TAPMM: A Traffic-Aware Page Mapping Method for Multi-level NUMA Systems

Fengkun Dong, Guoqing Xiao*, Haotian Wang, Yikun Hu, Kenli li, Wangdong Yang*
College of Information Science and Engineering, Hunan University, China
{dongfengkun,xiaoguoqing,wanghaotian,yikunhu,lkl,yangwangdong}@hun.edu.cn

ABSTRACT

With the development of chiplet technology, the architecture of Non-Uniform Memory Access (NUMA) has become increasingly intricate. The placement of memory page significantly influences application performance in NUMA systems. We found that memory access bottlenecks occur between high-level NUMA domains consisting of multiple chiplets. In this paper, we introduce a Traffic-Aware Page Mapping Method (TAPMM) designed for multi-level NUMA systems. TAPMM conceptualizes the multi-level NUMA system as a memory access tree, utilizing hardware performance events to be aware of system traffic and identify the optimal page mapping method for bandwidth efficiency. Our experiments demonstrate that TAPMM achieves a speedup of up to 2.12× on a real commodity machine compared to existing optimization tools.

KEYWORDS

Memory allocator, Traffic-aware, Multi-level NUMA

ACM Reference Format:
Fengkun Dong, Guoqing Xiao*, Haotian Wang, Yikun Hu, Kenli li, Wangdong Yang*. 2024. TAPMM: A Traffic-Aware Page Mapping Method for Multi-level NUMA Systems. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656503

1 INTRODUCTION

With the development of CPU and failure of Moore's Law, CPU vendors improve performance by increasing the number of CPU cores. In order to avoid the bottleneck caused by multiple cores accessing memory, the adoption of Non-Uniform Memory Access (NUMA) architecture has become more prevalent. Particularly, for scalability improvements in chip architecture, chiplet design is gaining popularity in the field. Chiplets leverage advanced packaging technology to integrate multiple chips into a single socket, resulting in a highly integrated, high-performance chip. Chiplets allow designers to mix and match different chiplet types to meet specific performance, power or functionality requirements. However the chiplet topology brings new memory management problems[11], which we call the multi-level NUMA problem in this paper. Multi-level NUMA system is a computer architecture design used to handle the memory subsystem of multiple NUMA domains, multiple CPU dies are packaged together to form a larger NUMA domain. Each core is associated with one or more domains in multi-level NUMA systems.

The multi-level NUMA structure is designed to increase the scalability of the CPU. AMD's EPYC series CPUs and Huawei Kunpeng series CPUs both have multi-level NUMA domains. Fig.1 shows the architecture of Kunpeng 920[5], The entire system has two sockets, and each socket has two Super CPU Clusters (SCCL). Each SCCL is attached to its own local memory, which is called level-1 (L1) NUMA domain. The socket composed of two SCCL is defined as level-2 (L2) domain. The sockets communitcate with Hydra bus.

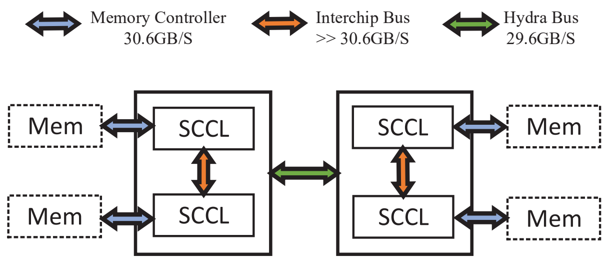
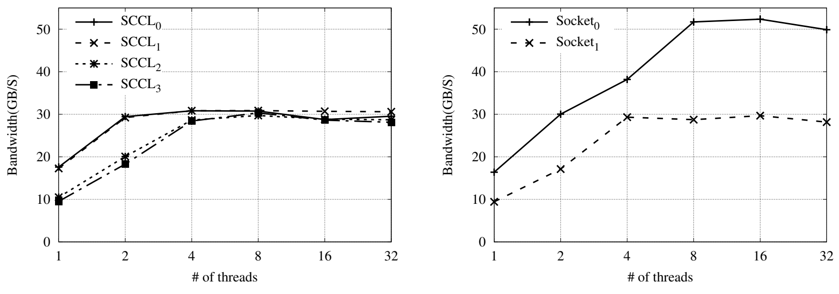


