Bandwidth-Aware Page Placement in NUMA

David Gureya, João Neto, Paolo Romano, Rodrigo Rodrigues, João Barreto
INESC-ID, Instituto Superior Técnico, University of Lisbon, Portugal

Vivien Quema
Grenoble INP/ENSIMAG, France

Pramod Bhatotia
University of Edinburgh, UK

Reza Karimi
Emory University, USA

Vladimir Vlassov
KTH Royal Institute of Technology, Sweden

***Abstract*—Page placement is a critical problem for memory-intensive applications running on a shared-memory multiprocessor with a non-uniform memory access (NUMA) architecture. State-of-the-art page placement mechanisms interleave pages evenly across NUMA nodes. However, this approach fails to maximize memory throughput in modern NUMA systems, characterized by asymmetric bandwidths and latencies, and sensitive to memory contention and interconnect congestion phenomena.**

We propose BWAP, a novel page placement mechanism based on asymmetric weighted page interleaving. BWAP combines an analytical performance model of the target NUMA system with on-line iterative tuning of page distribution for a given memory-intensive application. Our experimental evaluation with representative memory-intensive workloads shows that BWAP peforms up to 66% better than state-of-the-art techniques. These gains are particularly relevant when multiple co-located applications run in disjoint partitions of a large NUMA machine or when applications do not scale up to the total number of cores.

I. Introduction

Parallel architectures with non-uniform memory access (NUMA) are emerging as 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 multi-core CPUs as well as one or more memory controllers. The non-uniform memory access nature stems from this organization, since the memory access bandwidth (BW) and latency depends on the node the accessing thread runs and on the node the target page resides.

When one deploys a parallel application on a NUMA system, its threads allocate and access pages that need to be physically mapped to the available NUMA nodes. This raises a crucial question: *where should each page be mapped for optimal performance?* When the application is memory-intensive, a common strategy is to uniformly interleave pages across the set of *worker* nodes, i.e., the nodes on which the application threads run. This strategy is based on the rationale that, for a large class of memory-intensive applications, BW – rather than access latency – is the main bottleneck. Therefore, interleaving pages across nodes provides threads with a higher aggregate memory BW [9]. Hereafter, let us call this strategy *uniform-workers*. This is the essential approach of recently proposed runtime libraries for NUMA systems (e.g., [9], [24]), as well as the recommended or default option for prominent database systems (e.g., [2], [4], [5]).

This paper starts by questioning the effectiveness of the *uniform-workers* strategy in contemporary NUMA systems. Two key characteristics of *uniform-workers* seem to be at odds with the systems it aims to optimize. First, *uniform-workers* places pages at symmetric ratios across worker nodes, while the BW (and latency) of contemporary NUMA architectures is typically asymmetric across nodes [9]. Second, there are important scenarios where an application's threads are clustered together on a *subset* of NUMA nodes – notably, when the application is deployed in a given node partition of a co-scheduled system [36], or when the application does not scale beyond a subset of the available cores [17], [38]. If the remaining memory nodes are idle or underused (e.g., by CPU-intensive applications), *uniform-workers* will neglect an important portion of BW. Hence, as we show in Section II, it is not surprising that the memory BW attained by *uniform-workers* is considerably suboptimal for memory-intensive applications.

To overcome these inefficiencies, we propose BWAP, a novel BW-aware page placement for memory-intensive applications on NUMA systems. In contrast to *uniform-workers*, BWAP takes the asymmetric BWs of every node into account to determine and enforce an optimized application-specific *weighted interleaving*. Our proposal is inspired by recent research for hybrid memory systems [10], [26], [35]. These works have shown that, when a CPU (or GPU [26]) is served by different memory technologies (such as NVRAM or DRAM) with differing BWs, an optimal placement is one that (proportionally) place fewer pages at the lower-BW memories.

Still, applying the same principle to the context of NUMA systems is far from trivial. While previous BW-aware proposals for hybrid memory systems relied on the premise that a given memory node provides the same BW to every core, that is no longer true in a NUMA system. The same NUMA memory node may be accessible through different BWs by different threads, depending on each thread's location within the NUMA topology. This implies that optimizing page interleaving from the perspective of a given worker node (as done by the recent proposals for hybrid systems [10], [26], [35]) will not always yield the best overall performance. Instead, the optimization problem needs to consider a complex $W \times N$ BW matrix, where W and N denote the number of worker nodes and total nodes, respectively. Furthermore, this BW matrix is particularly hard to determine accurately, since it is sensitive to interconnect congestion and local-remote contention on memory controllers phenomena which, in turn, depend on the memory demand patterns of the deployed application(s).

