Asymmetry-aware Page Placement for Contemporary NUMA Architectures

David Gureya
INESC-ID Lisboa/IST
dgureya@gsd.inesc-id.pt

Rodrigo Rodrigues
INESC-ID Lisboa/IST
rodrigo.miragaia.rodrigues@tecnico.ulisboa.pt

Paolo Romano
INESC-ID Lisboa/IST
romanop@gsd.inesc-id.pt

Pramod Bhatotia
University of Edinburgh
pramod.bhatotia@ed.ac.uk

Vivien Quéma
Grenoble INP
vivien.quema@grenoble-inp.fr

Joao Barreto
INESC-ID Lisboa/IST
joao.barreto@tecnico.ulisboa.pt

ABSTRACT

Although new thread placement approaches for asymmetric NUMA systems have recently emerged, it is perhaps surprising to observe that the standard techniques for page placement still rely on the obsolete assumption of a symmetric architecture. This paper proposes a novel approach, called *Asymmetry-Aware Page Placement* (AAPP), for near-optimal placement of shared pages in asymmetric NUMA systems. Given a memory-intensive application clustered on a set of nodes, AAPP builds an approximated model of the potential throughput of the application and calculates a target near-optimal weight distribution to be adopted when interleaving shared pages.

ACM Reference Format:
David Gureya, Rodrigo Rodrigues, Paolo Romano, Pramod Bhatotia, Vivien Quéma, and Joao Barreto. 2018. Asymmetry-aware Page Placement for Contemporary NUMA Architectures. In *Proceedings of* . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Non-uniform memory access (NUMA) architectures have quickly become the norm in high-end servers. In a NUMA system, CPUs and memory are organized as a set of interconnected nodes, where each node typically comprises one or more multicore CPUs and a memory controller that provides access to a partition of the global physical address space. The non-uniform memory access nature stems from this organization – the latency of data access depends on which node the accessing thread runs and which node the target physical page resides on.

When compared to symmetric multiprocessing (SMP) architectures, the multi-node nature of NUMA architectures poses two new fundamental questions: where should the threads and the pages of each application be placed. Due to the underlying overheads and capacity limits, sub-optimal decisions to these questions can easily lead to huge impacts on the global throughput of applications, especially when these are strongly memory-intensive [6].

However, finding a proper page and thread placements in NUMA systems is far from trivial. On one hand, it depends on the complex and dynamic application-specific performance patterns. On the other hand, it needs to take into account increasingly intricate and asymmetric NUMA topologies. In fact, contemporary NUMA architectures are characterized eminently asymmetric since different pairs of nodes are interconnected at distinct bandwidths and latencies (not all nodes are connected by a single hop). To further exacerbate this asymmetry, the effective memory bandwidth at which a given page is accessible to some thread (either local or remote) varies significantly depending on the contention from other nodes trying to access pages on the same memory controller.

As some recent studies have shown [13, 17], the above sources of asymmetry render traditional placement schemes (originally designed under the symmetry assumption) inappropriate for the emerging NUMA architectures. Hence, the state-of-the-art needs to be deeply rethought to embrace the emerging asymmetry. The research community has recently started to respond to such a quest, proposing new thread placement approaches for asymmetric NUMA topologies [13, 17].

However, it is perhaps surprising to observe that the standard techniques for page placement still rely on the obsolete assumption of a symmetric architecture. With some variations, the general rule of thumb is that shared pages (i.e., pages that are accessed by all threads) should be interleaved across a set of nodes. To the best of our knowledge, most schemes restrict such set of nodes to the nodes where the threads of the application are clustered [6, 10, 13]. Our proposal, AAPP stretch out to a larger set of nodes that also includes CPU-idle nodes.

One key similarity that all these schemes share is that all interleave pages *uniformly* across nodes where the threads of the application are clustered. However, since the topology interconnecting such nodes is often asymmetric, the resulting performance is sub-optimal. As an example, consider the memory-intensive multithreaded application Streamcluster from PARSEC [2] running on 4 nodes of a larger asymmetrical NUMA machine. As Figure 1 depicts, uniformly interleaving this application's pages across these nodes (*Worker-Only Uniform Interleave (WOUI)*) will yield a lower throughput than uniformly interleaving in a larger set of nodes (*AAPP-Uniform*). However, both approaches are clearly sub-optimal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