Figure 1: Architecture of Kunpeng 920

As shown in the Fig.1, tasks running on the L1 domain can obtain almost the same bandwidth from the four nodes, but if one socket initiates access to two memory domains on another socket, the Hydra bus will become a memory access bottleneck. We will explain this in detail in Section2.

There has been some works previously to optimize memory access in NUMA architecture, such as [1, 3, 6, 7, 10], these works have shown that, bandwidths of memory and NUMA node interconnect bus have a big impact on the performance. However, most of the previous works only consider the bandwidth between L1 NUMA domains and ignore the influence between higher-level domains. To solve the problem of memory access path bottleneck in multi-level NUMA system, we propose TAPMM. TAPMM does not require special configuration for each program, it only needs to model the machine when TAPMM is installed. When modeling the machine, it establishes mapping functions from traffic to the bandwidth for each bus from CPU to DDR memory. For bus traffic collection in the memory access path, TAPMM utilizes lightweight hardware performance events (HPE) to record Uncore events. After the installation is completed, TAPMM functions as a runtime library that dynamically monitors the bandwidth of each bus in the memory access path. TAPMM abstracts the topology of the server's CPU and memory nodes into a memory access tree. TAPMM utilizes

∗ Corresponding authors.



(a) Level-1 bandwidth (b) Level-2 bandwidth

Figure 2: Bandwidth at different levels

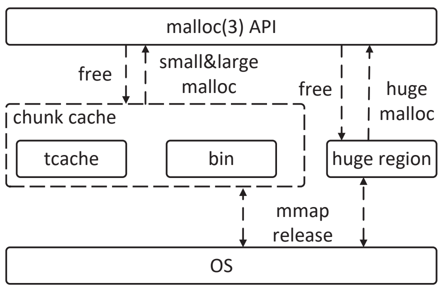


Figure 3: Jemalloc architecture

real-time bandwidth information obtained from the path to dynamically calculate and adjust the memory placement policy based on requests of different-sized memory.

The main contributions of this paper are as follows:

(1) We discover the bottleneck of memory bandwidth between high-level domains in a multi-level NUMA system. Our research results show that analyzing and optimizing memory bandwidth only from the L1 domain is not optimal to a large extent.
(2) We propose TAPMM, a runtime library that implements the malloc(3) interface , which can provide traffic-aware memory management for tasks running on multi-level NUMA systems. To our knowledge, TAPMM is the first NUMA memory management library that can run on ARM-based multi-level NUMA systems.
(3) We evaluate TAPMM through micro-benchmarks and real applications, proving that compared to the traditional interleave placement method, TAPMM can achieve up to a 1.44× increase in bandwidth. When compared with local memory access, TAPMM achieves speedups of up to 1.93× in single-application scenarios and 5.21× in multi-application scenarios, respectively.

2 BACKGROUND AND MOTIVATION

With the increase of cores, the enhancement of CPU scalability faces bottlenecks, leading to an increasing prevalence of multi-level NUMA processors. In contrast to single-level NUMA systems, multi-level NUMA systems exhibit a more intricate topology and present greater challenges in memory management. To further explore these issues, we carry out a series of simple experiments. We employ runtime analysis tools to monitor the allocation and deallocation operations for memory address, and obtain insights into the memory life cycle. And we utilize micro-benchmarks to analyze the bandwidth bottleneck of multi-level NUMA system. Our investigation leads to the following three observations:

