Master’s Thesis

Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems

Seongdae Yu

Department of Computer Science and Engineering

Graduate School of UNIST

2018

Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems

Seongdae Yu

Department of Computer Science and Engineering

Graduate School of UNIST

Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Seongdae Yu

June 15, 2018
Approved by

[illegible]

Advisor
Woongki Baek

Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems

Seongdae Yu

This certifies that the thesis of Seongdae Yu is approved.

6. 15. 2018

signature
[illegible]

Advisor : Woongki Baek

signature
[illegible]

Beomseok Nam : Committee Member #1

signature
[illegible]

Young-ri Choi : Committee Member #2

Contents

List of Figures

List of Tables

Abstract

Heterogeneous memory systems are composed of several types of memory, and are used in various computing domains. Each memory node in heterogeneous memory systems has different characteristics and performances. A particularly significant difference can be found in access latency and memory bandwidth. Therefore, the heterogeneity between memories must be considered to utilize the performance of a heterogeneous memory system. However, most of the previous works did not consider the bandwidth difference of the memory nodes constituting a heterogeneous memory system.

The present work proposes bandwidth-aware memory placement and migration policies to solve the problem caused by the bandwidth difference of the memory nodes in a heterogeneous memory system. We implement three bandwidth-aware memory placement policies and one bandwidth-aware migration policy on the Linux kernel, then quantitatively experiment on and evaluate them in real systems. In addition, we prove that our proposed bandwidth-aware memory placement and migration policies can achieve a higher performance compared to conventional memory placement and migration policies that do not consider the bandwidth differences between heterogeneous memory nodes.

Chapter 1. Introduction

A heterogeneous memory system consists of several kinds of memory nodes, and each node has different characteristics and performances. Many advantages can be obtained by the existence of various kinds of memory nodes in the system. For example, a heterogeneous memory system consisting of DRAM and NVM can use a DRAM with low access latency and high bandwidth for high-performance tasks, and can use NVM for tasks, such as logging, where persistence is important [1, 2].

This work proposes bandwidth-aware placement and migration policies that consider the bandwidth difference between each memory node in heterogeneous memory systems. These bandwidth-aware policies aim to maximize the bandwidth of heterogeneous memory systems, which can lead to significant performance improvements in bandwidth-intensive applications commonly used in HPC. We designed and implemented three bandwidth-aware placement policies and one migration policy. We then quantitatively evaluated the policies in the real system. The specific contribution of this work is as follows:

- We propose bandwidth-aware memory placement and migration policies. These policies allocate and migrate pages considering the bandwidth difference of each node constituting a heterogeneous memory system.
- We implement and experiment on the proposed bandwidth-aware memory placement and migration policies. The implementation is done by modifying the existing Linux kernel code and adding a new code.
- The performance of the proposed bandwidth-aware memory placement and migration policies is quantitatively evaluated in real systems. We prove herein that the bandwidth-aware memory placement and migration policies considering the heterogeneity of the heterogeneous memory system can have a large performance improvement compared to the existing policy that does not consider the heterogeneity.

The remainder of this paper is organized as follows: Section 2 provides the background knowledge needed to understand this paper and the motivation for this work; Section 3 presents the algorithms and implementation for the bandwidth-aware memory placement and migration policies; Section 4 describes the experimental environment and tools; Section 5 explains the experimental results of the proposed policies and its causes; Section 6 introduces related work and explains the differences from this paper; and Section 7 describes the conclusions.

Chapter 2. Background and Motivation

2.1 Physical Memory Description

Linux is used on a variety of hardware. It uses physical memory management methods that can be used regardless of the architecture [3]. Figure 2.1 shows the organization of a Linux physical memory that is hierarchically constituted. The top layer is a *memory node*, which is the set of DIMMs connected to the same CPU. A memory node has several `zones`, and a node in the `x86` architecture consists of three zones. The first is the `ZONE_DMA`, which occupies 0 MB to 16 MB of space. The second is the `ZONE_NORMAL`, which occupies 16 MB to 896 MB of space. The last zone is the `ZONE_HIGHMEM`, which occupies from 896 MB to the last space of memory. Each zone is made up of several *pages*, which are the basic units of memory placement and migration.

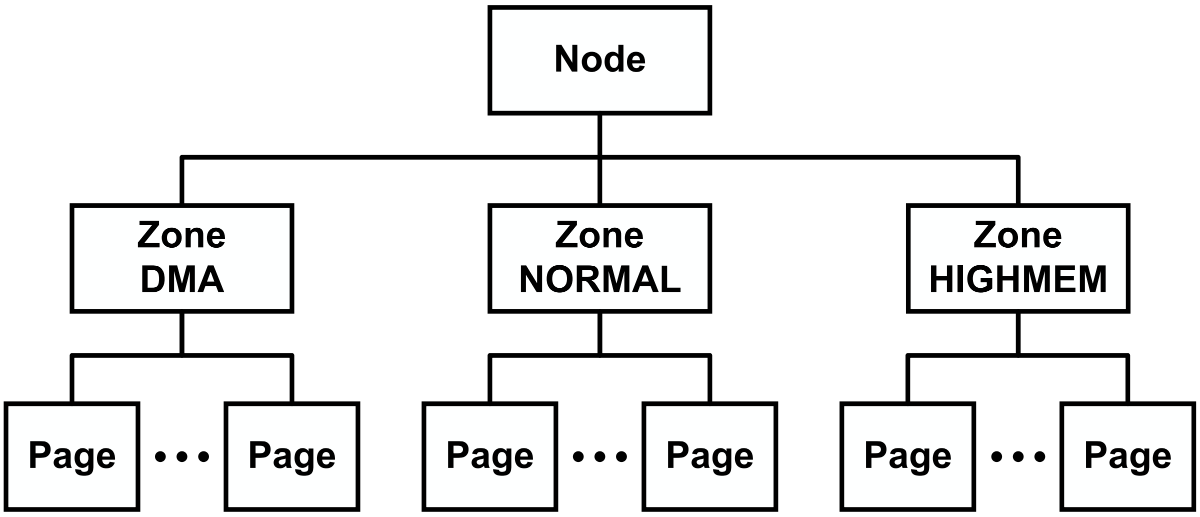

Figure 2.1: Physical memory description

2.2 Conventional Memory Placement and Migration Policies

A system with multiple CPU sockets and DIMMs in one system is called a non-uniform memory access (NUMA) system. Each CPU socket is locally connected to a memory node consisting of one or more DIMMs. Moreover, each CPU socket is remotely connected to other CPU sockets and memory nodes through the interconnection network. Accessing the CPU's core locally connected memory is called *local* access. Meanwhile, accessing the CPU's core remotely connected memory is called *remote* access. The latency of remote access is higher than that of local access because remote access has to go through the interconnection network [4].

There are two conventional memory placement policies in NUMA systems. The first is *local* policy. Local policy is used as Linux's default policy. It allocates pages to a memory node connected locally to the CPU socket on which the current task is running. Figure 2.2 shows an example of local policy.

