Rethinking Remote Memory Placement on Large-Memory Systems with Path Diversity

Wonkyo Choe, Sang-Hoon Kim, and Jeongseob Ahn
Ajou University
{heysid,sanghoonkim,jsahn}@ajou.ac.kr

ABSTRACT

In this study, we rethink the existing memory placement strategies for multi-chip server systems. Processor vendors such as Intel and AMD have introduced the multi-chip based high density server architecture for large-scale data centers. With the advance of the processor-interconnect, multiple CPU chips are connected through a scalable point-to-point network such as Intel UPI and AMD Infinity Fabric. Nevertheless, the existing operating systems including Linux do not fully take advantage of the processor-interconnect to manage the memory traffic across memory nodes. We present two memory placement techniques exploiting the point-to-point interconnect to improve overall throughput while minimizing the hot-spot problem.

KEYWORDS

NUMA, Large Memory, Memory Placement

ACM Reference Format:
Wonkyo Choe, Sang-Hoon Kim, and Jeongseob Ahn. 2021. Rethinking Remote Memory Placement on Large-Memory Systems with Path Diversity. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3476886.3477516

1 INTRODUCTION

To reduce the cost of server infrastructure, the multi-chip server architecture [3, 5] has been widely used to build cost-effective servers as a scale-up design in large-scale data centers. Through a high-speed interconnect such as Ultra Path Interconnect (UPI) and Infinity Fabric (IF), multiple chips are connected point-to-point to each other. Such multi-chip memory systems are analogous to traditional non-uniform memory accesses (NUMA) systems with high chip counts [7][1].

[1]The term node and chip are used interchangeably throughout this paper.

APSys '21, August 24–25, 2021, Hong Kong, China

ACM ISBN 978-1-4503-8698-2/21/08…$15.00
https://doi.org/10.1145/3476886.3477516

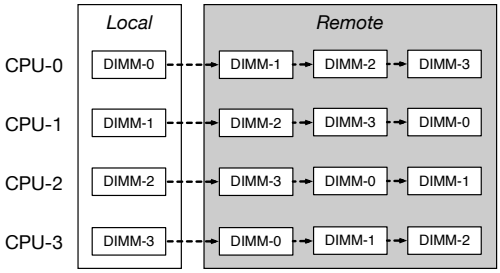


Figure 1: Default fallback node (chip) list used in memory allocation

Although there are significant efforts to optimize the performance of applications in traditional NUMA systems [2, 4, 6, 8], those studies did not explore the aspects of *remote* memory placement. In this study, we rethink the memory placement strategy on multi-chip server architecture especially when we are unable to allocate memory in the locally attached memory. Since remote memory accesses lead to a significant system performance degradation, Linux employs the default (`first-touch`) strategy which primarily allocates pages on the local memory where the requester is currently running on. However, when there is a lack of free pages in the local memory, remote memory accesses are inevitable. Figure 1 depicts the *fallback node list* which is used for selecting a remote node in memory allocation for each CPU chip. Remarkably, the default fallback list is a static linked list that is sorted simply by the chip number and does not change over time.

Even though there are multiple remote memory nodes, the memory requests are not dispersed across remote nodes in the current design. Although this approach performs reasonably well in small-scale NUMA machines such as dual-chip machines, the current design of the fallback paths fails to take full advantage of the emerging multi-chip NUMA systems which provide diverse paths of memory placement. We refer the advantage of the systems as *path diversity*. When utilizing remote machine's memory in disaggregated systems, the same consideration can be also applied. As a result, the memory access traffic can be concentrated on a (or few) certain memory node while other nodes are underutilized or even idle. This can lead to performance degradation and interference.

The goal of this study is to improve the overall system throughput while minimizing performance interference by balancing the memory traffic of multi-tenant applications across chiplets. To minimize this problem, we explore two new memory placement strategies. First, we take a `hybrid` approach of `first-touch` and `page-interleave` strategies. Our allocator follows `first-touch` approach to minimize remote memory accesses if the local node has free space available. If we fail to allocate memory on the local node due to the lack of memory, we switch our strategy to evenly distribute the memory requests across remote nodes in multi-socket systems like `page-interleave` policy. Second, we enable the operating system to dynamically select a remote node in such large memory machines by considering the memory usage for each node, instead of relying on the static fallback node list. It alleviates the potential performance interference by avoiding memory allocation from highly occupied memory nodes. Third, we investigate a potential applicability of dynamic migration on runtime. We optimize few features of AutoNUMA [9] and combine it with our proposed scheme.