Hence, optimal placements are eminently application-specific.

Putting it all together, an efficient page placement for asymmetric NUMA systems requires tuning N weights, taking into account complex phenomena that depend both on the underlying NUMA architecture and the application(s) itself. A naive approach is to search through the N-dimensional space of possible weight distributions and measuring the performance of each run to find the optimal placement. This is often impractical for NUMA systems of 4 nodes and beyond, since it easily falls in the range of hours or days to find an optimized distribution of per-node weights for a given application. An alternative would be to model the BW usage of the memory system and analytically determine the optimal page placement. However, to the best of our knowledge, the most successful analytical models of memory throughput are limited to single-node scenarios [37].

BWAP tames this complexity by combining techniques from the two extremes of the solution space. In a first stage, BWAP builds a memory BW model of the target system. From this model, BWAP calculates the optimal weight distribution that maximizes the performance of a reference *BW-intensive* application. The key insight behind BWAP is that, after analytically determining that *canonical* weight distribution, that distribution can be adjusted to fit the target application by applying a scalar coefficient on each weight. In other words, BWAP reduces what in theory is an N-dimensional optimization problem to the one-dimensional problem of finding an appropriate scaling coefficient that best fits the application. To achieve this, the second stage of BWAP relies on an iterative technique, which, when the application starts, places its pages according to the canonical weight distribution; then, on-the-fly, it uses an incremental page migration scheme that adjusts the weight distribution until a new (local) optimum is found.

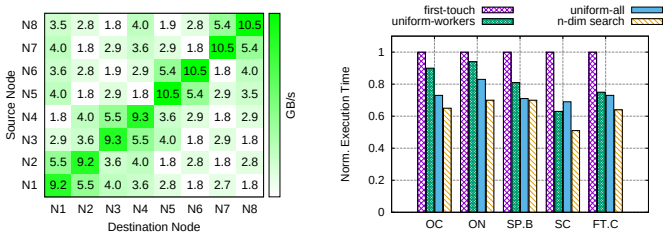
BWAP is implemented as an extension to Linux's original *libnuma*. BWAP enriches the original interface with a *bw-interleaved* policy option, which automatically determines at which set of memory nodes the application's pages should be placed and the appropriate per-node weights to balance the page interleaving across the NUMA nodes. BWAP is readily available and can be used transparently by any application, with no changes to the OS kernel.

This paper makes three main contributions:

1. We empirically study the performance of different page placement strategies on a range of memory-intensive applications on commodity NUMA machines. Our findings show that common practices that rely on the obsolete assumption of a symmetric architecture are largely suboptimal on contemporary NUMA systems.

2. We propose BWAP, an extension of the *libnuma* library that relies on a novel combination of analytical modelling and on-line iterative tuning.

3. We evaluate BWAP on a diverse set of memory-intensive workloads, showing that BWAP achieves up to $4\times$ speedup when compared to a *first-touch* policy (the default in Linux). This represents a 66% improvement over the performance gains that the most commonly used placement policies attain over the same baseline. These benefits are particularly relevant



(a) (b)

Fig. 1: (a) Node-to-node BWs (GB/s) on a 8-node AMD Opteron. (b) Performance of popular page placement schemes vs the placement found via n-dimensional search for Ocean_cp (OC), Ocean_ncp (ON), SP.B, Streamcluster (SC) and FT.C [12] (2 worker nodes, 8 threads each).

in scenarios where multiple co-scheduled applications run in disjoint partitions of a large NUMA machine or when applications do not scale up to the total number of available cores. To the best of our knowledge, this is the first proposal for BW-aware page placement in heterogeneous memory systems to be evaluated on real commodity machines, i.e. not based on simulation [10], [26], [35].

II. Motivation

To lay the groundwork, we selected different memory-intensive applications from the PARSEC [7], SPLASH [30], and NAS [12] benchmark suites and experimentally studied how different page placement strategies affect their performance. We used an 8-node NUMA machine with the asymmetric interconnect topology depicted in Figure 1a, on which each application ran stand-alone on 2 worker nodes. A larger number of architectures and baselines is evaluated in Sec. IV.