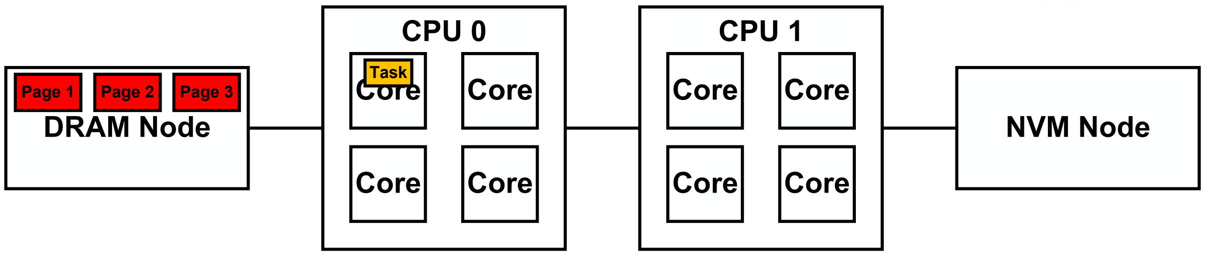

Figure 2.2: Example of local policy

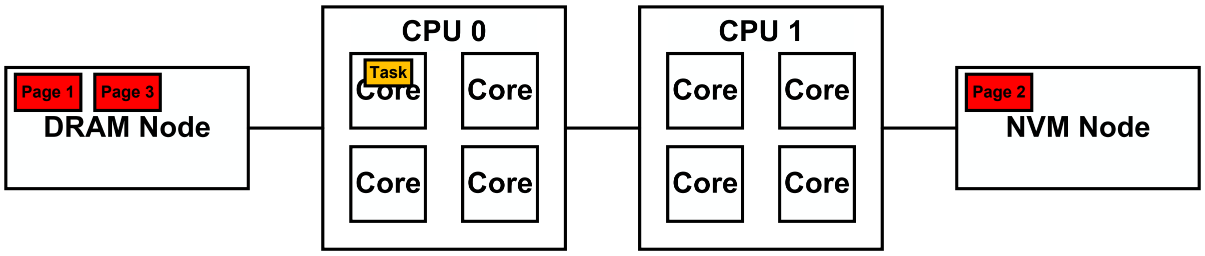

Figure 2.3: Example of local policy

When using local policy, the core in CPU socket 0 allocates pages only to locally connected memory node 0. The second is the *interleave* policy. The interleave policy allocates pages in a round-robin manner. Figure 2.3 shows an example of interleave policy. When using interleave policy, the page is first allocated to the local memory node of the CPU socket on which the current task is executed. The page is then alternately allocated to each memory node.

Local access is to access the pages allocated to the local memory of the CPU socket, where the task is currently located. When local policy is being used, the pages are allocated only to the local memory, and local access is made unless the task migrates to the core of another CPU socket. Local access has a lower access latency than remote access; hence, it can be beneficial in performance. However, if tasks are flocked to a single CPU socket, load balancing will fail because they do not use other memory nodes. Remote access is to access the page allocated to the remote memory node. The remote access latency is higher than that of the local access because the data must pass through the interconnection network. Interleave policy allocates pages to each memory node in a round-robin manner; hence, both local and remote access occur. Therefore, the interleave policy shows a performance lower than that of the local policy in access latency, but it can benefit from load balancing by evenly using each memory node.

In the NUMA system, automatic NUMA balancing (ANB) can migrate pages that have already been allocated to a memory node [5]. When task accesses a page configured as the *NUMA page* [5], a *NUMA hinting page fault* is set. Periodically, ANB checks the NUMA hinting page fault on each NUMA page and migrates the page where NUMA hinting page fault is set to the local memory node of a task for access locality if a task frequently causes remote access to the page.

2.3 Non-volatile Memory

Non-volatile memory is a newly emerging technology with various characteristics, such as persistence, byte addressability, and high density. NVMs, such as phase-change memory (PCM) [6], are

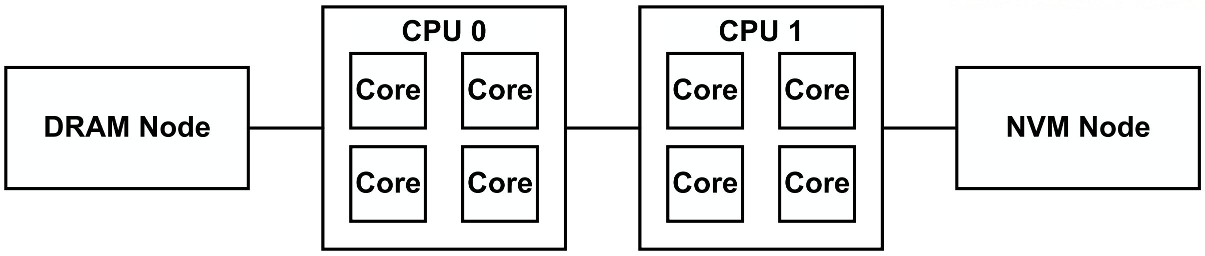

Figure 2.4: System architecture with heterogeneous memory

expected to be 100 times faster than SSD and two to five times slower than DRAM [2]. In this work, a heterogeneous memory system consisting of DRAM and NVM is used as a target architecture. Figure 2.4 shows the architecture, where each memory node consists of the same type of DIMM. Previous work [1] has constructed nodes in this manner to prevent the potential performance overheads of the block device interface that may occur by configuring nodes with different types of DIMMs.

2.4 Need for Bandwidth-aware Memory Placement and Migration Policies

The existing NUMA system does not consider the heterogeneity of each memory node to consist of different kinds of DIMMs. The local and interleave policies treat each node equally without considering the heterogeneity of each memory node. As a result, system resources may not be fully utilized when local and interleave policies are used in a heterogeneous memory system with a large performance gap between memory nodes.

Let us assume that a heterogeneous memory system consists of one DRAM node and one NVM node. The DRAM node bandwidth in this system is represented by B_{DRAM}, while the NVM node is represented by B_{NVM}. Let S be the total amount of data required for the application, p be the ratio of the data allocated to the DRAM node, and $1 - p$ be the data rate allocated to the NVM node. The time to transfer the data from the DRAM to the CPU (t_{DRAM}) is expressed in Equation 2.1, and the time (t_{NVM}) to transfer the data from NVM to the CPU is expressed in Equation 2.2.

$$t_{DRAM} = \frac{p \cdot S}{B_{DRAM}} \tag{2.1}$$

$$t_{NVM} = \frac{(1-p) \cdot S}{B_{NVM}} \tag{2.2}$$

In a bandwidth-intensive application, which is the main target of this work, data transfer takes up most of the execution time. Therefore, the total execution time (t_{TOT}) is determined to be a larger value of t_{DRAM} and t_{NVM}.

$$t_{TOT} = \max(t_{DRAM}, t_{NVM}) \tag{2.3}$$

The values of t_{DRAM} and t_{NVM} must be equal to minimize the total execution time. To do this, the bandwidth of each node and the amount of allocated data should be proportional. As a result, the ratio of B_{DRAM} to B_{NVM} should be equal to the ratio of p to $1 - p$. Therefore, the optimal allocation ratio (p_{OPT}) to be allocated to the DRAM node is calculated as follows:

$$p_{OPT} = \frac{B_{DRAM}}{B_{DRAM} + B_{NVM}} \tag{2.4}$$

The bandwidth-aware memory placement and migration policies proposed in this work allocate memory at the optimal rate considering the bandwidth difference between each node of the heterogeneous memory system. Moreover, the ratio is maintained even if migration occurs, allowing the system to make the most of the aggregated bandwidth. In contrast, conventional policies are bandwidth-oblivious memory policies that perform memory placement and migration without considering the bandwidth differences between the memory nodes. The amount of memory allocated to each node is not proportional to the bandwidth because of this, and migration can make this worse. This can cause a serious bottleneck at a particular node, which can significantly degrade the overall performance of the system.