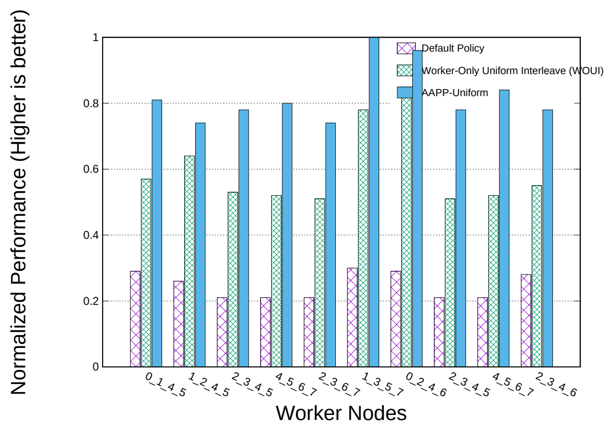

Figure 1: Performance of Streamcluster with 24 threads on 4 nodes under WOUI policy, AAPP-Uniform policy (best policy for this benchmark) and Default (First-Touch) policy on our experimental system

as they build on the false promise of a symmetric topology. Intuitively, it should follow a weighted interleaving policy where better connected nodes are favoured (with a higher portion of pages) than poorly connected ones.

Still, as we explain later in the paper, determining the optimal weight distribution is challenging for a number of reasons:

- First, many parallel applications have a large thread count, so their threads are clustered across multiple nodes. Optimizing a weighted page interleaving in such multi-node applications is not trivial, as the optimal weighted page interleaving will be different if considered from the perspective of threads in distinct nodes. Hence, the optimal weight distribution is usually determined as a compromise solution that, despite being sub-optimal from each node's perspective, is system-wide optimal.
- Second, if the interconnect topology is static, the effective interconnect capacities are transformed by the simple fact that a subset of a machine's nodes will be running application threads (while the remaining nodes may have their CPUs idle). In fact, the bandwidth that a given node offers to remote threads (accessing the node's pages) is lower when that node is hosting local threads, when compared to a node whose CPUs are idle.
- Finally, even when the capacity of each interconnect link and memory controller are well-known a priori, their precise behavior is hard to predict in practical situations where multiple threads contend for such resources [17, 18]. This includes contention across local (and remote) threads when accessing data on the same memory controller [18], across local threads when competing for the same interconnect link [13], as well as inter-node cache-coherency invalidation overheads [4]. High-precision models of such complex interferences are not usually available, hence simpler heuristic approximations are the only option.

This paper focuses on the problem of determining optimal weighted page interleaving in contemporary NUMA systems. To the best of our knowledge, this paper is the first one to address this problem.

The paper proposes a novel approach, called Asymmetry-Aware Page Placement (AAPP), for near-optimal placement of shared pages in asymmetric NUMA systems. Given a memory-intensive application clustered on a set of nodes, AAPP builds an approximated model of the potential throughput of the application and calculates a target near-optimal weight distribution to be adopted when interleaving shared pages. AAPP is fully implemented as a variant of `mmap` and `malloc`, and thus can be used transparently by any application, with no changes to the OS kernel.

An experimental evaluation with a representative set of memory-intensive workloads from PARSEC [2] shows that AAPP is able to achieve up to 1.4x speedups when compared to traditional uniform page placement policies for NUMA systems.

2 AAPP

This section describes AAPP. We start by introducing the system model assumptions, then present the AAPP model. Finally, we describe how the near-optimal weighted interleave configurations that AAPP recommends are enforced on the effective page to node mappings.

2.1 System Model

We assume a NUMA system comprising a set of N nodes, $n_0, n_1, ..., n_{N-1}$. Each node contains one or more multicore CPUs, which together are able to run up to T hardware threads. Furthermore, each node includes a memory controller, which maintains a partition of physical pages that constitute the global physical address space of the system. We assume that the nodes of the system are locally homogeneous; i.e., their CPU and memory controller have similar characteristics.

Any thread (running at any node) may read and write to pages that reside on the local node's memory, and on any remote node's memory. In the latter case, the read/write request is sent through the interconnect, which provides full connectivity among all nodes.

In contrast to the node homogeneity, the interconnect topology is asymmetric. More precisely, a thread running on a given node will observe different bandwidths and latencies when reading from or writing to the memory at different nodes. The most obvious difference is between local and remote accesses. Among remote accesses, the differences in bandwidth are explained by the heterogeneity among interconnect links, which can even have distinct bandwidths at each communication direction. The latency of remote accesses can also change since some nodes are directly connected, while others communicate through multiple-hop paths.