For each application, we measure its performance when its pages are mapped with Linux default *first-touch*, the common practice *uniform-workers*, and a *uniform-all* variant that uniformly interleaves pages across all nodes (both workers and non-workers). We also performed a long offline search in which we experimentally tested the performance of a large sample of weight distributions. The search used the hill climbing technique to explore the 8-dimensional space of possible solutions. Each point in the search space assigns to each memory node in the machine a weight that determines the portion of pages that will be placed at that node. The starting point in the search was *uniform-workers*. Each search covered approximately 180 iterations, taking more than 15 hours to complete for each application. For each application, the search identified a number of slightly different configurations that achieved performance within less than 3% from optimum. Thus, the values discussed next are averages over a selection of the top-10 best performing distributions for each application.

Figure 1b presents the performance of the baseline policies normalized with respect to that of hill climbing. The results suggest that, while *uniform-workers* and *uniform-all* considerably improve performance over Linux's default policy, they do not take full advantage of the BW of the underlying memory

architecture in our NUMA system. To understand why, we have studied the actual weight distributions obtained by hill climbing and drawn the following three main observations, which guided us towards the design of BWAP.

Observation 1: Pages are placed across all nodes, not just worker nodes. In many modern architectures, even applications that have moderate single-thread memory demands can easily saturate the local memory controller when multiple threads share the same node. This issue is further exacerbated if the application threads span across multiple NUMA nodes, since a fraction of accesses to pages will now be remote, thus limited by the BW of the interconnect. These results suggest that, for applications with high memory demands, page placement should not be restricted to the worker nodes; instead, the available (even if limited) BW of non-worker nodes should be harnessed by placing on these nodes a carefully selected fraction of the application's pages.

Observation 2: Pages are interleaved unevenly across nodes, with relevant cross-application variations. Every weight distribution obtained by hill climbing is highly asymmetric, with nodes with lower memory access throughput receiving fewer pages. This clearly reflects the inherent asymmetry of the underlying NUMA topology [9]. However, when we compare the best weight distributions found for different applications, we observe significant differences between applications, which we can explain by two main factors. On the one hand, the complex contention effects, both at the interconnect and memory controller [29], depend on the actual memory demand that the application places on each memory node. On the other hand, while memory BW is the dominant bottleneck of some applications, others are more sensitive to memory latency. The former benefit from exploiting the BW of remote nodes to its full extent; the latter call for approaches that, while spreading some pages remotely for increased BW, retain most pages locally for the sake of latency.

Observation 3: If one considers worker nodes and non-worker nodes separately, proportional similarities emerge among per-node weights. Let us pick two applications from our sample and compare the respective worker weights, as obtained by hill climbing, node by node. If we multiply the weights of one application by some scalar coefficient such that its aggregate worker weight becomes the same as the other application, the per-node weight variance decreases. The same occurs by following the same procedure for the non-worker nodes. In fact, if we cluster worker nodes and non-worker nodes separately and perform the above scaling on their (clustered) weight distributions, the average per-node coefficient of variation decreases by 1/3.

These key observations enable us to build a practical best-effort solution to BW-aware page placement in asymmetric NUMA systems, which we describe next.

III. BWAP: BW-Aware page Placement

Given a NUMA system and a parallel application running on a set of *worker nodes*, the goal of BWAP is to devise and enforce an efficient interleaving of the application's pages across the NUMA nodes. Since application threads access different memory nodes through potentially diverse BWs, BWAP assigns different *weights* to different nodes. A node's weight denotes the fraction of pages mapped to the node.

BWAP pipelines two key components, the *canonical tuner* and the *DWP tuner*. The first one is agnostic of the target application and runs offline. The second one is an online component that, at run-time, departs from the first component's output to reach an improved application-specific page placement.

The inner workings of each component can be summarized as follows. The canonical tuner models our knowledge of the BW of the NUMA topology. Since the effective per-node BW is sensitive to the demand that applications pose on the system, the canonical tuner assumes an extremely BW-intensive application as its reference. Its output is a set of *canonical weight distributions*, which optimize the memory throughput of the reference application. If one considers different scenarios where the application runs on different sets of worker nodes, the corresponding optimal weight distributions might also differ. Hence, the canonical tuner considers different combinations of worker nodes as input and accordingly computes the corresponding canonical weights.