Chapter 3. Design and Implementation

This section describes the design and the implementation of the bandwidth-aware memory placement and migration policies. Section 3.1 describes the concept of the memory clusters added to manage the heterogeneous memory systems. Sections 3.2 and 3.3 describe the algorithm of each bandwidth-aware memory placement and migration policy. Section 3.4 describes the implementation of the bandwidth-aware memory placement and migration policies. Section 3.5 discusses some design issues.

3.1 Heterogeneous Memory Description

We have added the memory cluster concept to efficiently manage the memory organization of the heterogeneous memory system. A memory cluster consists of one or more memory nodes, all of which are of the same type. A structure, called *memory cluster*, is used in the implementation. This structure has a pointer that points to each node belonging to the cluster. The structure also stores cluster attribute information, such as cluster bandwidth.

3.2 BW-aware Memory Placement Policies

This subsection describes the pseudocode of each bandwidth-aware memory placement policy. The pseudocode assumes a heterogeneous memory system to consist of one DRAM cluster and one NVM cluster. Furthermore, the bandwidth ratios of the DRAM and NVM clusters are expressed as D and N, respectively. For example, if the bandwidth of the DRAM cluster is 4GB/s, and that of the NVM cluster is 2GB/s, D becomes 2, and N becomes 1. A variable, called `aCount`, is added to the `task_structure` that stores task information to store the allocation history of the task. The bandwidth-aware memory placement policies use this history information to allocate pages according to the optimal allocation ratio.

3.2.1 The `BW-INTERLEAVE` Policy

The bandwidth-aware memory placement interleave (`BW-INTERLEAVE`) policy allocates pages in a round-robin manner considering each memory cluster bandwidth. Algorithm 1 is the pseudocode of the `BW-INTERLEAVE` policy. The page allocation proceeds in the order of (1) determining the cluster to which the page is to be allocated (Line 8), (2) determining the node to which the page is to be allocated in a round-robin manner within the cluster (Line 9), and (3) allocating the page (Line 10).

The `BW-INTERLEAVE` policy determines the memory cluster based on the task's `aCount` value. The DRAM cluster is selected if `aCount` is less than D, and the NVM cluster is selected if it is equal to or greater than D (Lines 3–6). `aCount` is then incremented by 1 and reset to 0 when it is $D + N$ (Line 5). Therefore, the `BW-INTERLEAVE` policy allocates memory to the DRAM cluster for D times, then allocates memory to the NVM cluster for N times. In other words, the page is proportionally

Algorithm 1 `BW-INTERLEAVE` policy

```
procedure getClusterInterleave(task)
    cluster ← DRAM
    if task.aCount ≥ $D$ then
        cluster ← NVM
    task.aCount ← (task.aCount + 1) % ($D + N$)
    return cluster
procedure bwAwareInterleave(task)
    cluster ← getClusterInterleave(task)
    node ← getNodeInterleave(task, cluster)
    page ← allocPage(task, node)
    return page
```

Algorithm 2 `BW-RANDOM` policy

```
procedure getClusterRandom(task)
    cluster ← DRAM
    $r$ ← getRandomInt() % ($D + N$)
    if $r \geq D$ then
        cluster ← NVM
    return cluster
procedure bwAwareRandom(task)
    cluster ← getClusterRandom(task)
    node ← getNodeRandom(task, cluster)
    page ← allocPage(task, node)
    return page
```

allocated to the bandwidth ratio, and the allocation ratio is equal to the optimal allocation ratio of Equation 2.4.

3.2.2 The `BW-RANDOM` Policy

The bandwidth-aware random memory placement (`BW-RANDOM`) policy selects a memory cluster to allocate pages using the probabilistic manner. Algorithm 2 shows the pseudocode of the `BW-RANDOM` policy. As with the bandwidth-aware interleave, the `BW-RANDOM` policy selects a memory cluster, to which a page is allocated, and a node in the cluster, then allocates a page (Lines 8–10). The `BW-RANDOM` policy generates a random variable and performs a modulation operation on the value of $D + N$ (Line 3). The DRAM cluster is selected as the cluster to which the page is to be allocated if the result of the modulation operation is less than D. The page is allocated to the NVM cluster if the result is equal to or greater than D (Lines 4–5). Therefore, the page is allocated to the DRAM cluster with the probability of $\frac{D}{D+N}$, and the page is allocated to the NVM cluster with the probability of $\frac{N}{D+N}$.

3.2.3 The `BW-LOCAL` Policy

The bandwidth-aware local memory placement (`BW-LOCAL`) policy preferentially allocate pages to the local node while maintaining the optimal allocation ratio. Algorithm 3 shows the pseudocode of the `BW-LOCAL` policy. The `BW-LOCAL` policy considers which cluster the current task is in. Page is allocated

Algorithm 3 `BW-LOCAL` policy

```
 1: procedure getClusterLocal(task)
 2:     currCluster ← task.cluster
 3:     aCount ← task.aCount[currCluster]
 4:     cluster ← DRAM
 5:     if currCluster = DRAM then
 6:         if aCount ≥ $D$ then
 7:             cluster ← NVM
 8:     else
 9:         if aCount < $N$ then
10:             cluster ← NVM
11:     task.aCount[currCluster] ← (aCount + 1) % ($D$ + $N$)
12:     return cluster
13: procedure bwAwareLocal(task)
14:     cluster ← getClusterLocal(task)
15:     node ← getNodeLocal(task, cluster)
16:     page ← allocPage(task, node)
17:     return page
```

to the NVM cluster if the cluster in which the task is currently located is a DRAM cluster and the value of `aCount` of the cluster is greater than or equal to D (Lines 5–7). And the page is allocated to the NVM cluster if the cluster, where the current task is located, is an NVM cluster and the value of `aCount` of the cluster is less than N (Lines 8–10). In other cases, page is allocataed to the DRAM cluster. The value of `aCount` corresponding to the cluster in which the current task is located is then incremented by 1, and `aCount` is reset to zero if the value is equal to $D + N$. Therefore, when a task is located in a DRAM cluster, it allocates a page to the DRAM cluster for D times, then allocates the page to the NVM cluster for N times. The page is allocated to the NVM cluster for N times if the task is in the NVM cluster. The page is then allocated to the DRAM cluster for D times.

3.3 Bandwidth-aware Migration Policy

The bandwidth-aware migration (`BW-MIGRATION`) policy performs page migration while maintaining the optimal allocation ratio. For this, the number of pages migrated between clusters is kept the same within the threshold. The `BW-MIGRATION` policy classifies migration into two types. The first is intra-cluster migration. Intra-cluster migration is a page migration between nodes belonging to the same cluster. The second is inter-cluster migration, which performs page migration between different clusters. The inter-cluster migration uses two global variables. These variables are used to maintain the optimal allocation ratio. The first is a migration threshold that prevents migration from being overloaded to a specific cluster because of migration, and is expressed as M in pseudocode. For example, the inter-cluster migration from the NVM cluster to the DRAM cluster can be performed up to 1010 times and at least 990 times if the migration count is 10, and the inter-cluster migration from the DRAM cluster to the NVM cluster is performed for 1000 times. The second variable, migCount, has a value between -M and M and stores how many more times the inter-cluster migration was performed in a particular direction.