2.2 Algorithm

The goal of AAPP is to find a near-optimal page placement for multi-threaded applications on the NUMA system described above. We consider that the application has t threads and all run a homogeneous workload; i.e., the memory access patterns are similar among all the threads of the application.

Within the application's address space, we distinguish between thread-private pages and shared pages. Thread-private pages are

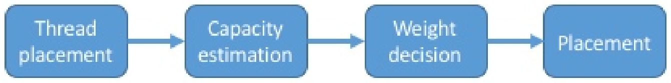

Figure 2: Stages of AAPP

trivially placed on the same node as the corresponding threads. Shared pages, in contrast, are harder to place optimally, as they are accessed by threads residing on different nodes, through diverse interconnect links. Hence, the focus of AAPP is on finding the optimal placement for the shared pages.

AAPP follows a 4-stage approach as depicted in Figure 2. Firstly, the *thread placement stage* resorts to some state-of-the-art thread placement scheme to decide at which *worker nodes* threads should run. Secondly, the *capacity estimation stage* decides the subset of nodes where pages should be placed and estimates node-to-worker capacities for such a subset. Thirdly, the *weight decision stage* uses the information obtained previously to compute optimal interleaving weights for each node where pages are to be placed. Finally, the *placement stage* place pages according to the previous recommendation. We now describe each component.

2.2.1 *Thread placement stage.*

We rely on some state-of-the-art thread placement tool (like [13, 17]) to decide the thread-to-node mapping. As a result, we expect that the t threads of the application are evenly balanced across a subset of well-connected nodes (mainly inter-connected by direct and high-bandwidth links). We designate these nodes as *worker nodes*. In general, any thread placement algorithm deemed suitable can be plugged into the AAPP.

2.2.2 *Capacity estimation stage.*

The first goal of this stage is to determine the subset of nodes where pages should be placed. The common best practice is to interleave the shared pages on the nodes which have threads running. We call this Worker-Only Interleaving (WOI). This policy is employed in most state-of-the-art placement solutions for NUMA systems, such as [6, 10, 13]. WOI follows the intuitive heuristic that, by placing shared pages in worker nodes, we ensure that at least some threads will access each shared page locally; since the worker nodes will typically be mutually well-connected, the toll paid by remote threads will be also attenuated. The remaining (non-worker) nodes are simply not chosen to hold pages because any page placed would be remote to every thread.

While counter-intuitive, it is easy to show that, when applications are sufficiently memory intensive, placing shared pages on *both* the non-worker and worker nodes unlike WOI can lead to significant performance benefits. We call this Global Interleaving (GI). The reason for the advantage of GI is that, as studied by recent works [13, 17], interconnect congestion becomes the decisive performance factor. In fact, for some applications, congestion itself has a higher impact in performance than access latency [13]. In other words, wisely balancing pages among NUMA nodes in order to maximize the available memory bandwidth becomes more important than co-locating shared pages and worker nodes.

Therefore, the first decision that AAPP takes is whether WOI or GI should be chosen for the given application. To take this decision, AAPP profiles the application under both options (WOI and GI) for a fixed profiling time, using a uniform interleave page placement. Based on the observed performance, AAPP selects the best-performing policy, WOI or GI. In each profiling window, AAPP measures the memory capacity (i.e., access bandwidth) from each node holding pages to each worker node. More precisely, given a worker node W, we determine the capacity (bytes per second) when reading from pages in a target node M. We refer to this capacity as the Capacity from node M to node W i.e. $Capacity(W \leftarrow M)$. This is achieved by hardware performance counters. The resulting map of node-to-worker capacities will be the main input to the next stage.

2.2.3 *Weight decision stage.*

We consider a simple example inspired from the example discussed in [19] with a memory-intensive multi-threaded application whose execution time is mainly determined by the time to transfer a given dataset (S) from the memory subsystem to the CPUs.

Let's first consider the set of threads running on a given worker node, N_0. Threads at N_0 need to read S bytes from shared pages that are interleaved at a set of N nodes (selected in the previous stage) in a weighted (i.e., non-uniform) fashion, where the portion (weight) of pages at a given node i is given by w_i. Naturally, $\sum_{i=0..N-1} w_i = 1$.