As discussed in Section II, the optimal weight distribution varies substantially across distinct workloads. Therefore, the canonical weight distributions, as produced by the canonical tuner, may not be suited to other applications than the idealized BW-intensive application. Hence, for a given worker node set, the canonical weight distribution is used as a hint about the relative weight distributions to be employed for the worker set and the non-worker sets of the target application. Then, by leveraging Observation 3 (Section II), the DWP tuner converts the canonical weight distribution to one that is optimized for the target application. This is done by finding an appropriate value for a *data-to-worker proximity factor* (*DWP*), which determines how many pages will be assigned to the set of worker nodes (retaining the canonical weight relations), while the remainder will be shared within the non-worker nodes (also according to the canonical distribution). The DWP tuner achieves this through an incremental page migration mechanism that searches for a good value of *DWP* for the target application. This stage is done on-the-fly, during the first seconds of the actual execution of the application.

The following sections detail each component of BWAP.

A. Calculating canonical weights

The first step of the canonical tuner is to model the underlying machine and an idealized canonical application running on it. Based on this model, the canonical tuner determines the canonical weight distribution.

1) System model: The canonical tuner makes a set of simplifying assumptions on the underlying machine and the target applications. We show in Section IV that this model suffices for BWAP to be an effective best-effort approximation, even when the assumptions we introduce next are not entirely met in reality. We assume a cache-coherent NUMA machine comprising a set of N nodes, $\mathcal{N} = \{n_1, n_2, ..., n_N\}$, where the set of nodes as a whole is entirely managed by a single instance

of an operating system. Each node contains one or more multi-core CPUs, which globally provide C hardware threads. For simplicity, we assume that every node's local computing and memory resources (like CPU frequencies, number of cores, or local memory BW) are identical. Furthermore, each node includes one or more memory controllers providing access to the local memory of that node. We abstract the memory controllers that are local to a given node as one single-channel memory controller, whose BW is the aggregate BW of each channel/memory controller in the real topology.

Threads may read and write 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.

The interconnect topology is asymmetric, i.e., a thread running on a given node will observe different BWs and latencies, depending on the memory node it is accessing. The most obvious difference in BW and latency is between local and remote accesses. Among remote accesses, different BWs and latencies may be observed, since distinct interconnect links may have distinct BWs (possibly distinct BWs for each communication direction), and paths between nodes can differ in number of hops (some nodes are directly connected, while others communicate through multi-hop paths).

On top of the NUMA system characterized so far, our model assumes a simplified parallel application, which we refer to as *canonical application*, that uses t threads, where $t \leq C \times N$. Threads are placed at a subset of nodes, $\mathcal{W} \subseteq \mathcal{N}$, which we call *worker nodes*. For simplicity, we assume that t is a multiple of the number of worker nodes, and the threads of the canonical application are evenly distributed across the worker nodes. We assume that parameters t and $\mathcal{W}$ have been previously tuned by some existing tool(s) for optimal parallelism tuning (tuning of t) and thread placement (tuning of $\mathcal{W}$), e.g., [9], [38]. In addition, the above parameters do not change while the application runs, and no other processes are co-located in the canonical application's worker nodes.

Further, we consider that the work performed and 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. We assume that the access volume to thread-private pages is negligible when compared to the available memory BW.

In contrast, we assume that the canonical application places an extreme memory demand on the shared pages. Moreover, the workload of the canonical application is BW-intensive, such that the memory access throughput that it attains is the dominant factor to its overall performance, rather than memory access latency (and other overheads unrelated to memory). Hence, hereafter we focus on memory throughput to shared pages, and omit latency from our equations. Furthermore, we assume that the canonical application accesses shared data predominantly in read-only mode; i.e., write accesses to shared pages are so rare that they have no relevant impact on performance. Finally, we consider that the canonical application accesses all shared pages with the same probability.

Shared pages are interleaved across all memory nodes in a weighted fashion, where some nodes may receive more pages than others. No matter which page interleaving is chosen, we assume the shared space fits the available physical memory. Pages are interleaved according to a weight distribution, $\mathcal{D} = \{w_1, w_2, .., w_N\}$, where w_i denotes the fraction of shared pages that node i will hold (such that $\sum w_i = 1$). Therefore, to maximize the performance of the canonical application, we need to find the optimal weight distribution that maximizes the overall throughput to shared data.