Algorithm 4 BW-aware migration policy

```
migCount ← 0
procedure bwAwareMigrate(task, page)
    currCluster ← page.cluster
    destCluster ← task.cluster
    if currCluster ≠ destCluster then
        interClusterMigAllowed ← false
        currCount ← migCount
        nextCount ← 0
        if destCluster = DRAM then
            if currCount < $M$ then
                interClusterMigAllowed ← true
                nextCount ← currCount + 1
        else                                                    ▷ destCluster = NVM
            if currCount > $-M$ then
                interClusterMigAllowed ← true
                nextCount ← currCount − 1
        if interClusterMigAllowed = false or atomicCompareAndSet(migCount, currCount, nextCount) = failed then
            destCluster ← currCluster
    doMigration(task, page, destCluster)
```

The pseudocode of the BW-MIGRATION policy is Algorithm 4. The intra-cluster migration is performed immediately without any additional process (Line 19). In the case of the inter-cluster migration, the migration direction, migCount, and migration threshold values determine whether migration is permitted. When the inter-cluster migration is to be performed from the DRAM cluster to the NVM cluster, the interClusterMigAllowed is set to true and the nextCount is set to currCount+1 if the current migCount less than or equal to M (Lines 9–10). On the contrary, when the inter-cluster migration is to be performed from the NVM cluster to the DRAM cluster, interClusterMigAllowed is set to true and the nextCount is set to currCount-1 if the current migCount greater than -M (Lines 13– 16). If the interClusterMigAllowed is true, then atomic compare and set is performed (Line 11). The atomic compare and set operation compares the migCount and currCount and sets the migCount to nextCount if they are equal. If the interClusterMigAllowed is false or the migCount and currCount are different, the atomic compare and set operation returns false, and the inter-cluster migration is not performed.

3.4 Implementation

We developed and tested the bandwidth-aware placement and migration policies in the centos 7, 3.10.0 kernel. Centos 7 is a widely used Linux distribution, and the default kernel is 3.10.0.

3.5 Discussion

Three or more memory clusters: The previous description of the bandwidth-aware memory placement and migration policies is based on the existence of only two types of clusters in a heterogeneous memory system. However, the bandwidth-aware memory placement and migration policies can normally operate even in the presence of three or more clusters. Let us assume a heterogeneous memory system to have three or more clusters (i.e., $C_1, C_2, \cdots, C_N$). The bandwidth of C_i is B_i. The optimal allocation ratio of each cluster is calculated as follows: $p_{i,OPT} = \frac{B_i}{B_1+B_2+\cdots+B_N}$.

The bandwidth-aware migration policy can also normally operate in the presence of three or more clusters. For page migration to be performed while maintaining the optimal allocation ratio, the amount of pages migrated between the clusters must be the same. To do this, a migration count is needed to store the number of page migrations between two clusters. The migration count specifically stores how many more times page migration has been performed in a particular direction between two clusters. Therefore, $\binom{n}{2}$ migration counts are needed when the number of clusters is n.

Latency: The bandwidth-aware memory placement and migration policies focus on optimizing the performance of bandwidth-intensive applications. Moreover, our experimental results showed that latency did not significantly affect performance in the case of multi-thread bandwidth-intensive benchmarks. As shown in Figure 5.1 in Section 5.1, the bandwidth-aware local policy allocates memory considering locality differently from other bandwidth-aware memory placement policies. However, the performance difference between the bandwidth-aware memory placement policies is very small. Furthermore, as shown in Figure 5.8 in Section 5.3, the bandwidth-aware memory placement and migration policies have little or no performance impact on the bandwidth non-intensive benchmark.

Chapter 4. Methodology

The experiments of the bandwidth-aware memory placement and migration policies were performed in a NUMA system with two nodes. The NUMA system had two eight-core CPUs, which were directly connected through the interconnection network, as shown in figure 2.4. Each CPU had 16GB of local memory. Table 4.1 lists a more detailed specification of the NUMA system used.

NVM is not yet released; hence, the experiment herein emulated one of the two DRAM nodes to the NVM through Quartz [7]. Quartz can control the thermal throttling of the memory controller. We reduced the DRAM bandwidth emulated as NVM by 2, 5, and 10 times.

CentOS 7 and kernel 3.10.0 were installed on the NUMA system used in the experiment. The kernel was modified, and approximately 500 lines of code were added for the implementation of the bandwidth-aware memory placement and migration policies. We also used OpenJDK 1.8.0 and Spark 1.5.0 [8] for the big-data benchmark. The size of the page used in the experiment was 4KB. Table 4.2 presents the data collection methods.

Seven bandwidth-intensive benchmarks and five bandwidth non-intensive benchmarks were used in

Table 4.1: System specification

Component	**Description**
Processors	2 Intel Xeon Processor E5-2640 v3 CPUs @ 2.6GHz, 8 cores per CPU
L1 I-cache	Private, 32KB, 8 ways
L1 D-cache	Private, 32KB, 8 ways
L2 cache	Private, 256KB, 8 ways
L3 cache	Shared, 20MB, 20 ways
Main memory	32GB (2 × 16GB DDR4 (PC4 17000))
SSD	128GB

Table 4.2: Performance Data collection

Perf. data	**Source**
Execution time	PARSEC, SPLASH, and Spark statistics
Time breakdowns	Linux statistics (/proc/stat)
Memory bandwidth	Intel performance counter monitor (pcm-memory.x)
Memory allocation	Redhat performance monitoring tool (numastat)
Local memory accesses	Performance monitoring counters (PMCs) – event: 0xB7, mask: 0x01, sub-event: 0x600400001
Remote memory accesses	PMCs – event: 0xBB, mask: 0x01, sub-event: 0x67F800001
Page migrations	Linux statistics (/proc/vmstat)

Table 4.3: Bandwidth requirements (reads and writes) of the evaluated benchmarks. The first seven benchmarks are bandwidth-intensive benchmarks, and the next five benchmarks are bandwidth non-intensive benchmarks.

Benchmark	**Bandwidth Requirements**	
	Reads (MB/s)	**Writes (MB/s)**
`canneal` (`CA`) [9]	8082.6	2988.7
`FFT` (`FFT`) [10]	6628.7	4678.8
`kmeans` (`KM`) [11]	8018.5	7228.5
`ocean_cp` (`OC`) [10]	13154.6	4777.0
`ocean_ncp` (`ON`) [10]	12047.8	4018.4
`streamcluster` (`SC`) [9]	14589.2	168.3
`wordcount` (`WC`) [11]	8795.9	6973.7
`blackscholes` (`BL`) [9]	2952.3	448.0
`facesim` (`FS`) [9]	3420.8	1076.1
`freqmine` (`FM`) [9]	1333.0	591.6
`raytrace` (`RT`) [10]	196.8	30.9
`swaptions` (`SW`) [9]	669.5	81.7

the experiment. The benchmarks used belong to the PARSEC [9], SPLASH [10], and BigDataBench [11] benchmark suite. The largest dataset was used when executing the PACSEC and SPLASH benchmarks. Wordcount and Kmeans were used in BigDataBench, with 8GB and 1GB datasets, respectively. A total of 16 threads were used in all the experiments, except for the multiprogrammed workload experiments. Two benchmarks each used eight threads in the multiprogrammed workload experiment.

Table 4.3 shows the bandwidth information of each benchmark used in the experiment. Local policy and 16 threads were used to measure the bandwidth information. Memory throttling was not used. The Intel performance counter monitor (pcm-memory.x) in Table 4.2 was used as the bandwidth measurement tool.

