HMMSched: Hybrid Main Memory-Aware Task Scheduling on Multicore Systems

Woomin Hwang and Kyu Ho Park
Computer Engineering Research Laboratory, KAIST
Daejeon, Republic of Korea
{wmhwang, kpark}@core.kaist.ac.kr

***Abstract*—The strong demand for larger memory capacity with high energy efficiency creates the need for a hybrid main memory of DRAM and NVRAM (Non-Volatile RAM). In an attempt to provide task co-scheduling method on hybrid main memory, we found that the conventional task scheduling algorithms do not reflect the features of different memory mediums into scheduling decisions. Long access latency of the NVRAM make memory bandwidth usage of each task misestimated, thereby incurring unpredictable contention on memory accesses. Different access latencies according to the access type to the NVRAM deteriorates the contention. As a remedy to these problems, we propose HMMSched, which reflects different access latencies of hybrid main memory to the bandwidth estimation. Our scheme prioritizes CPU-intensive tasks, and then co-schedules tasks having complementary bandwidth consumption on different memory medium. Consequently, HMMSched reduces execution time of tasks by reducing memory access contention of co-scheduled tasks on the same memory. The experimental results show that the new scheduler reduces the overall execution time up to 19% than the default Linux scheduler.**

Keywords-Hybrid Main Memory; NVRAM; PRAM; task scheduling; memory contention;

I. Introduction

Increased concurrency of the workload executions brings not only great advances in system performance but limitations. Increasing number of computing cores achieve high instruction per cycle (IPC) by executing multiple instruction streams concurrently on same chip. The more tasks run concurrently, however, the greater the demand of memory because of the size of aggregated working set. It causes problems in the aspects of capacity, energy, and access contention. Enlarging memory size is expected to be limited by the clamped scalability of DRAM [1], [2]. Energy consumption of memory occupying 30-40% of server system energy [3], [4] will grow with the increase of DRAM size. Contention on memory accelerated by the use of many-core processors is a well-known problem on DRAM-based main memory. Cores share parts of the memory hierarchy and compete for resources. It brings delay in memory accesses, which results in degradation of application performance. As solutions, several studies proposed contention-aware scheduling [5], [6], [7], [8], [9], [10], [11], [12] to reduce memory contention. A contention-aware scheduler detects tasks having potentials of competing for memory hierarchies. Found tasks are scheduled at different time and location.

Hybrid main memory of DRAM and NVRAM is a promising architecture to enlarge memory capacity with high energy efficiency and little performance loss [3], [13], [14]. By placing both media at the same level of memory hierarchy, both memory are complementary to each other. Non-volatility of the NVRAM can reduce the total energy consumption of the main memory compared with DRAM-only main memory of the same capacity. DRAM complements the demerits of NVRAM on performance caused by its longer and asymmetric access latencies.

Unfortunately, existing studies of contention-aware task scheduling algorithms focused on DRAM main memory. With DRAM-only main memory, access latencies to the same memory bank are independent of access request type. In the hybrid main memory architecture, on the contrary, access latency varies according to the medium of the target memory, memory access type, and the location of the target memory on NUMA (Non-Uniform Memory Access) architecture.

NVRAM read latency is longer than DRAM access latencies, and NVRAM write latency is much longer than NVRAM read. When we ran hybrid main memory-agnostic contention-aware scheduling algorithms on the target system, we found that they do not aware resource usage well. It results in performance degradation. We have summarized our motivation about the performance degradation.

- **Multiple tasks concurrently accessing same type of memory increases contention:** Hybrid main memory is a combination of DRAM and NVRAM, where larger size of NVRAM replaces DRAM. With them, frequently accessed data are located in DRAM while NVRAM maintains data generating burst accesses [14]. This deteriorates task performance when memory-intensive tasks accessing same memory bank are co-scheduled. Concurrent tasks accessing hot data on reduced size of DRAM generate more contention like shown in Fig. 3. Overlapped burst accesses to NVRAM amplify longer access latencies of the medium.
- **The number of memory transactions does not reflect bandwidth occupancy of a task:** Existing contention-aware scheduling algorithms [8], [9], [10], [15], [16] estimates memory bandwidth usage by counting the

number of memory transactions. In the hybrid main memory environment, however, a write to NVRAM occupies target memory longer than a DRAM read. Type-agnostic counting of memory transactions to the NVRAM makes the value not proportional to the actual memory bandwidth usage. It incurs wrong decision of the task scheduler that tries to avoid memory contention based on the bandwidth usage of each task.

In this paper, we propose HMMSched, a task co-scheduling method for tasks running on hybrid main memory. HMMSched collects memory access information for each task to acquire memory bandwidth usage of each type. We define that Effective Bandwidth (EBW) is the estimated bandwidth considering the different access latencies of each memory medium. It translates bandwidth usage of NVRAM to the same unit of DRAM's not to make the estimated bandwidth fluctuate according to the target medium and access type of the memory access requests.

HMMSched performs following two phases to determine the next task to schedule. On the first phase, HMMSched prioritizes tasks having lower EBW consumption than the predetermined threshold. It is from the fact that a task having lowest EBW have greatest potential to make progress in the CPU. As the second phase, HMMSched separates EBW of each task according to its target medium for the remaining schedulable tasks. The tasks having low complementary EBW on different memory medium are alternately co-scheduled.