Our preliminary experimental results show that our proposed schemes improve the overall system throughput while minimizing performance interference for various combinations with SPECCPU2017, NAS, and graph benchmarks.

2 MOTIVATION AND BACKGROUND

2.1 Large-Memory Systems

Multi-chip systems have been gaining traction to build scale-up servers in data centers because it can easily provide high core counts and memory capacity. Recently, AMD revealed a multi-chip server architecture which has four processors dies in one package. Each processor die has its own memory controller for scalability and the dies are connected through the point-to-point interconnect called Infinity Fabric [5]. Intel also introduced the multi-chip based high density servers [3]. In this study, we use a four-chip Intel machine where each chip is equipped with a 16GB DDR4 DIMM and the chips are connected point-to-point through the Intel UPI technology.

A distinct characteristic of such multi-chip systems is that accessing any of the remote memory nodes can be done in single hop through the point-to-point network. Figure 2 shows the access latency for a local memory (Node 0) node and three remote memory nodes (Node 1, 2, and 3) on our Intel server and two AWS instances based on AMD EPYC [1]. We used the Intel `MLC` benchmark to measure the latency. We can observe consistent performance in accessing any remote memory node out of three regardless of the configurations.

According to this experiment, we conclude that the memory placement should exploit the path diversity so that it can avoid the hot-spot problem to a remote memory node.

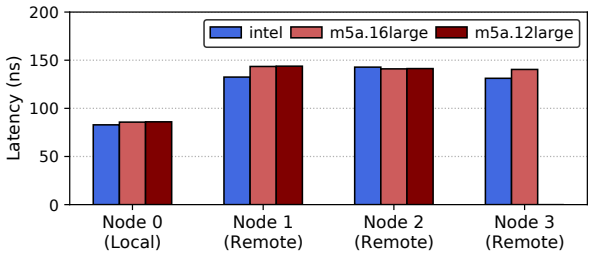


Figure 2: Read access latency (local vs. remote)

Furthermore, we would expect that the memory traffic is equally balanced across all the remote memory nodes in the first place.

2.2 Memory Placement Problem

We explain the observed inefficiency of the memory allocator on the multi-chip NUMA machines. First, the default (`first-touch`) strategy can make a hot spot to a certain remote memory node. If the local memory is fully occupied, the Linux kernel looks up free pages from a remote node as much as possible. For example, if the local memory (Node-0) is not enough to serve all the required memory for the application, the subsequent requests for memory allocation would be made on a remote memory (Node-1), specified in the fallback list depicted in Figure 1, rather than evenly distributed across remote nodes (Node-1, 2, and 3). Thus, we lose the opportunity to utilize diverse paths to Node-1, 2, and 3, for balancing the remote memory traffic. Moreover, the applications running on Node-1 can be highly affected by the memory bandwidth contention.

Second, the current memory allocator fails to find the best remote node. When selecting a remote memory node, it does not consider how much memory space is currently being used because the fallback node list is sorted by the node number and not changed at runtime. As a result, it can lead to imbalanced memory usage across memory nodes, making hot spots. For example, suppose two applications running on Node-0 and 1, respectively, and Node-2 and 3 are idle. Even though the Node-2 and 3 are idle, the application running on Node-0 would ask the memory of Node-1 when Node-0's memory is fully occupied. Then, the performance of the application on Node-1 can be significantly degraded due to the resource contention from the two applications.