Chapter 5. Evaluation

This section describes the experiment results of the bandwidth-aware memory placement and migration policies. The performance evaluation of the bandwidth-aware memory placement and migration policies are divided into four parts. First, Section 5.1 describes the overall performance of the bandwidth-aware memory placement and migration policies and the conventional memory placement and migration policies. Second, Section 5.2 describes how performance varies with the variation of the bandwidth ratios of memory clusters. Third, Section 5.3 describes the impact of the bandwidth-aware memory placement and migration policies on the bandwidth non-intensive benchmarks. Fourth, Section 5.4 describes the impact of bandwidth-aware memory placement and migration policies on multiprogrammed workloads. Unless otherwise stated, the benchmark used in the experiments is a bandwidth-intensive benchmark, and the bandwidth ratio of the DRAM and NVM clusters is 2:1.

5.1 Performance Impact on Bandwidth-intensive Benchmarks

Seven memory placement and migration policies were used in the experiment. First, two conventional policies, namely local (`Local`) and interleave (`IL`), were used. DRAM-only (`D-only`) was also used. It does not use an NVM cluster, but uses a DRAM cluster only. Next, four bandwidth-aware memory placement and migration policies, namely bandwidth-aware interleave (`BW-I`), bandwidth-aware random (`BW-R`), bandwidth-aware local (`BW-L`), and bandwidth-aware local augmented with the bandwidth-aware migration policy (`BW-LM`), were used.

Figure 5.1 shows the overall performance of the seven policies. The seven bandwidth-intensive benchmarks in Table 4.3 were used in the experiments. The benchmarks were run multiple times for each policy. Figure 5.1 presents the average execution time. The average execution time was normalized to `Local` policy. The variance of each execution time used in the average run time calculation is very small. For example, the geometric standard deviation (GSD) of `BW-I` is 1.029, which is very small. Note that the GSD 1 means no deviation.

As shown in Figure 5.1, the bandwidth-aware memory placement and migration policies achieved a higher performance than the conventional policies for bandwidth-intensive benchmarks. For example, `BW-L` achieved a 34.8% higher performance than the `local` policy on average. In contrast, the performance difference between the bandwidth-aware memory placement and migration policies was very small. The performance difference between `BW-I` and `BW-L` was only 0.5%. The performance of `BW-LM` was 5.8% lower than that of `BW-L` because the page migration overhead was larger than the performance gain of the page migration.

Figure 5.2 shows the details of the execution time. Figure 5.2 classifies the execution time as `User`, `System`, `Idle`, and `I/O`. As shown in Figure5.2, the bandwidth-aware memory placement and migration policies decreased the `User` time and the `Idle` time compared to the conventional policies. The reason for the `User` time decrement was that the bandwidth-aware memory placement and migration policies fully utilized all the system bandwidth, and the execution time of bandwidth-intensive benchmark was

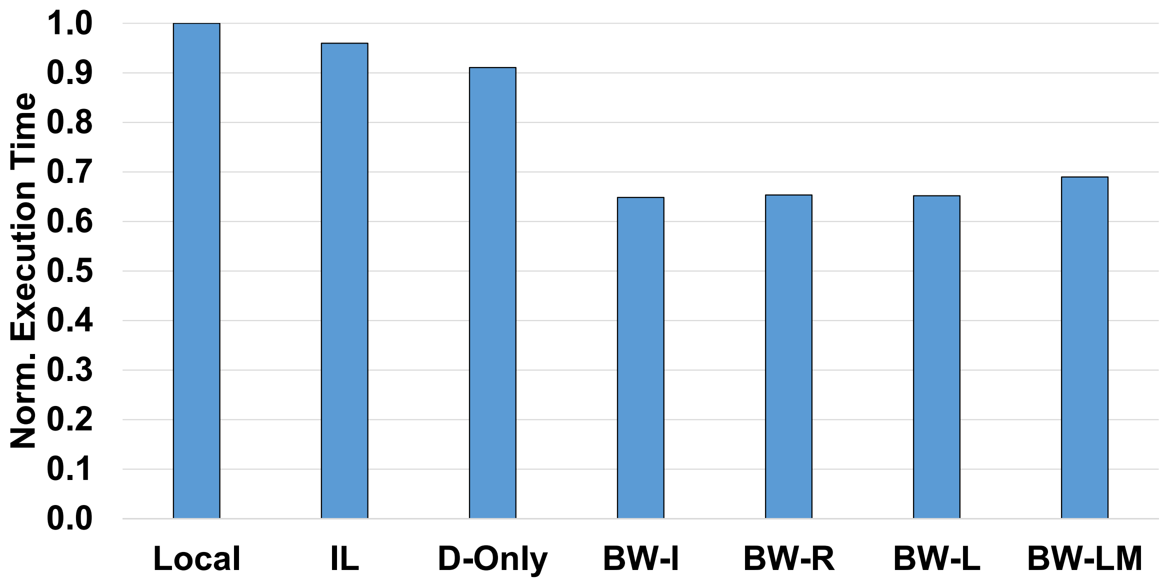

Figure 5.1: Overall performance results with bandwidth-intensive benchmarks

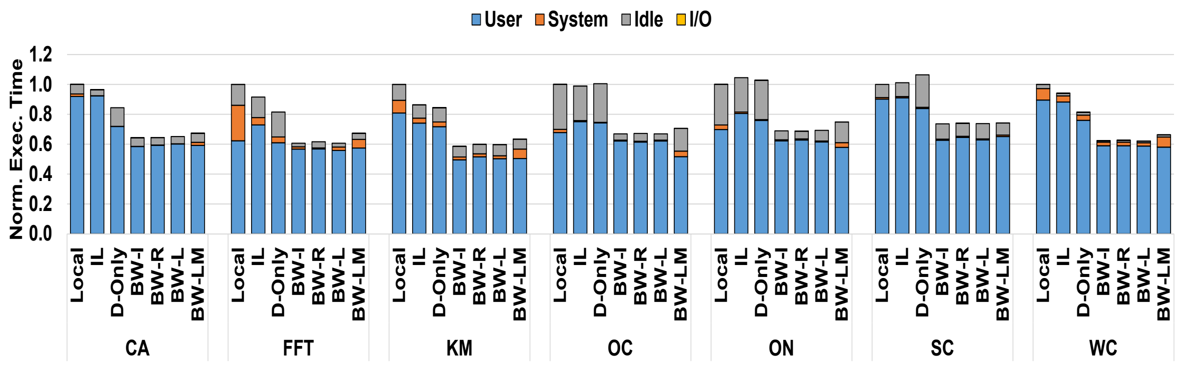

Figure 5.2: Execution time breakdowns

proportional to the data transfer time. The conventional policies allocate pages without considering the bandwidth, and throttling occurs because of low bandwidth of the NVM cluster. In contrast, the bandwidth-aware memory placement and migration policies allocate pages according to the optimal allocation ratio. As a result, throttling does not occur, and the `Idle` time decreases.

Figure 5.3 shows the bandwidth of the conventional policies and the bandwidth-aware memory placement and migration policies. The bandwidth of the bandwidth-aware memory placement and migration policies was higher than that of the conventional policies because the bandwidth-aware policies allocated pages according to the optimal allocation ratio. Figure 5.4 shows the allocation ratio of each policy. The conventional policies allocated memory at a rate close to 1:1 without considering the bandwidth of each cluster. The amount of bandwidth required for each cluster was proportional to the allocated memory. As a result, the ratio of the bandwidth that each cluster can provide and the bandwidth required by the application for each cluster were different from each other.