Observation1. The majority of memory utilized in applications exhibits a relatively short-term lifecycle. This phenomenon has been substantiated in numerous prior studies[4]. We gather runtime data from the NAS Parallel Benchmarks (NPB)[2], utilizing a medium-scale (class=C) input configuration for NPB operations. By tracking the memory information in user space,

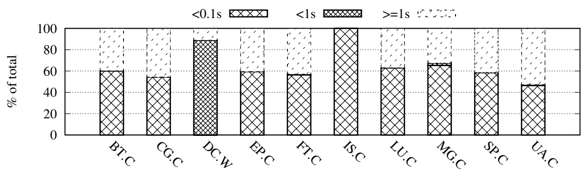


Figure 4: Memory lifecycle statistics for different applications

our findings reveal that the memory lifecycle in the majority of applications is notably brief. We categorize the statistical results into three groups, depicted in the Fig.4, almost all applications have more than 50% memory lifecycle less than 1 second. From another perspective, as memory capacity increases, the time required to scan a page table experiences a significant rise[9]. Therefore, the optimization method for migration is not well-suited for applications characterized by short-lived memory.

Observation2. Memory access bottleneck between high-level NUMA domains. As shown in the Fig.2, we conduct tests to assess the maximum bandwidth available at the L1 and L2 domains, with fixed threads running location at $SCCL_0$ and $Socket_0$, respectively. From the perspective of the L1 NUMA domain, as the number of threads increase, $SCCL_0$ consistently obtains nearly identical bandwidth from all four memory nodes. When examining the L2 NUMA domain, we observe that with an increase in the number of threads, task on $Socket_0$ no longer obtains the same bandwidth from both sockets. It becomes evident that the connection between sockets has evolved into a bottleneck for memory access.

Observation3. Allocators have a caching mechanism for normal size memory. We conduct an analysis of the existing memory allocator implementation and observe, as depicted in Fig.3, that to minimize system calls when allocating or releasing normal memory(size $\leqslant 4M$), operations are carried out from the *Chunkcache*. Previous studies[8] have indicated that each *free* operation refreshes the hotness of the address, thereby rendering data inaccurately detected by the hotness detection method. To ensure memory utilization when allocating huge memory(size $> 4M$), caching is typically not employed for optimization. From an empirical standpoint, huge and normal memories exhibit different sensitivities to bandwidth

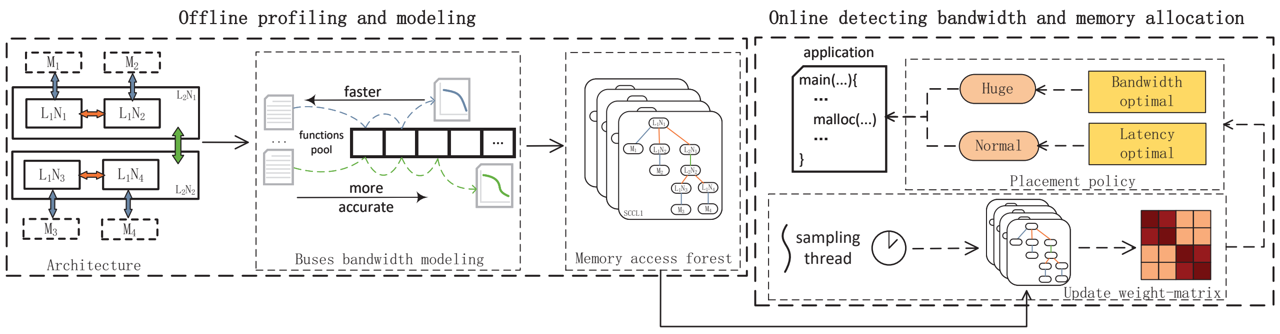


Figure 5: TAPMM overview

and latency. Specifically, huge memory has higher bandwidth requirements, whereas normal memory has higher latency requirements.