2) Finding the optimal weight distribution: Now we discuss how to find the optimal weight distribution for the canonical application. We introduce the function $bw(n_{src} \rightarrow n_{dst})$, which denotes the BW that a given thread located at worker node n_{dst} can use when reading from a node n_{src}. For presentation simplicity, we start by assuming that function $bw(n_{src} \rightarrow n_{dst})$ is well known for all node pairs in the system and does not change for different weight distributions. We discuss how to lift these assumptions in the next section.

Single-worker scenario. The simplest scenario is where all threads run on a single worker node, n_w. BWAP adapts the approach that Yu et al. proposed in the context of data placement in SMP systems with heterogeneous DRAM-NVM memory hierarchies [35]. We start by considering that the threads in n_w need to read a total of S bytes of shared data before completing their work. Since the canonical application's performance is dominated by memory throughput, its execution time is determined by the time that the threads in n_w take to transfer S bytes from memory. Since memory is interleaved among the memory nodes of the system, threads will need to read from pages held at distinct nodes. Furthermore, since the canonical application accesses any page with a uniform probability, the portion that is read from each node i will be proportional to the weight of i; more precisely, $S \times w_i$ bytes.

Moreover, we consider that the data sets that n_w reads from each node in the system are transferred in parallel to n_w. As Yu et al. have shown [35], this approximation is relatively accurate for memory-intensive applications in systems where the number of accessing threads in a node is substantially higher than the number of source memory nodes.

Putting it all together, we can then determine the execution time of the threads in n_w as the time to complete the longest (parallel) transfer from every memory node:

$$T = \max_{i \in \mathcal{N}} \frac{S \times w_i}{bw(n_i \rightarrow n_w)} \tag{1}$$

If pages were uniformly interleaved (i.e., equal weights for all pages), execution time would be determined by the time to transfer from the node with the lowest BW (to n_w). Hence, to minimize execution time, one can reduce the pages placed in that node (by decreasing the corresponding weight) until it no longer incurs the longest transfer time. After that, another node becomes the one that contributes with the longest transfer time and its weight should also be reduced; and so forth. It is easy to show that the resulting optimal solution consists of setting the weight of each node n_i as follows:

$$w_i = \frac{bw(n_i \rightarrow n_w)}{\sum_{i \in \mathcal{N}} bw(n_i \rightarrow n_w)} \tag{2}$$

Multi-worker scenario. We now generalize the previous solution to scenarios where the canonical application has a higher parallelism level, thus spans its threads across two or more worker nodes. In this scenario, the execution time of the application is given by the time that (the threads running at) the slowest worker node takes to complete, as follows:

$$T = \max_{n_w \in \mathcal{W}} (\max_{i \in \mathcal{N}} \frac{S \times w_i}{bw(n_i \rightarrow n_w)}) \quad (3)$$

Like in the single-worker scenario, we can minimize the execution time by adjusting the weight distribution. However, the multi-worker scenario introduces a subtle challenge: the BW that two distinct worker nodes, n_A and n_B, may use when reading from a target node n_i can be different. In fact, changing the weight of n_i to greedily optimize n_A's performance can degrade n_B's performance, and vice-versa. Therefore, the optimization of the weight distribution needs to take all the transfers by all the worker nodes into account.

We achieve this by defining *minimum BW* as the BW of the weakest path among the paths interconnecting a given target node to the worker nodes as $minbw(n) = \min_{n_w \in \mathcal{W}} bw(n_i \rightarrow n_w)$, and by transforming Equation 3 to employ this notion:

$$T = \max_{i \in \mathcal{N}} \frac{S \times w_i}{minbw(n_i)} \quad (4)$$

Finally, we can apply a similar optimization strategy as we do in the single-worker scenario, but this time by considering the minimum BW values. Consequently, the optimal solution in the multi-worker case consists of making each node's weight proportional to its minimum BW:

$$w_i = \frac{minbw(n_i)}{\sum_{i \in \mathcal{N}} minbw(n_i)} \quad (5)$$

3) Estimating BW: The solution built so far assumes that the BW between two nodes ($bw(n_{src} \rightarrow n_{dst})$) is well known and does not change for different weight distributions. We now question these assumptions and propose a practical solution that does not depend on them.