Our algorithm is implemented in Linux kernel with simulated per-task performance monitoring unit (PMU) attached to each workload. The preliminary experiment results show that our algorithm outperforms up to 19% better than the default Linux scheduler.

This study is an extension of our previous work [17], which is a part of resource management for the MN-MATE computing platform [18]. In the previous work, we focused on finding problems on task scheduling on hybrid main memory environment with small number of cores. Our objective in this paper, however, is to devise task scheduling policy with more consideration on real multicore execution environment and implementation issues.

The rest of this paper is organized as follows. Section II explains background of this work and Section III discusses related work. Section IV describes our motivation of this paper. Section V presents design of HMMSched and its implementation issues. Section VI provides the experimental results, and we conclude our work in Section VII.

II. Background

In this section, we explain two non-volatile memories as DRAM alternatives. Then we introduce a hybrid main memory, which is the target memory architecture of this paper.

Table I. Access latencies of memory technologies. Data obtained from [20], [21]

Technology	Latency (ns)	
	Read	***Write***
DRAM	25	25
STT-RAM	29.5	95
PRAM	67.5	215

A. Non-Volatile Memories

Non-volatile memory (NVRAM) technologies, unlike DRAM, will possibly enable memory chips that are non-volatile, require low-energy, and have density and latency closer to current DRAM chips [19]. Among emerging memories, Phase Change Memory (PRAM) and Spin-Transfer Torque RAM (STT-RAM) is one of the most promising technologies for future memory.

1) Phase-Change Memory (PRAM): PRAM is a byte-addressable, non-volatile memory based on phase change materials that can sustain phase persistently [22]. Persistent sustenance of phase gives non-volatility to PRAM, which makes leakage energy negligibly small. Because the length of the current pulse decides the direction of state change, a PRAM cell can be switched from 0 to 1 and vice-versa without any erase operation. It gives the memory in-place update property. PRAM is argued to be a scalable technology [19], [22], [1] for its high density. Recent works [4], [1] have demonstrated that the scalability will make PRAM a promising DRAM alternative for main memory.

On the other hand, there are several disadvantages. First, both read and write latencies of PRAM are several times slower than DRAM. Table I summarizes read and write latencies of main memory candidates. Second, large write energy mostly consumed by the reset operation is also one of the weak points of the PRAM. It makes the energy usage of PRAM depend on the number of writes. We will not cover the endurance problem of 10^8 rewrite sustainability [23].

2) Spin-Transfer Torque RAM: STT-RAM is another byte-addressable, non-volatile memory. It applies different write mechanism based on spin polarization [24], [25]. Compared with the PRAM access latencies, STT-RAM has very low read and write latencies. Read latency of STT-RAM is as fast as of DRAM according to [20]. Though the write latency is slower than DRAM, it is still very fast compared with the PRAM. It also has better endurance, reaching above 10^{15} cycles [24]. The weak point of STT-RAM is density. Its cell size has less density than current DRAM cells, shown in Table I. Smaller capacity from lower density depletes fast access speed, which is as fast as DRAM.

B. Hybrid Main Memory

Hybrid main memory is a combined architecture of DRAM and NVRAM for main memory [3], [13], [14], where both memory is located at the same level of memory

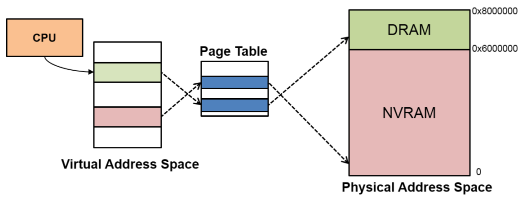


Figure 1: Hybrid main memory architecture. Both NVRAM and DRAM are located at the same level in the memory hierarchy.

hierarchy. It is designed to exploit the strenghs of each memory medium while avoiding their weaknesses as much as possible. By maintaining both media as a main memory at the same level, DRAM and NVRAM complements weak points of each other. Higher density of NVRAM provides main memory scalability, which will be limited by the DRAM. Lower standby power of NVRAM enlarges main memory capacity with lower energy consumption. Furthermore, shorter access latency of DRAM reduces performance loss due to long access speed of NVRAM [4].

In this paper, we target PRAM and STT-RAM as candidates for DRAM alternatives of main memory. Their byte-addressability and in-place update property enable them to be directly attached to the main memory interface, though they need separate memory controller. It makes the memory hierarchy provide both low latency and non-volatility. There are two studies [13], [14] providing kernel supports for the efficient use of the hybrid main memory architecture.

Fig. 1 shows our target structure of hybrid main memory architecture. In this architecture, NVRAM and DRAM are assigned to single physical memory address space. NVRAM has either lower or higher, contiguous physical address. OS can distinguish the medium of the target memory based on their physical address. OS manages all physical page frames and controls data placements based on reference characteristics. Any tasks can use both types of memory through the kernel.

There are some researches [26], [27] to extends the memory capacity with flash memories not attached to the main memory interface. However, they use either DRAM or Flash memory as a data cache or swap storage, which are not our concern in this paper. We motivated from the possibilities with the memory architecture having different read/write latencies. Therefore we only targets byte-addressable NVRAM and DRAM directly attached to the CPU at the same level of memory hierarchy.

III. Related Work

In this section, we explain previous researches about task co-scheduling. Contemporary studies present solutions that enhance system-wide performance in a system with multiple number of cores and DRAM-only main memory.