3 DESIGN AND IMPLEMENTATION

3.1 Overview

We implement the TAPMM in user space. Fig.5 shows an overview of TAPMM which includes two modules: *Offline profiling and modeling* and *Online detecting bandwidth and memory allocation*. As mentioned in Section2, To address the challenges posed by short-term memory lifecycles and insufficient memory migration, we propose a method in which memory page can be directly mapped to the optimal location when an application requests memory. In the offline phase, TAPMM completes traffic-to-bandwidth modeling for each bus in the memory access path, then conceptualizes multi-level NUMA into a memory aceess forest. Each tree in the forest is a memory access tree with L1 node as the root node. During the online phase, TAPMM uses asynchronous statistics of HPE to sense the real-time traffic of the system, then uses a lightweight graph algorithm to traverse the memory access forest, finds the bottleneck on each memory access tree, and adjust the memory mapping weight-matrix. From applications perspective, TAPMM is an extended *libjemalloc*[14], and applications running on it can be optimized without any modification. When an application applies for memory from the system, the memory allocator program acts as a middle layer, TAPMM uses the real-time updated memory allocation weight-matrix and the multi-level NUMA topology information obtained in the offline stage to place huge memory and normal memory in a bandwidth-optimal and latency-optimal manner respectively.

3.2 Offline profiling and modeling

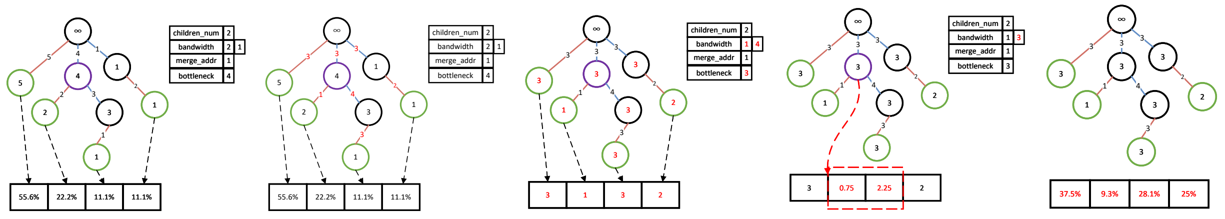
The offline profiling and modeling phase is completed prior to the compilation of TAPMM. Multi-level NUMA system information is obtained through configuration files and profiling results. After completing the construction of the memory access forest, the relevant code is generated and compiled into *libtapmm*.

Buses bandwidth modeling. In order to make TAPMM more versatile, we propose a modeling method that is independent of specific HPE in modeling traffic. Different manufacturers have different bus implementations, such as Intel's Ultra Path Interconnect (UPI), AMD's Infinity Fabric (IF) and HiSilicon's Hydra, therefore, we do not use certain events in the selection method of HPE, but select related events that can represent traffic. Taking Hydra as an example, CPU does not provide HPE that can directly obtain data flits like UPI, but we use Home Agent (HA) to characterize Hydra's traffic status.

After completing the selection of HPE, TAPMM begins to model various buses from traffic to bandwidth. Compared with the previous traditional modeling methods, the workload of profiling can be greatly reduced, we only need to pay attention to each type of bus on the memory access path, rather than every bus. As shown in the Fig.5, we only need to complete the sampling analysis of three buses: Memory controller, Interchip bus and Hydra bus. When profiling each bus, TAPMM continuously adjusts the backgroud traffic and record the HPE and available bandwidth in each state.

After profiling is completed, TAPMM utilizes the function pool to proactively model traffic to bandwidth. The function pool includes multiple function templates. As the index increases, the complexity and accuracy of the functions in the pool also increase. The approach employed in the function pool to achieve this goal is by continuously increasing the power of function templates, such as $ax + b$, $ax^2 + bx + c$ and so on. We use a threshold of error to trade off the accuracy of function fitting and the complexity of calculation. After this step, TAPMM completes the modeling of each bus from traffic status to bandwidth.