It is important to understand what main factors contribute to the BW that threads in a NUMA machine may effectively use. The value of $bw(n_{src} \rightarrow n_{dst})$ is limited by the nominal BW of the path from n_{src} to n_{dst} (if $n_{src} \neq n_{dst}$). However, the effective BW that threads in n_{dst} get when reading from n_{src} is also determined by the access demand that the application places on the system, in different complex ways.

First, it is known that the actual BW of a single memory controller is influenced non-linearly by the actual access demand to that controller [37]. To complicate matters, the effective BWs, as perceived from worker nodes, are also strongly affected by cross-thread and cross-node interference [27]. The access demand on the memory controller at node n_{src} includes concurrent accesses from multiple threads running in the same worker (either n_{src} or a remote node); further, if the canonical application spans multiple worker nodes, the memory demand on n_{src} combines contending accesses by threads residing on distinct nodes (besides local threads at n_{src}). Finally, BW is also affected by interconnect congestion. This may happen when multiple co-located threads access the same remote memory node (through the same link), or when threads from different worker nodes issue memory requests whose result is delivered through interconnect paths that share one or more links. Therefore, it is clear that our initial assumptions on $bw(n_{src} \rightarrow n_{dst})$ are questionable. First, we relied on an exact knowledge of BW in a NUMA machine. Second, when searching for the optimal weight distribution, we assumed that memory throughput was immutable during that search.

BWAP follows a pragmatic approach to approximate the $bw(n_{src} \rightarrow n_{dst})$ function. More precisely, for a fixed set of worker nodes, we start by deploying a memory-intensive benchmark (representing the canonical application) and uniformly interleaving the benchmark's pages across all nodes in the machine. We use a simple benchmark that spawns as many threads as the available hardware threads on the worker nodes, and each thread performs a random traversal of a shared array. At the same time, we rely on hardware performance counters to monitor per-node memory throughput.

The profiled throughputs between each pair of nodes, n_{src} and n_{dst}, are used as the values of $bw(n_{src} \rightarrow n_{dst})$. This approach neglects the differences in access demand that occur when page placement changes from the profile-time uniform interleaving scenario to the final weighted interleaving scenario. However, our results (in Section IV) confirm that, BWAP is still able to devise efficient weight distributions.

On the practical side, at installation time on a given machine, the canonical tuner needs to run the above profiling procedure for the relevant combinations of worker node sets (with different sizes). The set of explored worker node sets does not need to be exhaustive: i) a large number of worker node sets can be filtered out since they are unlikely to be used by a rational user (e.g., in a dual-socket machine with 2+2 nodes, a 2-worker set comprising nodes at each socket, thus interconnected with a low BW); ii) many worker node sets are symmetrical, hence only one needs to be configured (e.g., in a dual-socket machine with 2+2 nodes with symmetric links between sockets, the optimal weight distribution for the worker set comprising two nodes on one socket is symmetrical to the set comprising the nodes on the other socket).

B. On-line page placement tuning

The DWP tuner takes action when an application is launched. The tuner's API includes a main function `BWAP-init`, which should be called by the target application once it has allocated its initial shared structures. Note that DWP tuner targets applications, which, after an initial stage, enter an execution stage with stable memory access behavior (identically to systems such as Carrefour [24] and Asymsched [9]). The main argument of `BWAP-init` points to the set of worker nodes on which the application is running.

The goal of the DWP tuner is to tune the weight distribution by searching for an appropriate application-specific *DWP*. We recall that *DWP* determines the balance between pages mapped to the set of worker and non-worker nodes, while preserving the relative weight relations within the sets of worker and non worker nodes. The canonical weight distribution corresponds to $DWP = 0$, while $DWP = 1$ corresponds to the extreme where all pages are mapped to the worker node set. This allows

BWAP to be used with both BW-sensitive (low *DWP*) and latency-sensitive workloads (high *DWP*).

1) Tuning data-to-worker proximity: Initially, the DWP tuner obtains a pre-computed canonical weight distribution for the worker node set. The application's shared pages are initially placed based on this weight distribution ($DWP = 0$).