Memory bandwidth is one of the most popular criteria for memory-aware task scheduling. Koukis and Koziris [5], [6] proposed a scheduling method for SMP clusters. They profile bandwidth usage of each process and calculate average available bandwidth for new task. The scheduler selects the most fitting process whose bandwidth consumption best matches the average available bandwidth. Though it utilizes per task memory bandwidth consumption, it targets SMP clusters, which does not share caches.

In addition, Xu et al. [8] quantified the impact of memory bandwidth fluctuation on overall performance for tasks on multicore system. They found that bandwidth fluctuation measured using very fine time intervals could distorts the parameters of the job scheduler, especially when the total bandwidth usage approaches to peak system bandwidth. They proposed new scheduling criteria maintaining the total bandwidth requirement at a steady level instead of maximizing bandwidth utilization. Though they enhanced performance by reducing fluctuation in bandwidth consumption of the workloads, they targeted tasks running on a system with DRAM main memory. We borrowed some of their idea by not selecting tasks to fill in the available bandwidth. HMMSched co-schedules tasks by prioritizing latency-sensitive tasks followed by matching task consuming largest bandwidth. It doesn't compare bandwidth consumption with the available system bandwidth to fill in.

Miss rate of shared caches among cores is also a popular criterion for task co-scheduling on DRAM main memory. El-moursy et al. [7] tried to select thread pairs on each Simultaneous Multi-Threaded (SMT) processors combined with DRAM main memory. They add new hardware PMU and monitor contention on functional unit, register files, and caches. They define a phase as changing utilization of target resource and tried to find compatible phases that generate less conflicts on the resource. Compatible threads minimizing total cache miss rate are co-scheduled on the same SMT core.

Zhuravlev et al. [9] and Blagodurov et al. [10] also proposed a contention modeling heuristic named Distributed Intensity on DRAM main memory. They found that memory bus, prefetch hardware, and DRAM controller affecting performance degradation rely on cache miss rate. Based on the observation, they assigned threads to caches to even out the miss rate across all the caches. Though it showed good performance enhancement and performance stabilization, we target different execution environment where it utilizes NVRAM as a part of main memory. In addition, recent studies [15], [16] give an insight that miss rates does not reflect memory bandwidth because of the different write latency in PRAM main memory. Therefore, it is hard to apply to systems with hybrid main memory.

Several researches try to reduce energy consumption by scheduling tasks. Merkel and Bellosa [28] proposed memory-aware scheduling for energy efficiency on DRAM.

They also used performance counters to measure memory intensity. After sorting tasks on runqueues of cores, they paired cores. The scheduler selected two tasks from each runqueue, one having smallest memory intensity and the other one having largest memory intensity. Selected tasks are co-scheduled during time calculated by the unit time divided by the number of tasks in the source runqueue. They also applied frequency scaling for energy efficiency. DeVuyst et al. [29] also found that unbalanced number of threads on each core bring less energy consumption.

Suleman et al. [11] and Bhadauria and McKee [12] controlled the number of homogeneous threads to maintain bandwidth usage below the bandwidth limit. They utilized change of concurrency level on scheduling. But it is hard to apply tasks with noncontrollable concurrency. Again, their target systems include DRAM only while our target system includes NVRAM main memory.

Data placement is another issue on hybrid main memory for its effect to access patterns. Park et al. [14] proposed a data placement and migration policy between DRAM and PRAM at the same level in memory hierarchy. Ramos et al. [30] also proposed page placement methods where small-sized DRAM and large size PRAM is combined in a memory system. While they focused on the location of data and has little scheduling features, it affects access pattern of running tasks. Those policies can be complementarily cooperated with our scheduling policy in spite of the changed memory environment.

In this paper, we consider scheduling for manycores with hybrid main memory for objectives that are a composition of both performance and energy.

IV. MOTIVATION

Previous researches concentrate on memory bandwidth usage estimation on an environment where DRAM is used as a main memory. However, these approaches are inappropriate for the hybrid main memory of DRAM and NVRAM. In this section, we explain our motivations in more detail.

A. Co-scheduling contention on Hybrid Main Memory

In the hybrid main memory environment, co-scheduling tasks intensively accessing same type of memory generates contention in several locations.

Fig. 2 shows contention on DRAM of hybrid main memory according to the task type. With 444.namd, which is classified as CPU-intensive task, accessing same bank of memory has little effect on performance due to low memory access intensity. They undergo rather little slowdown. With memory-intensive task such as 429.mcf, however, task performance is greatly affected by the contention on memory. The more memory accesses converge to the same memory bank, it makes execution time of each task longer.

The situation is getting worse with the hybrid main memory of DRAM and NVRAM. With the hybrid main memory,

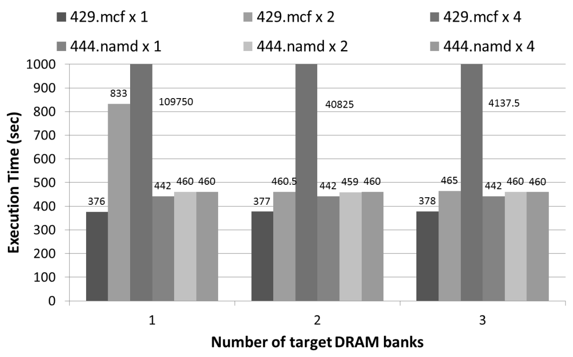


Figure 2: Slowdown of tasks according to the number of target memory banks and the number of concurrent tasks.

frequently accessing or short-lived data are usually located in DRAM while others are stored in NVRAM [3], [13], [14]. Memory accesses are flocked to the reduced number of DRAM banks. It increases memory access intensity, which is the number of memory accesses during a unit time. NVRAM accesses tend to be burst for the stored data's high spatial locality and low popularity. The performance deteriorates if the number of target memory bank decreases.