Memory access forest. As shown in the Fig.5, the notation $L_l N_n$ is employed to signify that the node is the n-th computing node of the l-th level, while M_m indicates the m-th memory node. For any $L_1 N_n$, we construct a memory access tree, where $L_1 N_n$ serves as the root node, the memory nodes function as the leaf nodes, and buses that interconnect different nodes form the edges, considering the topology of the multi-level NUMA system, all other computing nodes are interconnected by edges. The access tree depicted in the Fig.5 is built upon $SCCL_1$ and its associated access path. After completing the construction of the memory access forest, TAPMM generates architecture-specific code. Fig.7 shows the structure of the memory access node. We use it to build the memory access forest, and then compile it to generate a runtime dependency library.



(a) Last weights (b) Update bandwidth (c) Update bottleneck (d) Traffic merge (e) Update weights

Figure 6: Weights refresh

```
typedef struct {
    // number of the child nodes and pointer to them
    int children_num;
    node *children;
    // bandwidth between the node to its child nodes
    uint_64 *bandwidth;
    // address of descendant node weights array
    int merge_addr;
    // bottleneck of root to the node
    uint_64 bottleneck;
}MANode;
```

Figure 7: struct of memory access node

3.3 Online detecting bandwidth and memory allocation

TAPMM works as a runtime library during the online phase. Before the program begins execution, it initializes the entire initial state of the memory access forest. It modifies the memory allocation weights by asynchronously monitoring the system's traffic status and offers the optimal memory mapping method for both large-size and normal-size memory when an application requests memory allocation.

Update weight-matrix. We assume that all levels of the multi-level NUMA system consist of a total of x CPU nodes and y memory nodes. The node set is denoted as $N = \{n_1, n_2, ...n_x, n_{x+1}, ...n_{x+y}\}$, where $n_1, ...n_x$ represents CPU nodes and $n_{x+1}, ...n_{x+y}$ represent memory nodes. The set of child nodes owned by each node is defined as $C = \{C_1, C_2, ..., C_x, \Phi, ...\Phi\}$, for the i-th node n_i and its child node n_c, the bandwidth between them is denoted as BW_{ic}. At a specific time t, traffic state on each bus can be obtained through HPE, and the real-time bandwidth on the bus is calculated based on the traffic-bandwidth model. Assuming that at a particular moment, the weight of the memory node n_i is w_i, we can determine that $\sum_{n=n+1}^{n+m} w_n = 1$. The weight of the parent node n_p is defined as $w_p = \sum_{n_c \in C_p} w_c$. If we intend to allocate S bytes, the i-th node either provides or transmits $S \times w_i$ (if the i-th node is a memory node, it provides the bytes; otherwise, if it is a CPU node, the bytes pass through it). We denote the time required for the i-th node to retrieve S bytes of memory parallelly from its child nodes as $T(n_i)$, then the expression as follows:

$$T(n_i) = \max\{\max_{n_c \epsilon C_i} \frac{S \times w_c}{BW_{ic}}, \max_{n_c \epsilon C_i} T(n_c)\} \quad (1)$$

From the formula, it is evident that the time required to retrieve S bytes of memory depends on the bus with the smallest bandwidth on the memory access path. Thus, TAPMM uses a graph traversal algorithm to identify memory access bottlenecks. As shown in Algorithm.1, TAPMM utilizes depth-first search to pinpoint memory access bottlenecks and update the memory mapping weight-matrix. The update step of a memory access tree in the memory access forest is illustrated in Fig.6. The entire procedure is segmented into five steps, with the label in the upper right corner of the subgraph indicating the status of the purple node, and the red number signifying the value updated at each step. During the *traffic merge* stage shown in Fig.6(e), the algorithm detects bandwidth bottleneck and adjusts the weights information of the child nodes. This iterative process continues until every tree in the forest has been updated.