This can lead to a performance degradation of the bandwidth-intensive benchmark. No performance degradation is caused by the bandwidth if the bandwidth required by the application is less than twice the bandwidth that the NVM cluster can provide. However, if it is more than two times, throttling occurs

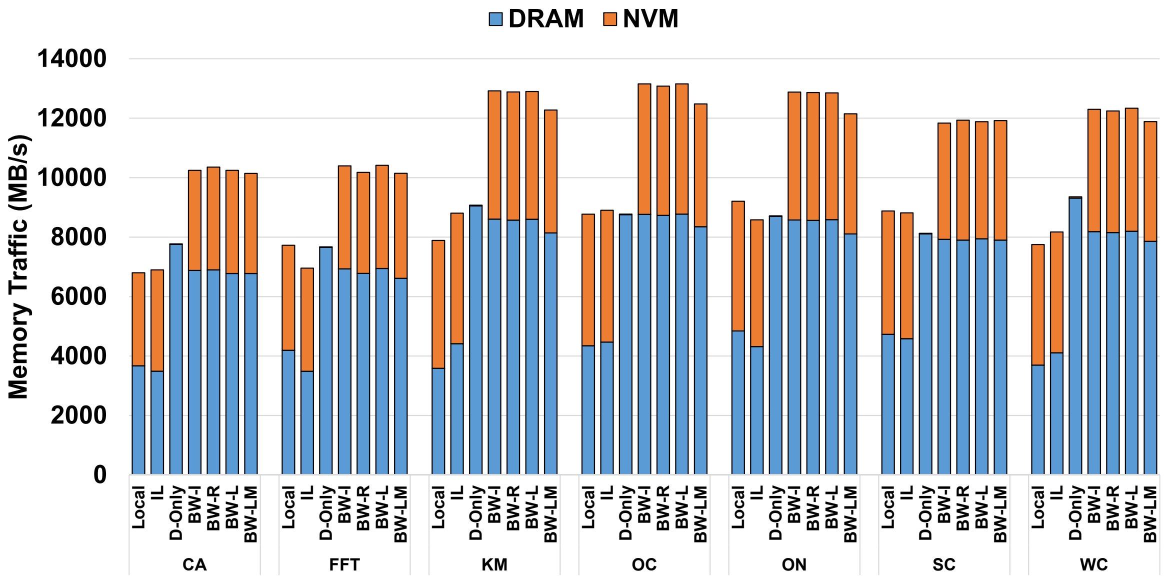

Figure 5.3: Memory traffic

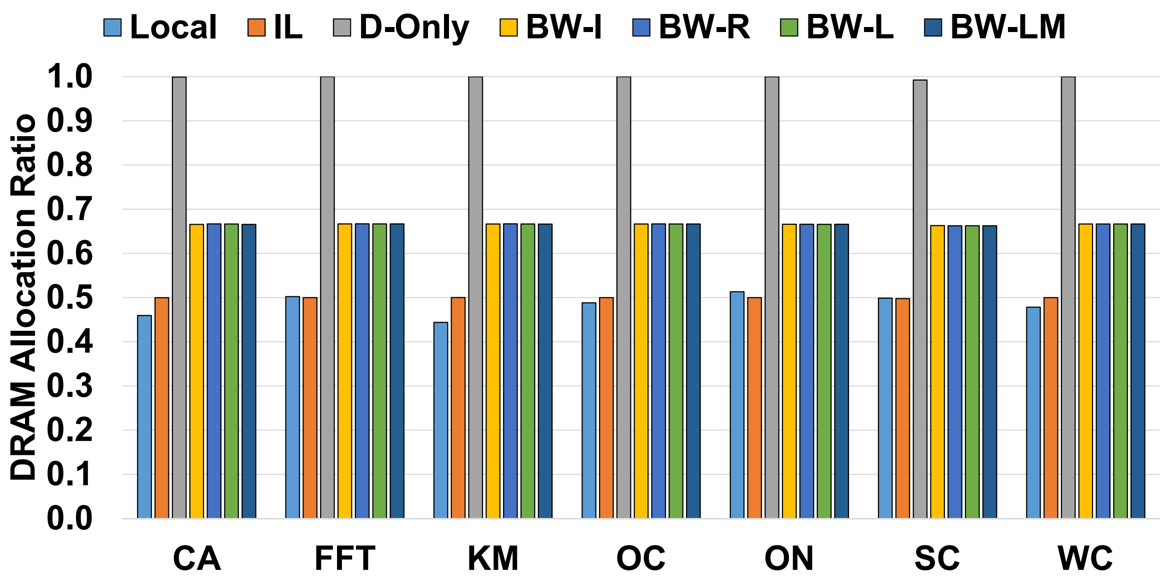

Figure 5.4: Memory allocation ratio

because the NVM cluster does not provide all the bandwidth required by the application. When throttling occurs in the NVM cluster, data is slowly transferred from the NVM cluster to the CPU. Even if the data is quickly transferred in the DRAM cluster, the CPU cannot process the operations because it needs data in the NVM cluster. As a result, the `Idle` time is increased. On the contrary, since the bandwidth-aware memory placement and migration policies allocate pages according to the optimal allocation ratio, the application requests bandwidth in proportion to the bandwidth for each cluster. Figure 5.5 show memory access breakdowns. A comparison of the memory access of Figure 5.5 with the memory traffic of Figure 5.3 shows that the allocated memory and bandwidth were proportional. The bandwidth-aware memory placement and migration policies allocate memory according to the optimal allocation ratio; hence, the application requires bandwidth in proportion to the bandwidth for each cluster. Therefore,

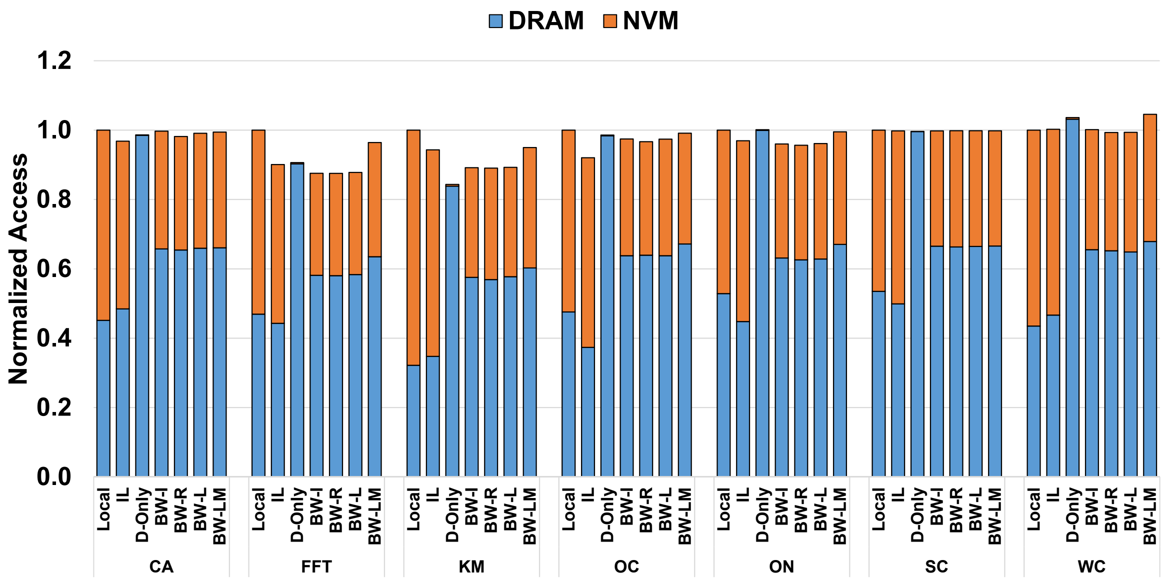

Figure 5.5: Memory access breakdowns

the bandwidth-aware memory placement and migration policies utilize all the bandwidth of each cluster, which eliminates throttling caused by the low bandwidth of a particular cluster.