B. Inappropriate Bandwidth Estimation

Memory bandwidth (BW) usage has been one of the popular criteria for task scheduling working on DRAM main memory [5], [6], [8], [31]. With DRAM, access time of the memory does not depend on the type of the memory access request. Therefore, BW usage can be represented by the number of memory transactions generated during a unit time, shown in (1).

$$BW \approx \frac{Number\ of\ Memory\ Transactions}{Unit\ Time} \quad (1)$$

Several researches try to relieve the unequal distribution of memory bandwidth usage with the counted memory bandwidth [5], [6], [8]. Basic approaches of previous researches are to select next task consumed memory bandwidth not greater than the available bandwidth. More specifically, they measured total bandwidth usage of running tasks, BW_{Occ}. Based on the peak bandwidth, BW_{Peak}, estimated before, a scheduler calculates available bandwidth BW_{Avail}. It selects a next task that has memory bandwidth usage similar to the BW_{Avail}.

However, these methods are not applicable with the hybrid memory architecture where both DRAM and NVRAM are used as a part of main memory. Equation (5) is a decomposition of (2) according to the memory access type when the target memory is NVRAM. As we describe in Section II-A, access latency to the NVRAM is vary according to the request type.

TABLE II. SYMBOLS TO DESCRIBE (3) TO (5).

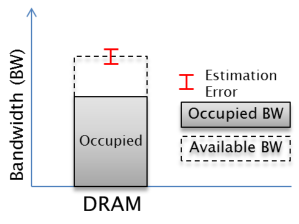
Symbol	Description
BW_D	occupied DRAM bandwidth
BW_{NV}	occupied NVRAM bandwidth
$BW(R)$	occupied memory bandwidth by read
$BW(W)$	occupied memory bandwidth by write
$BW_{D,Avail}$	available DRAM bandwidth
$BW_{NV,Avail}$	available NVRAM bandwidth
$BW_{D,Occ}$	occupied DRAM bandwidth
$BW_{NV,Occ}$	occupied NVRAM bandwidth
$BW_{D,Peak}$	peak DRAM bandwidth
$BW_{NV,Peak}$	peak NVRAM bandwidth
$BW_{D,Peak}(R)$	peak DRAM bandwidth by read
$BW_{D,Peak}(W)$	peak DRAM bandwidth by write
$BW_{NV,Peak}(R)$	peak NVRAM bandwidth by read
$BW_{NV,Peak}(W)$	peak NVRAM bandwidth by write

$$BW_{Avail} = BW_{Peak} - BW_{Occ} \tag{2}$$
$$BW_{Avail} = (BW_{D,Avail}, BW_{NV,Avail}) \tag{3}$$
$$BW_{D,Avail} = BW_{D,Peak} - BW_{D,Occ} \tag{4}$$
$$\begin{aligned} BW_{NV,Avail} &= BW_{NV,Peak} - BW_{NV,Occ} \\ &= (BW_{NV,Peak}(R) \\ &+ BW_{NV,Peak}(W)) \\ &- (BW_{NV,Occ}(R) + BW_{NV,Occ}(W)) \end{aligned} \tag{5}$$

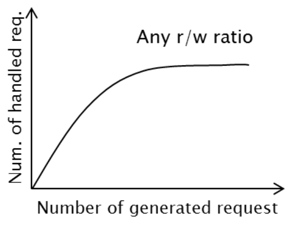
An NVRAM access request occupies target memory rank for a specified time according to the type of the request. Different latencies of each memory request type change the number of memory access requests handled during a unit time. Even a read operation consumes more time than the DRAM's, it is hard to represent the bandwidth consumption of a task by the number of memory access requests. It should be consider the target memory medium and the proportion of read and write operation to the number of memory transactions.

Fig. 3 illustrates the effect of different latencies to the bandwidth usage estimation. With DRAM main memory, occupied bandwidth of running tasks can be represented by the number of memory transactions. With hybrid main memory of DRAM and NVRAM, memory access requests can be classified into three categories based on access latencies: DRAM access, NVRAM read, and NVRAM write.

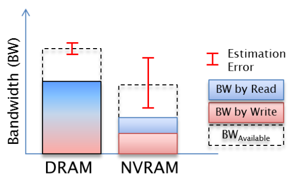
The problem is that estimated bandwidth of running tasks changes over the proportion of write to overall accesses. It makes the number of memory transactions hard to reflect the bandwidth usage of the task. It results in wrong decision about selecting next task satisfying decision criteria. Therefore, bandwidth estimation for NVRAM should reflect type of accesses by differentiating $BW_{NV}(W)$ and $BW_{NV}(R)$ as shown in (5).



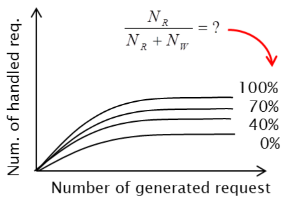
(a) DRAM-only main memory



(b) expected bandwidth of DRAM-only main memory



(c) Hybrid main memory



(d) expected bandwidth of Hybrid main memory

Figure 3: Number of memory transactions handled during a unit time. NVRAM bandwidth usage changes according to the ratio of read and write.