Algorithm 1: $Refresh(n, bw, bl)$

```
input  : struct of MANode $n$, bandwidth between node $n$
         and its parent node $bw$, bottleneck of parent node
         to root $bl$
output : Optimal weights array $W$
$n.bottleneck \leftarrow$ Min($bw$, $bl$)
$d\_num \leftarrow 0$ // record number of descendant leaf node
$b\_sum \leftarrow 0$ // record sum of descendant node's bottleneck
if $n.children\_num > 0$ then
    for $i \leftarrow 0$ to $n.children\_num$ do
        $d, b \leftarrow$Refresh($n.children[i]$, $n.bandwidth[i]$,
          $n.bottleneck$)
        $d\_num \leftarrow d\_num +$ Max($d$, 1)
        $b\_sum \leftarrow b\_sum + b$
    // merge traffic from descendant node
    if $b\_sum > n.bottleneck$ then
        for $k \leftarrow n.merge\_addr$ to $n.merge\_addr + d\_sum$
          do
            $W[k] \leftarrow W[k] \times \frac{n.bottleneck}{b\_sum}$
else
    $W[n.merge\_addr] \leftarrow n.bottleneck$
return $d\_num$, Max($b\_sum$, $n.bottleneck$)
```

Placement policy. When an application requests memory, TAPMM provides the optimal memory placement based on the requested size. For huge-size memory allocations, TAPMM determines the CPU node where the thread is located, identifies the corresponding memory access tree, and calculates memory placement information using the weights array associated with tree. Subsequently, the

Table 1: Benchmarks and their memory access information

case	Bandwidth requirements		Memory footprint	
	read(MB/s)	write(MB/s)	huge(MB)	normal(KB)
BT.C	13760.43	7339.75	689.60	685.20
CG.C	6062.83	248.28	906.94	7711.71
EP.C	3696.95	3139.25	0.00	660.21
FT.C	25367.57	21130.32	5130.00	659.96
IS.C	13804.32	8256.18	0.00	654.18
LU.C	5862.06	4197.79	558.25	1109.55
MG.C	8575.59	3583.2	3408.81	683.47
SP.C	18640.06	11064.7	722.44	685.20
DC.W	2212.83	797.35	305.18	19847.89
UA.C	15159.03	7420.85	444.82	41259.89
XSBench	15948.61	3064.93	5678.79	63287.06
BFS	12338.62	2221.04	3520.44	623.36

mbind system call is utilized to map the memory to the corresponding node.

For normal-size memory requests, TAPMM first attempts to allocate in the chunk cache. In cases where the chunk cache is depleted, it applies for new memory from the operating system. Given that small memory is particularly sensitive to latency, TAPMM avoids crossing high-level NUMA domain, which can introduce higher delays. Instead, it traverses the memory nodes within the L2 domain where the thread requesting memory is situated and selects the node with the highest weight for memory mapping.

4 EVALUATION

4.1 Methodology

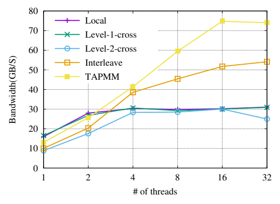
Platform and tools. We evaluate TAPMM on a two-level NUMA system with two **Hisilicon Kunpeng 920 CPU @ 2.6Ghz** processors with 2 SCCLs in each and 256GB of RAM, the server running aarch64 with kernel version is 5.4.0. We use *numactl* to manage the page placement in comparative experiments.

Applications and inputs. We use micro-benchmark for bandwidth testing, and use NPB, XSBench[12] and BFS[13] to conduct real application experiments. For NPB testing, we set the input scale to class=C (DC only supports class=W input). XSBench is the key calculation kernel function of the Monte Carlo neutron transport algorithm, and BFS is a key workload in the Graph Pattern Mining program. The bandwidth requirements of the above mentioned applications (tested on Intel server by using PCM) and memory footprint information are shown in the Table1.