2.3 Dynamic Memory Migration

Linux has adapted AutoNUMA [9] which provides thread and memory placement to reduce remote access and ensure better local access as possible. It is a supplement of memory management since the initial memory placement of the kernel does not always result in the best performance. The kernel using it keeps tracking `numa_faults` while workloads

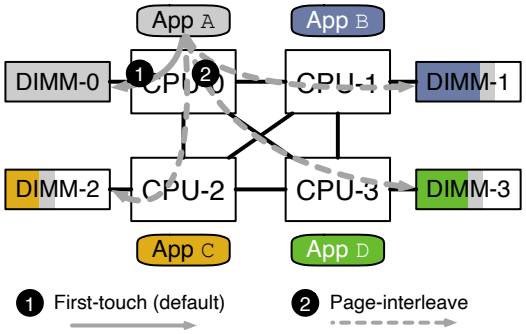


Figure 3: First-touch vs. Hybrid placement

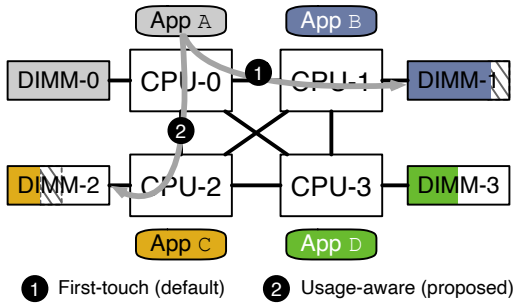


Figure 4: First-touch vs. Usage-aware allocation

are running. If the kernel thinks that there are heavy accesses on remote nodes, the kernel judiciously concludes whether it brings remote data to the local node or moves threads to remote nodes. This dynamic migration works in the general case. However, AutoNUMA cannot migrate pages to the local memory when the local memory is fully occupied. For instance, when a large footprint workload uses memory beyond the local memory, AutoNUMA may not be able to move data pages to the local node even if they have many remote accesses. Besides, AutoNUMA may migrate threads to remote nodes. As a consequence, it generates undesirable thread interference because there are other workloads on remote nodes.

Thus, we conclude that dynamic migration would not efficiently work on our scenario so initial memory placement and smart allocation during runtime are crucial. In this study, our scope is primarily to analyze memory placement and build alternative strategies for the default memory placement of Linux, but also discuss runtime policy in short.

3 DESIGN

This section presents simple yet effective design principles taking advantage of the path diversity of the point-to-point interconnection. We introduce two memory placement techniques balancing remote memory usage across them. The main benefit is to avoid that memory accesses are concentrated on a (or few) certain memory node(s).

3.1 Static Memory Placement

We first present a static approach to allocate memory called `hybrid` allocation policy: a combination of `first-touch` and `page-interleave` policies. In the stock Linux operating system, when a thread is not able to allocate memory on its local memory node, the kernel tries to allocate the memory on other remote memory nodes according to the static fallback memory list depicted in Figure 1. Suppose `App A` in Figure 3 runs out of local memory (`DIMM-0`). By consulting the fallback list, the traditional operating system attempts to allocate memory on a remote node (`DIMM-1`). If the memory access traffic from `App A` is significant, it can affect the performance of `App B` due to performance interference. On the other hand, our `hybrid` policy interleaves the allocation requests across three remote nodes evenly. This approach can reduce the memory hotspot problem while exploiting the abundant memory bandwidth across nodes. At the beginning of allocation, ❶ our approach follows the same behavior as what the default allocator does. However, when the application needs more memory than the memory DIMM-0 has, ❷ it utilizes each remote DIMM to obtain the full advantage of memory bandwidth.

3.2 Dynamic Memory Placement

Although the `hybrid` policy is simple and effective, it does not consider how each memory node is being utilized. To overcome such a limitation, we propose a `usage-aware` memory allocation policy that dynamically selects a remote memory node in a multi-chip NUMA system. Instead of statically choosing the next memory node when the local memory is no free space, it considers the memory usage for each memory node and then allocates memory on the least used memory node. Figure 4 depicts our `usage-aware` scheme. Similar to the `hybrid` policy, the initial allocation is the same as the `first-touch` policy. However, if the operating system is unable to allocate a page from the local memory node due to the lack of free space, the kernel will find the other memory node to get the page. ❶ In the `first-touch` policy, it selects DIMM-1 according to the static fallback list. ❷ Our `usage-aware` approach steers such requests to DIMM-2, which is the least utilized memory node. As a result, it leads to minimizing such performance interference.