V. HYBRID MAIN MEMORY-AWARE TASK SCHEDULING (HMMSCHED)

A. Basic Design

The primary design goal of the HMMSched is to accelerate task performance by co-scheduling tasks consuming complementary amount of memory bandwidth on different type of memory. To achieve this design goal, HMMSched 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. During the running time of each task, hardware PMU collects memory access information for each task. Collected information includes type of target memory medium, memory access type, and the access counts. The HMMSched selects next task on the basis of the proposed policy regarding the collected statistics during the last scheduled period.

1) Per-task memory access monitoring: With HMMSched, memory bandwidth usage is an important criterion to choose next task to be run on an idle core. A software PMU, described in Section VI-A collects segmentalized memory access statistics for each task based on the access types and the target memory type. We have three criteria to classify memory accesses: target memory type, access type, and distance from the core where the target task runs to the target memory.

Target memory type is our primary consideration for classifying memory accesses. We targeted hybrid main memory that DRAM and NVRAM are located at the same level in

TABLE III. VARIABLES TO CALCULATE EFFECTIVE BANDWIDTH IN THE HYBRID MAIN MEMORY ENVIRONMENT

Variable	Description
$EBW(T)$	Effective bandwidth of a task T
EBW_D	Effective bandwidth for DRAM
EBW_{NV}	Effective bandwidth for NVRAM
BW_D	Conventional bandwidth for DRAM
N_{Req}	Number of memory access transactions
T_D	Memory access latency of DRAM
N_R	Number of read transactions to the NVRAM
N_W	Number of write transactions to the NVRAM
γ	NVRAM read latency / DRAM access latency
δ	NVRAM write latency / DRAM access latency

the memory hierarchy. However, each medium has different access latencies. If a task accesses data located in NVRAM, it can access less number of data within a unit time compared with when accessing data in DRAM. It makes the task make progress in the CPU. Therefore, we differentiate NVRAM accesses from DRAM accesses to find tasks having more potentials of making progress. Because the latency of write to NVRAM is much longer than read from the NVRAM, we also differentiate NVRAM writes from NVRAM reads based on the same criteria.

Generally, the DRAM bandwidth usage of a task is measured as the number of memory transactions during a unit time. In case of NVRAM, memory access latencies are different from each other according to the target media and access type. It results in that the number of memory access does not reflect memory bandwidth usage of a task. We therefore use a new metric, Effective Memory Bandwidth (EBW) to translate NVRAM's bandwidth usage and bandwidth usage of remote node memory into the number of DRAM transactions. Table III shows items that the per-task software PMU collects and translates. We can calculate effective bandwidth using (6).

$$\begin{aligned} EBW &= EBW_D + EBW_{NV} && (6)\\ EBW_D &= BW_D \\ &\simeq N_{Req} \times T_D \\ EBW_{NV} &\simeq \gamma \times N_R \times T_D + \delta \times N_W \times T_D && (7) \end{aligned}$$

When a task is about to be scheduled, the software PMU starts monitoring of memory accesses generated from the task. If the task consumes all allocated time slices or is preempted by other task, collected values are stored in the kernel memory. Software PMU then translates measured data to estimated values having same unit using (7). HMMSched use latest EBW value of each task to select next task to be scheduled on an idle core.

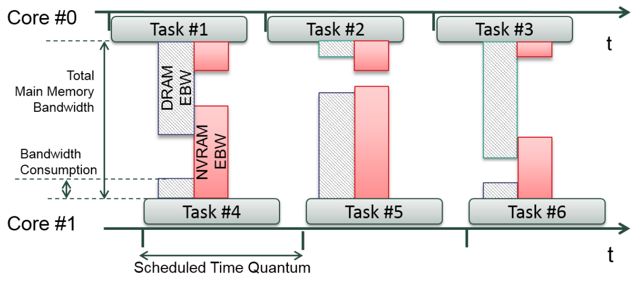


Figure 4: Basic idea of HMMSched. A task consuming complementary EBW to different medium is selected to be scheduled

2) Choosing a next task to be scheduled: Whenever a core becomes idle, HMMSched selects a task from scheduling candidate tasks based on a converted EBW.

As a preliminary work for choosing a next task to be scheduled, HMMSched arranges all schedulable tasks in order of EBW. HMMSched then classify them into two categories based on their latest EBW: latency-sensitive and bandwidth-sensitive. Latency-sensitive tasks consume less amount of memory bandwidth. Bandwidth-sensitive tasks spend more time to access both types of memory.

HMMSched use EBW as a criterion to classify tasks according to the bandwidth usage. Before classification, all tasks are arranged in order of EBW calculated by the (6). Let $Task_i$ indicates a task having *i*th lower EBW and $EBW(Task_i)$ indicates estimated effective bandwidth of $Task_i$. We then calculate $TotalEBW = \sum_{i=1}^{n} EBW(Task_i)$, where *n* is the number of all schedulable tasks. Tasks occupying some of total memory bandwidth greater than a predefined threshold α are classified as latency-sensitive. In other words, *K* tasks satisfying $\sum_{i=1}^{K} EBW(Task_i) < \alpha TotalEBW$ are classified into latency-sensitive tasks. Others are classified as bandwidth-sensitive. Here α is a parameter $0 \leq \alpha \leq 1$, where lower α indicates a stronger threshold. In order to prevent bandwidth-sensitive tasks from being subjected to starvation, α reflects total execution time of each category. HMMSched selects latency-sensitive task first as long as the total execution time of all latency-sensitive tasks is less than a tenth of the total execution time of bandwidth-sensitive tasks.