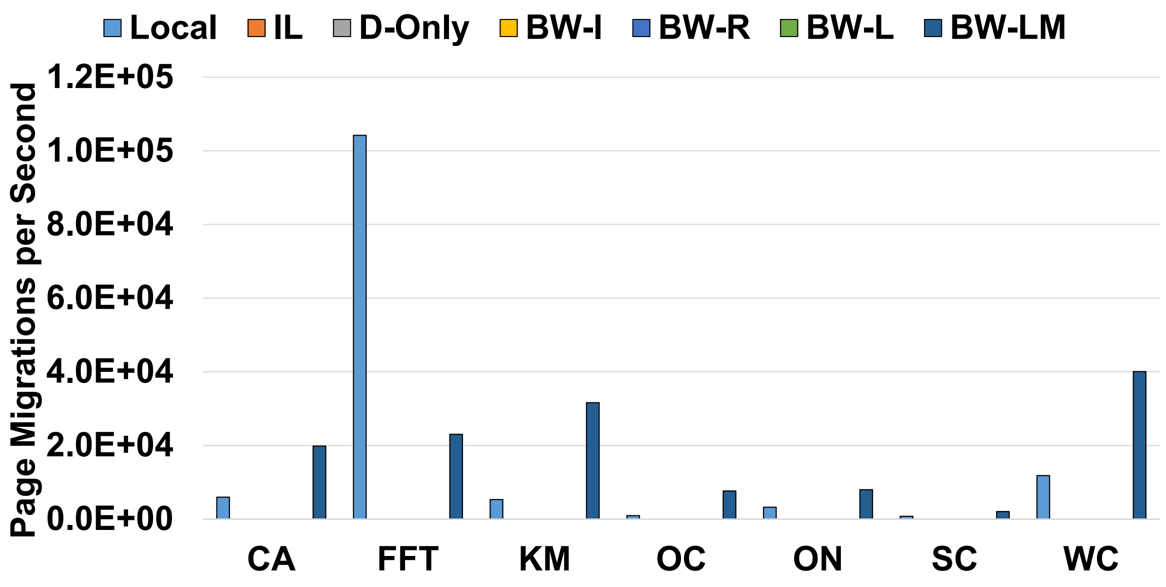

Figure 5.6: Migrated pages

The `Local` and `BW-LM` policies have a higher `System` time than the other policies because of the migration overhead. Figure 5.6 shows the number of page migration for each policy. Only the `local` and `BW-LM` policies migrated, while the other policies did not perform migration; hence, the number of migration was zero.

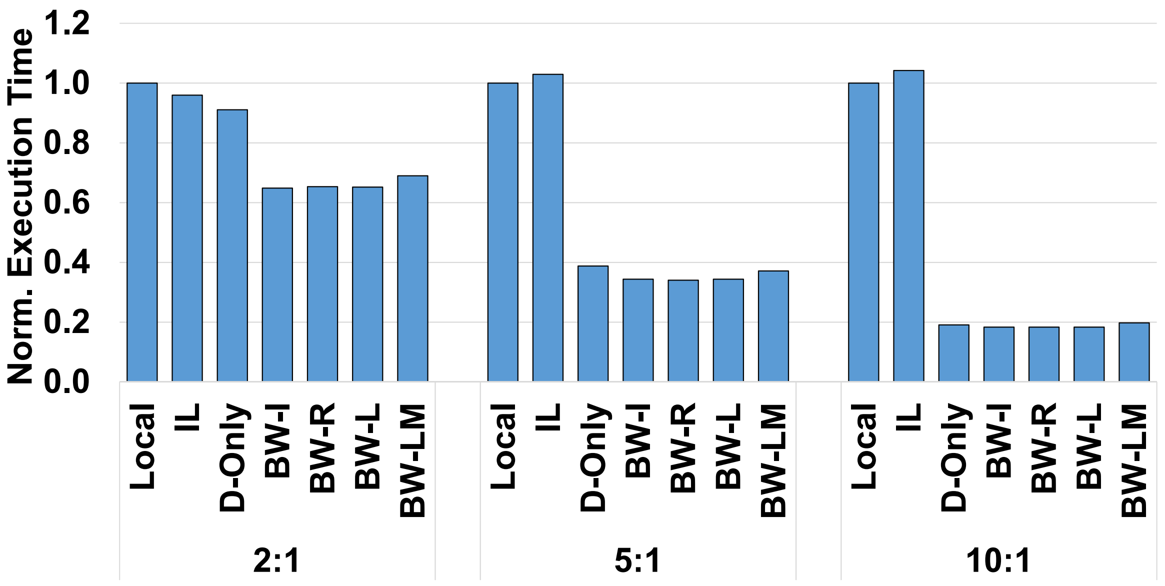

Figure 5.7: Performance sensitivity to the bandwidth ratio

5.2 Sensitivity to the Bandwidth Ratio

The previous subsection described the experimental results when the bandwidth ratio was 2:1. This subsection will explain the performance change depending on the DRAM and NVM cluster bandwidth ratios. Figure 5.7 shows the average execution times for each policy with bandwidth ratios of 2, 5, and 10. The average execution time of each policy was normalized to a `Local` policy. As shown in Figure 5.7, the performance difference between the conventional and bandwidth-aware policies increased as the bandwidth ratio increased. The `BW-L` policy showed a 34.8% higher performance than the `local` policy when the bandwidth ratio was 2. Meanwhile, it showed an 81.6% performance improvement when the bandwidth ratio was 10. This finding was attributed to the lower bandwidth of the NVM cluster causing more severe throttling as the bandwidth ratio increased. In contrast, the performance difference between the D-Only and bandwidth-aware policies became smaller as the bandwidth ratio increased because the gain of the NVM cluster became less as the bandwidth of the NVM cluster became smaller despite utilizing the NVM bandwidth.

5.3 Performance Impact on Bandwidth Non-intensive Benchmarks

Figure 5.8 shows the average execution time for each non-intensive benchmark when using each policy. The average execution time was normalized to `Local` policy. The average execution time for each policy was approximately the same, indicating that the bandwidth-aware memory placement and migration policies had little or no performance effect on bandwidth non-intensive benchmark.