3.3 Hybrid vs. Usage-aware Placement

Although both placement strategies outperform the default `first-touch` policy, they perform sub-optimally and complement each other. For the `hybrid` policy, it distributes memory across the multiple memory nodes, thereby benefiting from the wide memory bandwidth. As a result, the policy inevitably interferes with all workloads running on

the remote nodes. The amount of interference on some workloads is lighter than the default policy, but some workloads may have sensitive to mere interference. On the other hand, `usage-aware` policy shows different outcomes. Since it tries to find a node that has the freest pages, it may keep falling onto the same node. For instance, in Figure 4, DIMM-2 is the target node for allocation. As far as other remote DIMMs have less free pages than DIMM-2, the kernel would only choose DIMM-2. Due to this feature, the `usage-aware` policy is able to overcome interference on every node caused by the `hybrid` policy. However, this characteristic may show a harmful effect on the performance of an application located on DIMM-2. If it runs similar to the above case, focusing only on the same node, the workload would not take advantage of multiple DIMM's bandwidth. Furthermore, selecting the same node leads to centralizing the access into the single node (e.g. DIMM-2), causing severe interference to the tenant workload. In some cases where all the remote memory nodes have the same amount of free pages, `usage-aware` memory allocation works just like `page-interleave` policy across them. This case may lessen the skewed access problem.

With this perspective, we realize that memory management during runtime is required to maximize the performance of multiple applications. For checking the effectiveness of dynamic management we extend AutoNUMA functions to ease the skewed remote accesses. We did evaluate and discuss this feature; however, there are overheads of dynamic migration so we leave a further deep study on this as future work.

3.4 Summary

In this preliminary study, we focus on how we can exploit the path diversity in terms of memory placement. The main contribution of this paper is to improve performance and mitigate interference easily with simple alternatives. We implement our schemes on top of the Linux operating systems easily because many building blocks already exist.

4 IMPLEMENTATION

Hybrid and Usage-aware policy: Our policies utilize provided functions in Linux. Linux already keeps track of the number of allocated and free pages in each memory node. In Linux, `struct pglist_data` represents a node struct and it contains several zones (`struct zone`). They hold information that how many pages are allocated or how many free pages are available. By leveraging the information, we are able to statically or dynamically pick the next best node for allocation in each policy. Note that when we consider the number of allocated pages, these include all active and inactive anonymous and file pages.

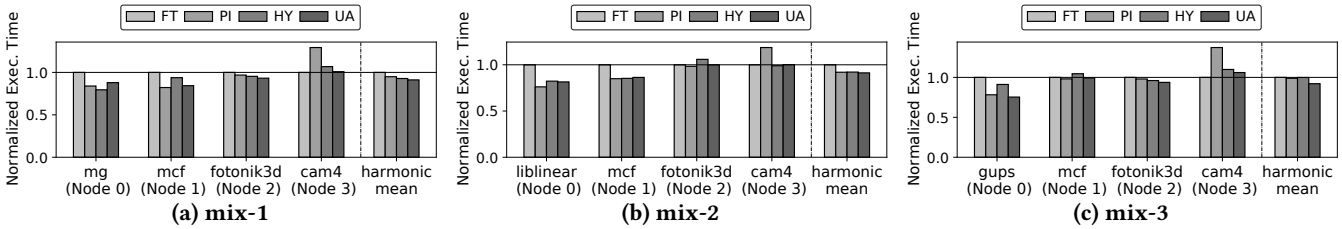
AutoNUMA extension: We make use of few functions of AutoNUMA and enhanced them for dynamic migration. AutoNUMA collects the local and remote statistics of `numa_fault`. We use these data to estimate which node suffers from excessive accesses caused by multiple applications. With this information, we periodically move pages from the busiest node that has the most memory traffic to the least busy one.