Hence, the time spent for the worker threads in N_0 to read the fraction of S placed in node i is approximated by Equation 1:

$$TimeToReadFrom(N_i) = (S * w_i)/Capacity(N_0 \leftarrow N_i) \quad (1)$$

If we now take into account that the worker threads in N_0 are actually reading from pages in different nodes, concurrently, the time that we need to wait until all bytes in S are completed can be approximated by a parallel race between the read access batches from each node, as denoted by equation 2. A corollary of this is that the total execution time of threads at N_0 is determined by the batch of reads from a given node that takes the longest time.

$$TotalTime = max_i(TimeToReadFrom(N_0 \leftarrow i)) \quad (2)$$

Our goal is now to find weights that minimize the $TotalTime$. It is easy to show that the solution to this optimization problem is given by Equation 3.

$$w_i = Capacity(N_0 \leftarrow N_i)/sum_k(Capacity(N_0 \leftarrow N_k)) \quad (3)$$

Finally, the previous optimization can be generalized to multiple worker nodes using Equation 4.

$$w_i = MinCapacity(i)/sum_k(Capacity(N_0 \leftarrow N_k)) \quad (4)$$

where $MinCapacity(i)$ denotes the lowest capacity from node i to any worker node. For space limitations, we omit the proof of Equation 4.

2.2.4 *Placement stage.*

After devising a target weight distribution, AAPP simply needs to ensure that the application's shared pages are mapped to nodes according to that distribution. To achieve this, AAPP currently employs a portable memory allocation library (with an API similar to the *ptmalloc* library [7]) that allows the programmer to allocate memory in pages that are interleaved according to a user-defined weight distribution.

3 PRELIMINARY EVALUATION

Our goal is to evaluate how much performance advantage AAPP introduces when compared to the usual page placement policy, namely *Worker-Only Uniform Interleave (WOUI)*. In WOUI, memory

,
,

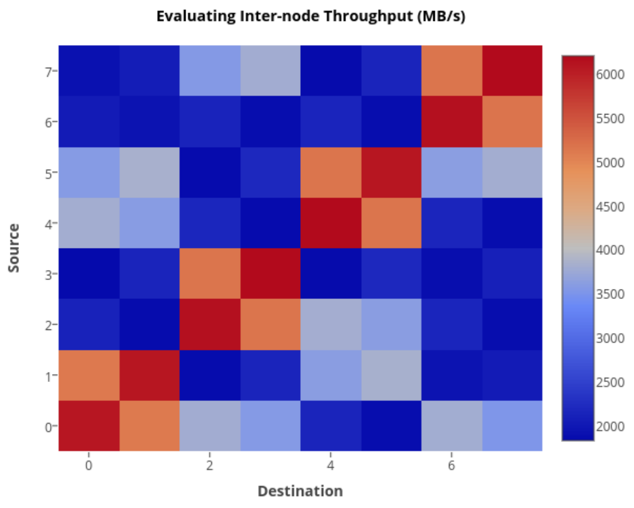

Figure 3: Node-to-node remote and local capacities in the evaluated machine (MB/s)

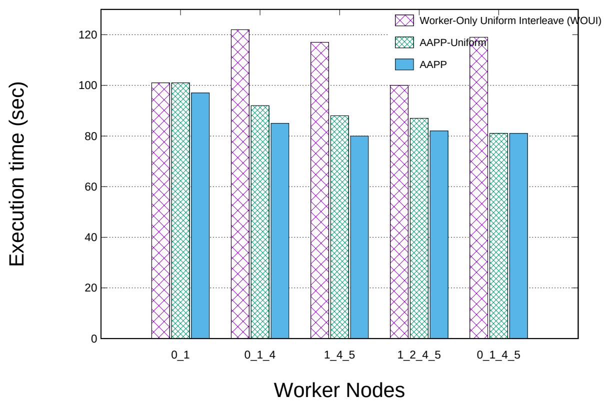

Figure 4: Performance of Streamcluster with 24 threads on 4 worker nodes, with 18 threads on 3 worker nodes and with 12 threads on 2 worker nodes under WOUI, AAPP-Uniform and AAPP

pages are spread evenly across the chosen subset of worker nodes. This is the most widely used policy to mitigate congestion.

We also compare WOUI with a simplified variant of AAPP, denoted AAPP-Uniform, where we disabled the weight decision stage. Thus, AAPP-Uniform after opting between WOI and GI, it skips the following stage and trivially assigns uniform weights to each node.

Our preliminary results were taken in an AMD Opteron(tm) Processor 6168 with eight nodes, each hosting six cores. Each CPU is locally connected with a 16GB memory node. The server system uses the Linux kernel (version 3.16.0). Figure 3 shows the bandwidth from one node (source) to all the other nodes (destination) on our system. As it is evident, interconnect links exhibit many asymmetric disparities.