Implementation and Comparison. The tested program does not need to be modified. As the current optimization tools does not support Arm-based systems, we compare TAPMM with strategies provided by linux.

- AutoNUMA[15]: It can achieve page-level hot memory identification and memory migration, utilizing the first-touch mechanism for page initialization.
- Local: When memory is allocated, it gets mapped to the node where the thread requesting memory is running.
- Interleave: The memory pages are evenly interleaved and distributed across all memory nodes. This is also the placement method that can obtain the maximum memory bandwidth from the perspective of L1 NUMA domain.

4.2 Evaluation results



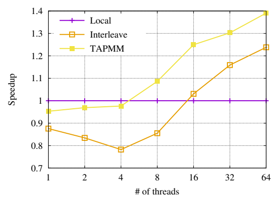


Figure 8: Bandwidth **Figure 9: Scalability**

Micro-benchmark for bandwidth analysis. We employ a sequential memory access micro-benchmark to assess bandwidth variations under different placement strategies. We use *numactl* to fix threads in a specific SCCL, and place the memory in the local node, L1-cross node (the memory node to which another SCCL in the same socket), L2-cross node (the memory node of the remote socket), and Interleave, the result is shown in the Fig.8. Because the interchip bus bandwidth is much larger than the memory controller bandwidth, the local and L1-cross bandwidth performance are almost the same. As the number of threads increases, so do the bandwidth requirements. With a small number of threads, specifically when bandwidth demands are modest, local and L1-cross configurations exhibit optimal performance. However, when the threads count reaches or exceeds 4, leading to a bottleneck in memory access bandwidth, TAPMM outperforms others, achieving up to 1.44× the bandwidth of Interleave.

Single-application workloads. We test different placements in an exclusive environment. Exclusive means that there is almost no additional traffic in the system when applications start running, and all traffic is generated during the executions. The results are depicted in the Fig.10, it can be seen from the Table.1 that EP, CG and DC are all computationally intensive programs, and modifying

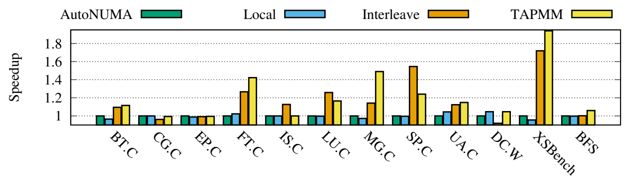


Figure 10: Single-application workloads

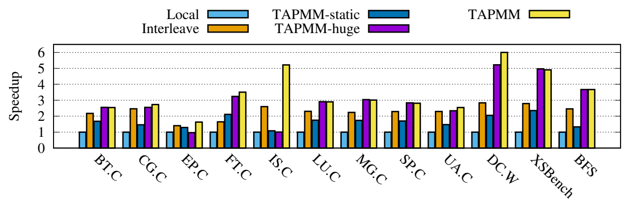


Figure 11: Multi-application workloads

the placement method will have almost no impact on their performance. Because the running time is short, AutoNUMA's migration mechanism barely works, and its first-touch mechanism causes the memory placement location to be almost the same as that of the Local strategy, resulting in the two placement methods having almost the same performance. Among programs with high memory footprint requirements, TAPMM is always the best placement strategy. Notably, on FT, which has the highest bandwidth demands, it achieves acceleration ratios of 1.4× and 1.12× compared to the AutoNUMA and Interleave placement strategies, respectively.