5 EVALUATION

We evaluated `first-touch` (FT), `page-interleave` (PI), our proposed schemes, `hybrid` (HY), and `usage-aware` (UA) on a multi-chip NUMA system equipped with four Intel Xeon Gold 6242 processors (16 physical cores). We use 4 sockets machine and each of them has a single CPU chip. For PI policy, we allow each workload to use all memory nodes. For instance, if a page is allocated on Node-0 at first, the next time another page will be allocated on Node-1 and so on. When a page is allocated on the last node then the next page will be assigned on the very first node (Node-0). Each NUMA node has a 16 GB DRAM DIMM so that the total main memory is 64 GB. We used Linux kernel 5.3 as the baseline of our implementation. We select memory-intensive applications: SPECCPU2017, MG from the NAS parallel benchmark, Liblinear, and GUPS from a HPC Challenge benchmark for evaluations. We evaluate them with different mixed sets to observe various access and allocation patterns. Each mixed case differs only from the first application on Node-0. On Node-0, we configure a benchmark that has a large working-set so that it requires the use of a remote memory node while other `SPEC` benchmarks running on Node-1, 2, and 3 fit on their local memory node. We pick 3 memory-intensive workloads (`mcf`, `fotonik3d`, and `cam4`) from `SPEC`. The default option of AutoNUMA is enabled in the current Linux. We configure the same option, but this function does not work as intended for two reasons. First, AutoNUMA does not move pages on the fully allocated node. Second, threads of applications are pinned on each socket. Thus, the system cannot benefit from using the AutoNUMA. All performance numbers are measured three times and their average values are applied on the graphs.

5.1 Performance of Proposed Policies

We deploy four applications on each CPU chip. All the applications are tuned with 16 threads to fully utilize the cores on the chip. This fully utilized environment can exhibit intensive usage of memory bandwidth for each application. Moreover, memory interference between applications due to using the same memory node can be shown significantly in this environment. Figure 5 shows the speedup of individual applications normalized to FT policy. For `mg`, `liblinear`, `gups`, and `mcf`, PI policy can provide better performance than



(a) mix-1 (b) mix-2 (c) mix-3

Figure 5: Performance comparison of `first-touch` (FT), `page-interleave` (PI), proposed `hybrid` (HY) and `usage-aware` (UA) schemes on fully utilized environments (Lower is better)

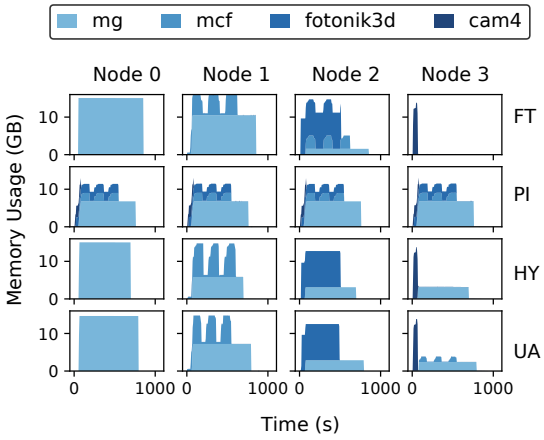


Figure 6: Memory usage comparison across four policies for `mix-1`

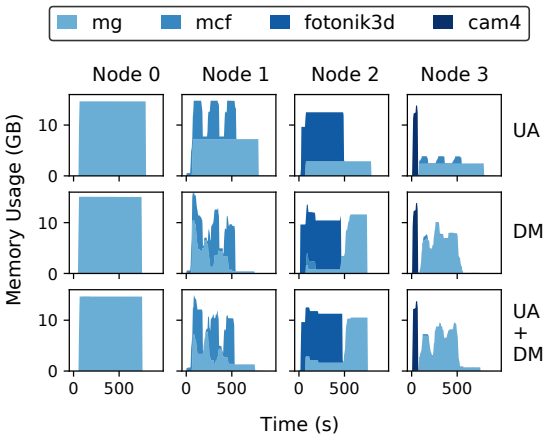


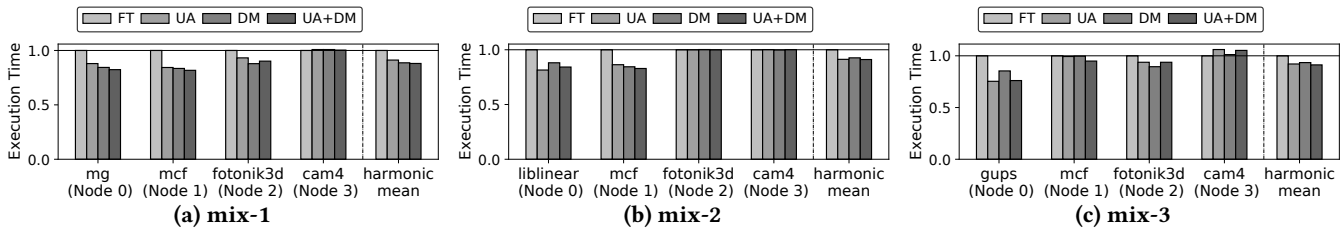
Figure 7: Memory usage comparison for dynamic polices