We used an instrumented version of Streamcluster, a data-intensive PARSEC[2] benchmark suite to show how data placement affects performance. For data size, the *"native"* input set was used. This benchmark was chosen because of its memory-intensiveness. We consider the Streamcluster benchmark under different thread placements for 12 threads (pinned on 2 worker nodes), 18 threads (3 worker nodes) and 24 threads (4 worker nodes). Naturally, the more threads are spawned, the highest the memory demand, thus the application becomes more bandwidth-intensive. The pinning of benchmark processes is done at run-time using the numactl tool[11]. The mapping of shared memory pages to nodes is done using our memory allocation library. We measure the average execution time over 5 runs.

For space limitations, we omit a broader evaluation with other types of benchmarks.

Figure 4 shows the performance difference in completion time of Streamcluster under different memory management policies and different thread placements. Here, we picked the best well connected nodes for thread placement, which is a common strategy [13]. The results presented in Figure 4 show that, with a few exceptions, AAPP achieves substantial performance gains relatively to WOUI, with an average speedup of 1.4x. This is because AAPP dramatically reduces memory controller and interconnect congestion by alleviating the load imbalance. AAPP also outperforms AAPP-Uniform as it takes into account how the nodes of a system are connected (asymmetry-aware).

The results show that the advantage of AAPP increases with a combination of two factors: i) the application exhibits higher memory demand; and ii) the memory capacity between the nodes holding pages and the worker nodes are more unbalanced.

The first factor (memory demand) allows AAPP opt for a global page placement, instead of the common practice of restricting page placement to the worker nodes. The gains of this smarter strategy are evident when we compare the performance of AAPP-Uniform and WOUI on the experiments with 3 and 4 worker nodes (18 and 24 threads, resp.). In contrast, the experiment with fewer threads (hence, lower memory demand) reflects a case where reaching out for the non-worker nodes for placing pages is not advantageous, hence the performance of AAPP becomes closer to the trivial WOUI.

The impact of the second factor (capacity asymmetry) is evident when we compare the performance of AAPP with AAPP-Uniform. In fact, the scenarios which yielded the highest bandwidth asymmetry (between the nodes holding pages and the worker nodes) are precisely the ones where an appropriate weight selection leads to stronger speedups. At the extreme of high asymmetry, like the 0-1-4 and 1-4-5 experiments, AAPP achieves the highest gains over the uniform policy. In contrast, the 0-1 scenario, where pages are placed only at the worker nodes, is only slightly asymmetric since the two worker nodes share very similar high-bandwidth links between them; hence, AAPP will effectively select a uniform weight distribution in this case.

Table 1 shows that the pages are allocated across the nodes in proportion to their node-to-worker capacities.

Table 1: Memory allocation ratio for Worker Nodes 0, 1 and 4

Node	Capacity (GB/s)	Weight (%)
0	4.4	21.2
1	4.2	20.2
2	1.7	8.2
3	1.4	6.7
4	3.3	15.9
5	2.7	13.0
6	1.7	8.2
7	1.4	6.7

Overall, our evaluation demonstrates that asymmetry-aware policies like AAPP are promising in that they provide significant performance gains over conventional policies like WOUI with memory-intensive benchmarks.

4 RELATED WORK

Optimizing thread and memory placement on NUMA systems has been extensively studied [4, 6, 12–16]. The most common strategy for thread and data placement in NUMA is locating data as close as possible to cores. However, dynamic conditions of the architecture resources such as congestion on the interconnect and the memory controller often lead to other allocation decisions. For instance, when the nodes are connected by links of different bandwidth, we must consider not only whether the threads and data are placed on the same or different nodes, but how these nodes are connected[13]. As another example, it turns out that interleaving within the worker nodes dramatically reduces memory controller and interconnect congestion by alleviating the load imbalance and mitigating traffic hot-spots. This results in improved memory latency[6]. Additionally, when a memory controller linked to local memory is congested, placing data in remote memory instead of local memory could yield better performance [14].