Multi-application workloads. In order to simulate the real operating environment and ensure that the traffic status of the system is the same for every benchmark, we sample the traffic status on the server running MySQL and then utilize *memhog* to replay the traffic to simulate running MySQL and benchmarks simultaneously. We complete the multi-application workloads experiment and the result are shown in the Fig.11. In addition to the Local and Interleave placement strategies, we also test TAPMM-static, which is a placement strategy without traffic awareness. TAPMM-huge is an optimization strategy that only enables optimization of huge memory, and for normal memory the default placement strategy is used. Judging from the results, TAPMM-static suffers performance damage because nodes with higher bandwidth are heavily accessed by MySQL. Traffic-aware strategies can effectively avoid this problem. Because EP and IS do not require huge memory, they can only achieve better results when normal memory optimization is turned on. Experiments have proven that under the load of multi-application workloads, TAPMM effectively mitigates memory access competition, resulting in an acceleration ratio ranging from 1.15× to 2.12× when compared with the existing optimization strategy.

Scalability. For the scalability experiment, we choose the FT benchmark, which has higher bandwidth and footprint requirements, and complete the experiment under exclusive conditions. As shown in Fig.9, we use Local strategy as the baseline. When the number of threads is small, the Local method is the optimal placement. As the number of threads exceeds 4, TAPMM begins to outperform the Local placement. When the number of threads surpasses 16, the Interleave placement strategy becomes superior to Local. The overall experimental outcomes align with the bandwidth test results, demonstrating that TAPMM exhibits better scalability in most cases.

Overhead. We conduct an overhead test on TAPMM and observe that, compared with the original *jemalloc*, the average time to allocate huge memory increases by 1.4×. Due to the presence of cache, there is no significant increase in overhead for allocating normal memory. In the DC which is the most frequently request memory benchmark, our test reveal that the number of instructions required to run TAPMM account for only 0.09% of the entire test case, compared with 0.08% of *jemalloc*, it only brings about 0.01% overhead, so we believe that TAPMM's overhead is far less than the performance gain it brings.

5 CONCLUSION

This paper is the first research on traffic-aware memory optimization on Arm-based NUMA systems. The innovation lies in considering the multi-level NUMA memory access challenges caused by chiplet technology, and designing a topology and traffic aware memory allocator. The allocation method TAPMM uses the memory access forest to find memory access bottlenecks and selects the best placement method according to different memory sizes. TAPMM can sense traffic in the actual production environment and bring great performance improvements to applications. We believe that TAPMM's ability to sense NUMA system topology and traffic can also play a role in more complex memory structures such as CXL in the future.

6 ACKNOWLEDGEMENT

This work was supported in part by the National Key R&D Program of China (Grant No. 2023YFB3002702), the Programs of NSFC (Grant Nos. U21A20461, 62321003, 62172157, 92055213, 62227808), the Major Projects of Xiangjiang Laboratory (22xj01011), and the Programs of Hunan Province (Grant Nos. 2024JJ2026, 2023GK2002).

REFERENCES

[1] Dashti et al. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. *ACM SIGPLAN Notices* (2013).
[2] D. H. Bailey et al. 1991. The NAS Parallel Benchmarks—summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*.
[3] Gureya et al. 2020. Bandwidth-aware page placement in numa. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.
[4] Hassan et al. 2015. Software-managed energy-efficient hybrid DRAM/NVM main memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*.
[5] Jing Xia et al. 2021. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* (2021).
[6] Lepers et al. 2015. Thread and memory placement on {NUMA} systems: Asymmetry matters. In *2015 USENIX annual technical conference (USENIX ATC 15)*.
[7] Laso et al. 2022. CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters. *Future Generation Computer Systems* (2022).
[8] Maruf et al. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery.
[9] Raybuck et al. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery.
[10] Sánchez Barrera et al. 2020. Modeling and optimizing numa effects and prefetching with machine learning. In *Proceedings of the 34th ACM International Conference on Supercomputing*.
[11] Shiwu Lo et al. 2023. RON: One-Way Circular Shortest Routing to Achieve Efficient and Bounded-waiting Spinlocks. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*.
[12] Tramm et al. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*.
[13] Xuhao Chen et al. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* (2020).
[14] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.
[15] RedHat. 2014. AUTOMATIC NUMA BALANCING. (2014). https://access.redhat.com/