FT policy because performances of these applications are bounded in the memory bandwidth. In other words, those applications will improve performance when they are able to use an extra memory channel or bandwidth. On the other hand, the performance of `cam4` benchmark is significantly degraded on PI policy. Differently from PI policy, HY and UA have the benefits of not (or little) impairing the performance of `cam4`. For `mcf`, it suffers from interference from `mg` and `liblinear` on FT policy. However, this workload has less interference in HY and UA policy. Thus, it has a better result than FT policy. The reason for the improvement of `mcf` on UA policy can be also explained by using extra memory bandwidth on Node-3. `fotonik3d` is considered as memory-intensive workload, but it is not sensitive to interference from other workloads.

For better understanding of memory interference, Figure 6 presents how four applications utilize each memory node. Compared to FT scheme, HY policy places pages of `mg` on each remote node as the interleave mode when the local memory cannot take further memory allocation. For UA policy it distributes the remote memory allocation of `mg` across Node-1, 2, and 3 based on memory usage. Specifically, when `mg` requests more memory, the first targeted node is Node-1 because Node-1 has fewer allocated pages than other remote nodes. After that, the kernel will select Node-2 and Node-3 according to their memory usage. Due to this dynamic memory placement of UA, we can improve overall performance by minimizing interference.

5.2 Performance Comparison with Extended Dynamic Migration

We compare our proposed schemes with the extended AutoNUMA features to investigate `dynamic migration` on our targeted scenario discussed in Section 3.3. We evaluate it on the same experimental setup in the previous section. The first modified one, `dynamic migration` (DM), is to provide dynamic data migration on the remote node when a certain node is highly stressed due to excessive interference. The second policy is the optimized DM, but augmented with UA policy (UA+DM). This policy basically operates like UA at allocation time. However, when the system needs to balance traffics across nodes, it follows DM policy. Figure 7 clearly presents how those features operate. For DM and UA+DM, the memory usage of `mg` on Node-1 shows decreasing pattern as time goes by while its usage on Node-3 is increasing. This



(a) mix-1 (b) mix-2 (c) mix-3

Figure 8: Performance comparison of `first-touch`, `usage-aware`, `dynamic migration` (DM), and `usage-aware + dynamic migration` (UA+DM) schemes on fully utilized environments (Lower is better)

phenomenon tells that memory traffic is moving into Node-3 from Node-1. As a result, not only does `mg` get the abundant bandwidth from multiple paths, but `mcf` can also effectively run by minimizing a negative interference of the noisy neighbor (`mg`). The result of those experiments is shown in Figure 8. In all sets of mixes, DM and UA+DM demonstrate better or similar performance with UA policy. The basis of improvement is that some workloads being pressed by excessive traffics are relieved by DM. For DM, the initial allocation works as the FT at allocation time so this inefficient allocation curtails the amount of performance improvement. On the other hand, UA+DM employs UA policy; therefore, it eliminates the inefficient placement.

Overall, the optimized AutoNUMA does gain improvement over the FT and PI. Nevertheless, it still needs more optimization. The AutoNUMA facility relies on the page fault mechanism causing performance overheads. It periodically scans a fixed amount of memory pages of the virtual memory, making them inaccessible to identify remote memory accesses. Once the pages are accessed again, it incurs a page fault. The cost of handling page faults is non-negligible. In addition, migrating pages to a different memory location requires the costly TLB shootdown operation. As a result, the effectiveness of dynamic page migration can be reduced.