As a general comparison, we observed that most related work either performs thread or data placement, but not both of them together. Thread placement mechanisms such as [17], are not able to reduce the amount of remote memory accesses on NUMA architectures. On the other hand, data placement mechanisms such as [6], are not able to reduce cache misses or correctly handle the mapping of shared pages. Existing mechanisms that perform both placements together have several disadvantages. For instance, [13] uses a simpler/best-effort heuristics to find the best thread placements which might not be optimal and also only considers interconnect asymmetry as the only NUMA memory resource. On the other hand, Cruz et al.'s solution[5] requires hardware support and doesn't consider the NUMA asymmetry. Several other proposals require specific architectures, APIs or programming languages to work, limiting their applicability. For instance, Pandia [8] assumes a fully-connected symmetric interconnect which removes some of the complexities observed in this research. Prior works such as the ones presented in [1, 19], investigate page placement strategies within heterogeneous memory systems, thus they are insightful to our work.

Recent works such as [3, 9] are exclusively dedicated to discovering NUMA topologies, hence they complement our work.

5 CONCLUSION

Although new thread placement approaches for asymmetric NUMA systems have recently emerged, it is perhaps surprising to observe that the standard techniques for page placement still rely on the obsolete assumption of a symmetric architecture. This paper proposes AAPP, a novel approach for near-optimal placement of shared pages in asymmetric NUMA systems. Our initial results suggest that there is an unexplored opportunity in incorporating the asymmetry of NUMA topologies when placing pages of memory-intensive applications.

6 ACKNOWLEDGEMENTS

This work was supported by national funds through Fundação para aCiência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

REFERENCES

[1] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. *SIGPLAN Not.* 50, 4 (March 2015), 607–618. DOI:http://dx.doi.org/10.1145/2775054.2694381

[2] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[3] Georgios Chatzopoulos, Rachid Guerraoui, Tim Harris, and Vasileios Trigonakis. 2017. Abstracting Multi-Core Topologies with MCTOP. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17).* ACM, New York, NY, USA, 544–559. DOI:http://dx.doi.org/10.1145/3064176.3064194

[4] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. 2015. LIRA: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '15).* ACM, New York, NY, USA, Article 2, 8 pages. DOI:http://dx.doi.org/10.1145/2768405.2768407

[5] E. H. M. Cruz, M. Diener, M. A. Z. Alves, L. L. Pilla, and P. O. A. Navaux. 2014. Optimizing Memory Locality Using a Locality-Aware Page Table. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing.* 198–205. DOI:http://dx.doi.org/10.1109/SBAC-PAD.2014.22

[6] 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. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 381–394. DOI:http://dx.doi.org/10.1145/2490301.2451157

[7] Wolfram Gloger. 2006. A fast, memory-efficient implementation of malloc for Unix systems. (March 2006). Retrieved March 28, 2018 from http://www.malloc.de/en/

[8] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: Comprehensive Contention-sensitive Thread Placement. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17).* ACM, New York, NY, USA, 254–269. DOI:http://dx.doi.org/10.1145/3064176.3064177

[9] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. 2016. Machine-aware Atomic Broadcast Trees for Multicores. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16).* USENIX Association, Berkeley, CA, USA, 33–48. http://dl.acm.org/citation.cfm?id=3026877.3026881

[10] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. 2015. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15).* USENIX Association, Berkeley, CA, USA, 263–276. http://dl.acm.org/citation.cfm?id=2813767.2813787

[11] Andi Kleen. 2002. numactl. (March 2002). Retrieved March 28, 2018 from https://linux.die.net/man/8/numactl

[12] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12).* USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=2342821.2342826

[13] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15).* USENIX Association, Berkeley, CA, USA, 277–289. http://dl.acm.org/citation.cfm?id=2813767.2813788

[14] Zoltan Majo and Thomas R. Gross. 2011. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)*. ACM, New York, NY, USA, Article 12, 10 pages. DOI: http://dx.doi.org/10.1145/1987816.1987832

[15] Zoltan Majo and Thomas R. Gross. 2012. Matching Memory Access Patterns and Data Placement for NUMA Systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 230–241. DOI: http://dx.doi.org/10.1145/2259016.2259046

[16] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. 2013. Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 306–317. DOI: http://dx.doi.org/10.1109/HPCA.2013.6522328

[17] W. Wang, J. W. Davidson, and M. L. Soffa. 2016. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 419–431. DOI: http://dx.doi.org/10.1109/HPCA.2016.7446083

[18] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa. 2014. DraMon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 380–391. DOI: http://dx.doi.org/10.1109/HPCA.2014.6835948

[19] Seongdae Yu, Seongbeom Park, and Woongki Baek. 2017. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 18, 10 pages. DOI: http://dx.doi.org/10.1145/3079079.3079092