We applied different management scheme to each categories for ease of candidate selection. In the latency-sensitive category, all tasks are arranged in ascending order of own $EBW(T)$. In the bandwidth-sensitive category, each task belongs to two management lists; DRAM-intensive and NVRAM-intensive. DRAM-intensive list sorts all tasks in ascending order of their $EBW_D(T)$. NVRAM-intensive list arranges all tasks in ascending order of $EBW_{NV}(T)$.

When HMMSched tries to select a task to be scheduled next, the primary rule is that the scheduler always chooses

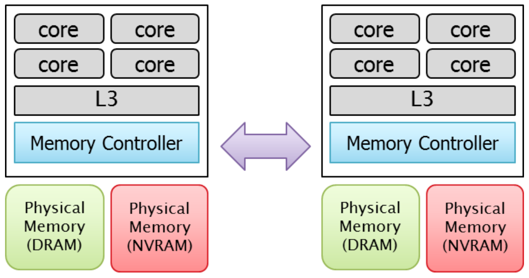


Figure 5: Target hardware architecture of HMMSched. DRAM and NVRAM share same memory controller on each local node of the NUMA system.

a latency-sensitive task prior to selecting a bandwidth-sensitive task. A task consumed lowest EBW has highest priority if there are multiple latency-sensitive tasks.

If there is no latency-sensitive task, HMMSched picks a task from the bandwidth-sensitive category based on the following order:

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

Fig. 4 illustrates this policy. It first makes the scheduler co-schedule intensively accessing different type of memory together. It then co-schedules tasks having complementary access intensities to the same type of memory.

VI. Evaluation

In this section, we evaluate the performance of HMMSched described in the previous section. We used two experiment systems for evaluation with Intel I7 960 processor running at 3.2GHz, 6GB of RAM, 2GB per DIMM. The operating system is Linux 2.6.38.2. Simulated NVRAM and PMU supporting per-task memory access monitoring is described in the next section. In this paper, we targeted a manycore system combined with hybrid main memory of DRAM and NVRAM, shown in Fig. 5. DRAM and NVRAM in a local node share same memory controller.

A. Modeling customized PMU and NVRAM

To evaluate our idea, we need two items: NVRAM main memory module and the hardware performance monitoring unit (PMU), which counts per-task memory accesses. Unfortunately, there is no released NVRAM module for main memory and commercial CPUs with PMU counting per-task memory accesses. We use Intel Pin [32] to perform a detailed simulation of the system's main memory augmented with NVRAM and the per-task memory access monitoring support that HMMSched requires. The memory simulator adds designated delay between a read or a write to the NVRAM and the operations that follow, allowing it to accurately model the longer read and write times of NVRAM. For NVRAM, we use the performance model of PRAM from [21], which gives an access latency ratio shown in Table I.

As a software PMU, the simulator monitors main memory accesses for each task and counts the number of following requests separately: the number of DRAM accesses, the number of NVRAM reads, and the number of NVRAM writes. Fig. 6(a) illustrates a simulation environment where each task runs with own per-task simulation module. Each simulation module acts like a PMU for single task. To estimate the number of memory accesses, each simulation module has own cache hierarchy model. Each simulation module shares its last-level cache model with other modules when their target tasks run on the cores sharing same last-level cache. Collected memory access information is transferred to the task scheduler via shared memory.

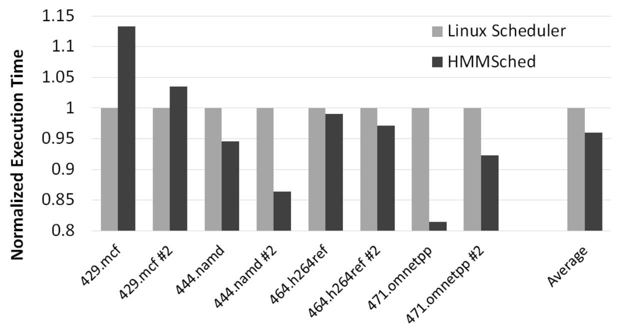
Note that collected information from the simulation module should reflect actual memory access information generated by the combination of the target task and the attached simulation module. It is because of the fact that the simulated PMU monitors memory accesses of the target task while the scheduler schedules the target task running with the simulation module. We calibrated the simulated PMU by adjusting miss rates of the shared last level cache. As a result, the software PMU reflects the memory access characteristics generated from the combination of the target task and the simulation module.

B. Workloads

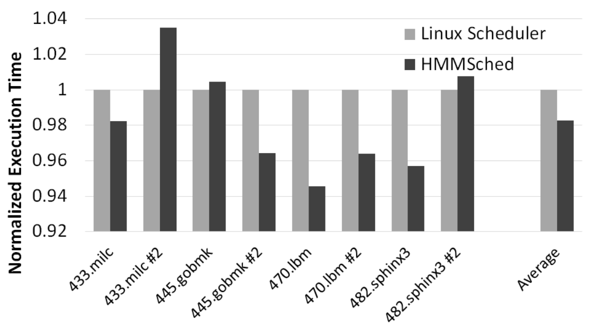
We use the SPEC CPU2006 benchmarks [33] for evaluation. We compiled each benchmark using gcc 4.4.3 with default optimizations of the benchmark, and executes on Linux 2.6.38.2 kernel. Two workload set is used to evaluate the effectiveness of HMMSched. Each workload set consists of four applications, where each application has two instances. In the first set, half of the applications are memory-intensive tasks while others are CPU-intensive tasks. Second set has six memory-intensive tasks and two CPU-intensive tasks. We configured them to check the effect of prioritization to the latency-sensitive tasks and the effect of various memory access intensities.