5.4 Performance Impact on Multiprogrammed Workloads

This subsection describes the effect of the bandwidth-aware memory placement and migration policies on multiprogrammed workloads. Figure 5.9 shows the average execution time when two

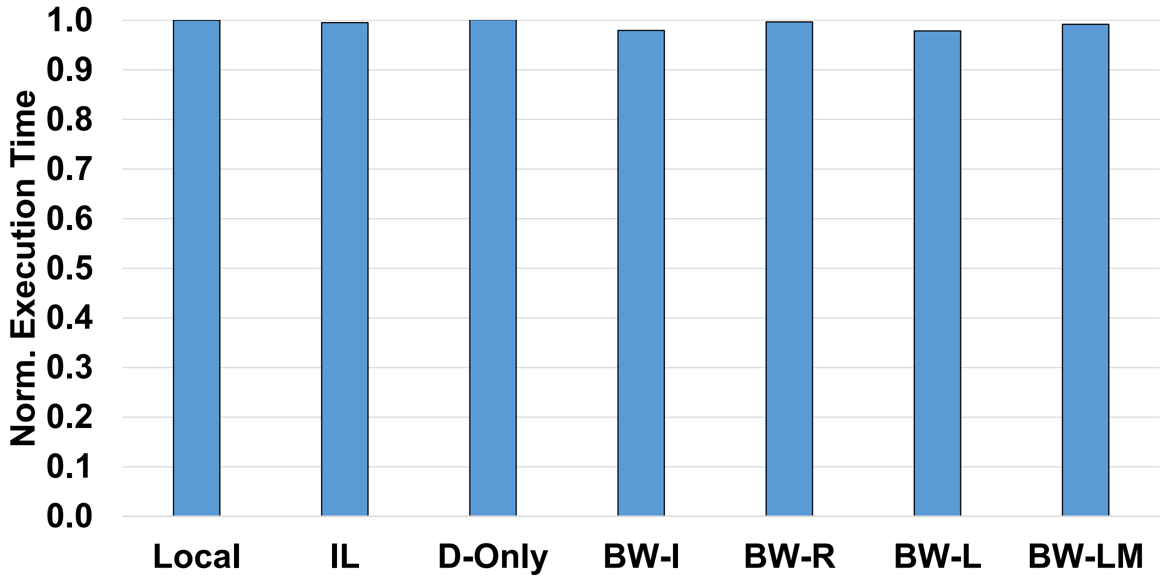

Figure 5.8: Overall performance results with bandwidth non-intensive benchmarks

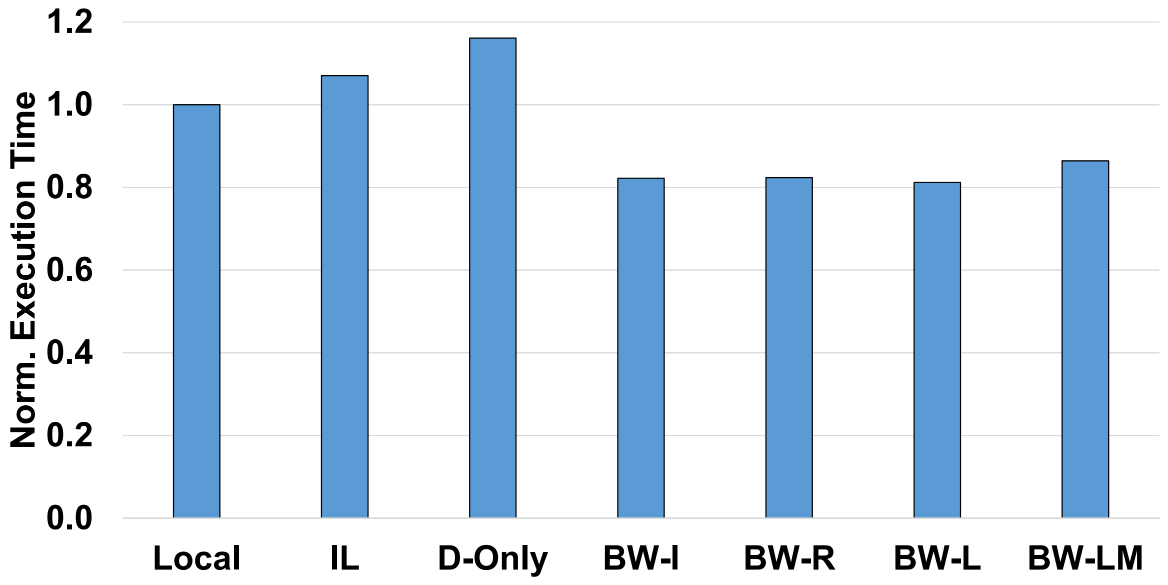

Figure 5.9: Overall performance results with multiprogrammed workloads

bandwidth-intensive benchmarks were simultaneously executed with eight threads. The average execution time was normalized to `Local` policy. Only `CA`, `FFT`, `OC`, `ON`, and `SC` were used in the experiment because `KM` and `WC` had a large change during the execution time because of I/O. All combinations that can be made with five benchmarks (i.e., $\binom{5}{2} = 10$) were used in the experiment. Each execution time was measured with makespan.

As shown in Figure 5.9, the bandwidth-aware memory placement and migration policies can also improve the performance in multiprogrammed workloads. However, the performance improvement of `BW-L` was 18.8% compared with that of the `Local` policy in multiprogrammed workloads. This performance gain was less than that of the other performance gains that the bandwidth-aware memory placement and migration policies obtained from the single-programmed workload. This finding can be attributed to the required bandwidth being reduced if one workload is terminated first in multiprogrammed workloads. When the required bandwidth was reduced, throttling did not occur or only weakly occurred even if the bandwidth-aware memory placement and migration policies were not used. In other words, the performance gain that the bandwidth-aware memory placement and migration policies can achieve was reduced in multiprogrammed workloads.

Chapter 6. Related Work

There are many previous works [2, 12, 13, 14] on heterogeneous memory systems where DRAM and NVM coexist. Kannan et al. particularly allowed to manage DRAM and NVM in a single virtual space and studied how NVM can be used to perform tasks that require persistence attributes. However, they did not study how to maximize the bandwidth of different types of memory clusters on the system.

The most similar previous work was that of [12]. This work studied the bandwidth-aware memory placement policy that considers the CPU and GPU bandwidths. However, this work only proposed a policy that corresponded to the bandwidth-aware interleave policy of our work. Moreover, the experiment was performed on a simulator, not in a real system. In contrast, our work presents not only bandwidth-aware interleave, but also other bandwidth-aware memory placement and bandwidth-aware migration policies, such as the bandwidth-aware random, bandwidth-aware local, and bandwidth-aware migration policies. In addition, we implemented the bandwidth-aware memory placement and migration policies by adding a new code to the Linux kernel or modifying an existing code. We also performed the experiments on real systems.

Chapter 7. Conclusions

This work investigated the design and implementation of bandwidth-aware memory placement and migration policies to maximize the performance of bandwidth-intensive applications in heterogeneous memory systems. The bandwidth-aware memory placement and migration policies allocate pages according to the optimal allocation ratio. In this work, the bandwidth-aware memory placement and migration policies were implemented and experimented on real systems. As shown in the experimental results, the policies outperformed the conventional policies for bandwidth-intensive benchmarks even if the bandwidth ratio changed. In addition, they achieved performance gains in multi-programmed workloads, and did not have a negative or positive effect on the bandwidth non-intensive benchmark.

Bibliography

[1] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 15:1–15:15. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592814

[2] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 13:1–13:16. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901325

[3] M. Gorman, "Understanding the linux virtual memory manager," https://www.kernel.org/doc/gorman/html/understand/, 2007.

[4] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on numa systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 231–242. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643659

[5] M. Gorman, "Foundation for automatic numa balancing," https://lwn.net/Articles/523065/, 2012.

[6] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555760

[7] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: ACM, 2015, pp. 37–49. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814806

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 24–36. [Online]. Available: http://doi.acm.org/10.1145/223982.223990

[11] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA '14, Feb 2014, pp. 488–499.

[12] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 607–618. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694381

[13] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 126–136.

[14] H. Wang, J. Zhang, S. Shridhar, G. Park, M. Jung, and N. S. Kim, "Duang: Fast and lightweight page migration in asymmetric memory systems," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 481–493.