6 PRIOR WORK

There are significant efforts to optimize the performance of applications in NUMA systems. Especially, Carrefour [4] has a similar goal to ours. This work proposed memory traffic managements for NUMA systems by threads and memory placement. Carrefour does address memory placement to mitigate the skewed traffic on a certain node and also reduce remote access. However, this work does not take account of interference between multiple applications. Moreover, there is a potential problem on their main feature, replication when multiple workloads are running. Multiple workloads should be isolated as possible to avoid interference among them. If replication is applied, replication pages will take a portion of memory resource of other workloads, thereby leading to interference. Unlike their approach, our work focuses on diminishing interference of multiple workloads with relatively simple approaches. Blagodurov et al. [2] studied the memory contention in NUMA and proposed a NUMA-aware scheduling. Lepers et al. [6] conducted an extended work on the asymmetry of the processor-interconnect. However, those prior studies did not consider the case where the memory footprint cannot be fit on the local memory, leading to the use of remote memory. When we are unable to place memory locally due to some reasons such as the lack of capacity, the next best plan we should prepare is to minimize performance impacts to all the applications sharing the system. In the real world, it is not always possible to make all the required memory on the local node. While running systems, memory is fragmented so that we need to leverage the remote memory in the NUMA environments. Thus, it is important to allocate memory on the remote nodes by minimizing performance interference.

7 DISCUSSIONS AND CONCLUSIONS

Decreasing the opportunity to localize data: It is known that Linux localizes data of a process in a (or few) NUMA domain(s). Potentially, this design approach can improve performance through thread migration rather than page migration. The proposed policy also follows a rule, keeping the data for a process to the local memory. However, when the system allocates the data to the remote memory, it disperses the data across NUMA domains, rather than accumulating it on the same NUMA domains. Therefore, it is difficult for the scheduler to minimize the remote memory accesses. In our future study, we will investigate gathering the scattered memory pages on a node when the memory space becomes available at runtime.

Node	Local	Remote-1		Remote-2		
	0	1	2	3	4	5
0	10	16	16	32	32	32

Table 1: Distance between Node-0 and other nodes in AWS `m5a.24large` [1]

Generalization for different typologies: On higher-end models, the topology of the processor-interconnect is composed of hierarchical paths with the variation of the distance. For example, the `m5a.24large` AWS instance consists of six nodes based on AMD EPYC 7571. TABLE 1 shows the node distance from the perspective of node-0. This topology can be seen in Intel eight-socket servers. Although we did not explicitly deal with the hierarchical topology, our approach can be extended easily. When distributing remote memory placements, we first consider the remote nodes which have the same distance. Once we do not find a remote node in the closest neighbor group (`Remote-1`), then we try to search a node in the faraway group (`Remote-2`).

8 CONCLUSIONS

Existing memory placement policies in the Linux operating systems can lead to the hot-spot problem while not exploiting the abundant remote memory bandwidth. In this paper, we proposed two alternative memory placement schemes and evaluate them with the extension of dynamic migration. Our policies take advantage of the path diversity of the processor-interconnect used in multi-chip NUMA systems. We plan to extend this preliminary design of the dynamic traffic management to be scalable and robust for diverse workload scenarios.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their invaluable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2019R1C1C1005166).

REFERENCES

[1] Jeff Barr. 2019. New Lower-Cost, AMD-Powered M5a and R5a EC2 Instances.

[2] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the USENIX Conference on Annual Technical Conference.*

[3] Ian Cutress. 2019. Intel's Enterprise Extravaganza 2019: Launching Cascade Lake, Optane DCPMM, Agilex FPGAs, 100G Ethernet, and Xeon D-1600.

[4] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[5] Kevin Lepak, Gerry Talbot, Sean White, Noah Beck, and Sam Naffziger. 2018. The Next Generation AMD Enterprise Server Product Architecture. In *A Symposium on High-Performance Chips (HotChips).*

[6] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the USENIX Conference on Annual Technical Conference.*

[7] Samuel Naffziger, Kevin Lepak, Mahesh Subramony, Noah Beck, Sean White, and Gabriel Loh. 2021. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA).*

[8] Wonjun Song, Gwangsun Kim, Hyungjoon Jung, Jongwook Chung, Jung Ho Ahn, Jae W. Lee, and John Kim. 2017. History-Based Arbitration for Fairness in Processor-Interconnect of NUMA Servers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems.*

[9] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA Balancing.