C. Preliminary Result

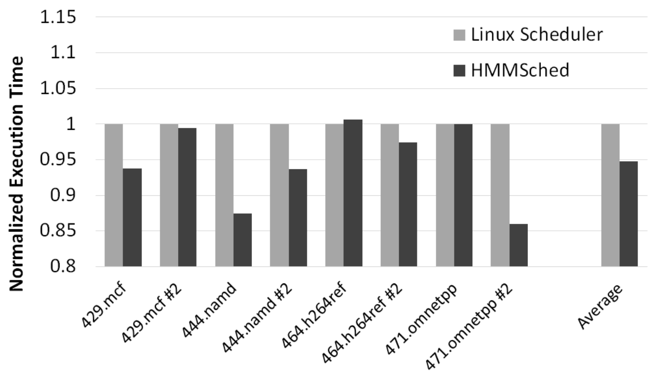
Fig. 7 shows preliminary experimental results of the HMMSched compared with the default Linux scheduler. Y-axis of the graphs indicates normalized execution time of the benchmarks. In this paper, we add experiment results with local node sharing same memory controller. We leave experiments related to NUMA architecture as our further work. Here we used following NVRAM access latency specifications. First set is DRAM : NVRAM(read) : NVRAM(write) = 1 : 1 : 3, which is similar to the specification of STT-RAM shown in Table I. Second set is DRAM : NVRAM(read) : NVRAM(write) = 1 : 3 : 8, which is also similar to the specification of PRAM in



(a) Workloads mix #1 with STT-RAM-like access latency specification

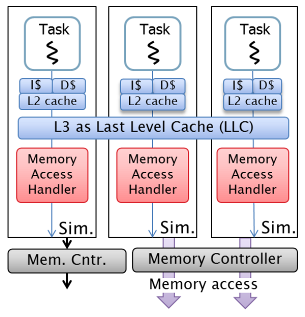


(b) Workloads mix #2 with STT-RAM-like access latency specification

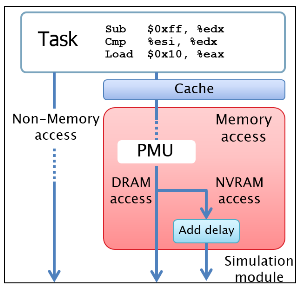


(c) Workloads mix #1 with PRAM-like read/write access latency specification

Figure 7: Elapsed time of selected benchmark sets under HMMSched compared with default Linux scheduler.



(a) For multiple tasks



(b) Internal flow of a simulation module

Figure 6: Overall simulation environment running multiple tasks. Each task runs with per-task simulation module.

Table I. We also composed two benchmark set to analyze the effect of the priority change of the scheduler when the latency-sensitive tasks are run with bandwidth-sensitive tasks. Benchmark mix #1 consists of four memory-intensive tasks and four CPU-intensive tasks. Benchmark mix #2 contains six memory-intensive tasks and two CPU-intensive tasks. On each experiment, all benchmarks of the target set run concurrently.

As we can see in the two experiment results, prioritizing latency-sensitive tasks and utilizing EBW on co-scheduling bandwidth-sensitive tasks can effectively reduce completion time of them. An instance of 471.omnetpp completed by 19% faster than under the default Linux scheduler. In addition, we can reduce average running time of all workloads by 5% under HMMSched even though it hurts the performance of 429.mcf in Fig. 7(a) With the workloads mix #2, we can also see improvements with smaller performance degradation of 433.milc #2. If we target PRAM as an alternative of main memory, our scheduling policy affects more on concurrently running application performance. Figure 7(c) shows experiment result with PRAM specification for NVRAM part of main memory. Because of the long read access latency of PRAM compared with DRAM, co-scheduling bandwidth-sensitive tasks utilizing EBW shows better performance than with the default linux scheduler. With STT-RAM specification, tasks generating more NVRAM write get performance penalty. With PRAM specification, however, read latency also affects the effective bandwidth usage of tasks. It leads to leveling-off of each task's EBW usage so that they divide up the penalty from scheduling delay. Consequently, though several tasks are slightly hurt their performance, we can get more balanced performance gain.

VII. Conclusion and Further Work

As manycore increases the need for larger memory capacity, contention on memory has become a key issue. Although hybrid main memory of DRAM and NVRAM can enlarge memory capacity with high energy efficiency, hybrid main memory-agnostic contention-aware scheduling degrades task performance. Different access latencies to the NVRAM compared with DRAM prevent bandwidth consumption of each task hard to measure. In this paper, we propose HMMSched, a novel task co-scheduling method for many-core and a hybrid main memory of DRAM and NVRAM. We define effective bandwidth to reflect different access latencies according to the access type to the NVRAM. We then prioritize CPU-intensive tasks to make more progress in the CPU. Tasks consuming complementary EBW on different memory are alternately selected to be co-scheduled with running tasks. Consequently, HMMSched co-schedule tasks consuming different memory type with complementary access intensity, thereby reducing delay from contention on the memory. The experimental results show that our scheme outperforms default Linux scheduler by up to 19% in terms of time efficiency.

