601880

Weiming Zhao and Zhenlin Wang. 2009. Dynamic memory balancing for virtual machines. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*. ACM, 21–30.

Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 91–104. http://dl.acm.org/citation.cfm?id=647055.715773

Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, 129–142. DOI:http://dx.doi.org/10.1145/1736020.1736036

Received March 2014; revised July 2014, October 2014; accepted December 2014