As further works, we will add hybrid main memory management features proposed in [14] to analyze effect of data migration policy on the HMMSched scheduling. We will also add task co-scheduling policy when the target memory hierarchy includes NUMA architecture. Finally, we will carefully analyze the effect of cache hit ratio according to the ratio of read and write to the NVRAM main memory.

References

[1] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture. New York, NY, USA: ACM, 2009, pp. 2–13.

[2] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture. New York, NY, USA: ACM, 2009, pp. 14–23.

[3] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: a hybrid pram and dram main memory system," in DAC '09: Proceedings of the 46th Annual Design Automation Conference. New York, NY, USA: ACM, 2009, pp. 664–469.

[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture. New York, NY, USA: ACM, 2009, pp. 24–33.

[5] E. Koukis and N. Koziris, "Memory bandwidth aware scheduling for smp cluster nodes," in PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing. Washington, DC, USA: IEEE Computer Society, 2005, pp. 187–196.

[6] ——, "Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of smps," in Proceedings of the 12th International Conference on Parallel and Distributed Systems - Volume 1, ser. ICPADS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 345–354.

[7] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a cmp of multi-threaded processors," in Proceedings of the 20th international conference on Parallel and distributed processing, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 10–19.

[8] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques. New York, NY, USA: ACM, 2010, pp. 237–248.

[9] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems. New York, NY, USA: ACM, 2010, pp. 129–142.

[10] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in Proceedings of the 19th international conference on Parallel architectures and compilation techniques, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 557–558.

[11] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps," in Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 277–286.

[12] M. Bhadauria and S. A. McKee, "An approach to resource-aware co-scheduling for cmps," in ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing. New York, NY, USA: ACM, 2010, pp. 189–199.

[13] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for nvm+dram hybrid main memory," in Proceedings of the 12th conference on Hot topics in operating systems, ser. HotOS'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 14–18.

[14] Y. Park, S. K. Park, and K. H. Park, "Linux kernel support to exploit phase change memory," in Linux Symposium, 2010, pp. 217–224.

[15] S. Lee, H. Bahn, and S. Noh, "Characterizing memory write references for efficient management of hybrid pcm and dram memory," in Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on, july 2011, pp. 168 –175.

[16] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mossé, "Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems," ACM Trans. Archit. Code Optim., vol. 8, no. 4, Jan. 2012, pp. 1–21.

[17] K. H. Park, S. K. Park, W. Hwang, H. Seok, D.-J. Shin, and K.-W. Park, “Resource management of manycores with a hierarchical and a hybrid main memory for mn-mate cloud node,” in Services (SERVICES), 2012 IEEE Eighth World Congress on. IEEE, 2012, pp. 301–308.

[18] K. H. Park, Y. Park, W. Hwang, and K.-W. Park, “Mn-mate: Resource management of manycores with dram and nonvolatile memories,” in High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on. IEEE, 2010, pp. 24–34.

[19] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, “Overview of candidate device technologies for storage-class memory,” IBM J. Res. Dev., vol. 52, July 2008, pp. 449–464.

[20] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, “Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing,” in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[21] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories,” in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ser. ASPLOS ’11. New York, NY, USA: ACM, 2011, pp. 105–118.

[22] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, “Phase-change random access memory: A scalable technology,” IBM Journal of Research and Development, vol. 52, no. 4.5, july 2008, pp. 465 –479.

[23] S. Eilert, M. Leinwander, and G. Crisenza, “Phase change memory: A new memory enables new memory usage models,” in Memory Workshop, 2009. IMW ’09. IEEE International, may 2009, pp. 1–2.

[24] H. Li and Y. Chen, “An overview of non-volatile memory technology and the implication for tools and architectures,” in Design, Automation Test in Europe Conference Exhibition, 2009. DATE ’09., april 2009, pp. 731 –736.

[25] B. Dieny, R. Sousa, G. Prenat, and U. Ebels, “Spin-dependent phenomena and their implementation in spintronic devices,” in VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on, april 2008, pp. 70 –71.

[26] M. Saxena and M. M. Swift, “Flashvm: virtual memory management on flash,” in Proceedings of the 2010 USENIX conference on USENIX annual technical conference. USENIX Association, 2010, pp. 14–27.

[27] A. Badam and V. S. Pai, “Ssdalloc: hybrid ssd/ram memory management made easy,” in Proceedings of the 8th USENIX conference on Networked systems design and implementation, ser. NSDI’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 16–29.

[28] A. Merkel and F. Bellosa, “Memory-aware scheduling for energy efficiency on multicore processors,” in HotPower’08: Proceedings of the 2008 conference on Power aware computing and systems. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–5.

[29] M. DeVuyst, R. Kumar, and D. M. Tullsen, “Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors,” in Proceedings of the 20th international conference on Parallel and distributed processing, ser. IPDPS’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 140–149.

[30] L. E. Ramos, E. Gorbatov, and R. Bianchini, “Page placement in hybrid memory systems,” in Proceedings of the international conference on Supercomputing, ser. ICS ’11. New York, NY, USA: ACM, 2011, pp. 85–95.

[31] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, “Thread cluster memory scheduling: Exploiting differences in memory access behavior,” in Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO ’43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 65–76.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” SIGPLAN Not., vol. 40, June 2005, pp. 190–200.

[33] “SPEC CPU 2006 Benchmark Suite.” [Online]. Available: http://www.spec.org/cpu2006