The BWAP library adapts *DWP* on-the-fly during the first seconds after the application calls `BWAP_init`, based on hill climbing. Departing from the $DWP = 0$, we periodically monitor average resource stall rates (stalled cycles per second), by reading hardware performance counters via a portable library [22]. It is well known that stalled cycles are strongly correlated to execution time [17]. At each period, we collect n measurements over an interval of t seconds. We then sort and discard the first and the last c measurements to filter outliers. At each iteration, we compare the current average stall rate with the previous one, and accordingly vary *DWP* by a constant step, x. If the stall rate decreased, it is likely that increasing *DWP* improved the overall performance. Hence, we continue increasing *DWP*. Otherwise, it is likely that we found a (local) optimum and we stop the hill-climbing.

At each iteration where we decide to increase *DWP*, applying the new value is ensured by incrementally migrating pages from non-worker nodes to worker nodes, where the number of pages that is removed from/added to a given node is proportional to that node's canonical weight. This ensures that the relative canonical weights within the worker and non-worker node set are preserved as *DWP* changes, as intended.

2) Initial placement and incremental migration: At each iteration, the DWP tuner needs to place pages according to a weighted interleaving strategy. However, no direct support for *weighted* interleaving exists in mainstream OSs. Hence, we have to build such support for weighted-interleaved page placement. We achieve this by two alternative means: at kernel and user levels. At the kernel level, we implemented a new policy to support weighted interleaved memory allocation, exposed by a new system call. We also added the weighted interleave option to *numactl* tool and *libnuma* library to avoid the burden of application-level changes.

Our user-level alternative has the advantage of portability (avoiding the need for patching the underlying kernel), yet at the cost of a less accurate interleaving. Algorithm 1 summarizes our user-level page placement algorithm. We start by determining the currently allocated page ranges that are likely to hold shared data. This includes the *.data* and BSS segments, as well as dynamic memory mappings. We divide each address range into contiguous sub-ranges and, for each sub-range, we call *mbind* with the uniform interleaving option over a different set of nodes (line 8 in Alg. 1). More precisely, the first sub-range's pages are interleaved over all nodes, the second sub-range's pages are interleaved over all nodes except the one with the lowest weight, and so forth. The key insight is that, by setting the size of each sub-range (line 7 in Alg. 1), we can ensure that the overall per-node page ratios will be proportional to the desired weights.

This solution is not as accurate as the kernel-level alternative, since it does not enforce that all the sub-ranges reflect the weighted interleaving. Still, it ensures a best-effort shuffling of pages, while keeping the number of *mbind* calls low. By using the *MPOL_MF_MOVE* and *MPOL_MF_STRICT* flags of *mbind*, this approach also works correctly (without kernel modifications) when weight distributions are changed dynamically by the DWP tuner. It is easy to show that, as *DWP* increases in each step, Algorithm 1 will call *mbind* on sub-ranges that were previously mapped according to the same or a wider interleaving. In this case, *mbind* seamlessly performs the necessary page migrations. (The reverse migration/operation is currently unsupported by *mbind*.)

Algorithm 1: User-level weighted interleaving approximation used by the DWP tuner

```
   Input: segment // Struct with start address and length
   Input: nodes // Set of NUMA nodes and respective weights
1  begin
2      address ← segment.startAddress();
3      weight_prev ← 0;
4      while nodes ≠ ∅ do
5          node ← getNodeWithMinWeight(nodes);
6          weight ← node.weight − weight_prev;
7          size ← |nodes| ∗ weight ∗ segment.length();
8          interleave_uniform(address, size, nodes);
9          nodes ← nodes − {node};
10         address ← address + size;
11         weight_prev ← node.weight;
12     end
13 end
```

3) Co-scheduled variant: The mechanism described so far assumed a stand-alone application that runs on a subset of nodes and also has the remaining nodes idle to place its pages. However, many NUMA systems will consolidate workloads in a single physical machine in order to minimize idle hardware resources. Workload consolidation is today an active research topic, both in academia and industry (e.g., with Intel's recent introduction of Resource Director Technology [3] into its high-end processors). A common formulation of the problem considers one or more *high-priority* workloads that are supposed to perform as well as if they were accessing isolated resources (e.g. memory); and one or more *best-effort* workloads that benefit if provided with some resources that are originally assigned to the former workloads [13], [31]. Recent proposals to this problem [20], [39] focus only on single-socket scenarios. In BWAP, we improve on this by enabling the allocation of the BW of the full set of NUMA nodes across two or more applications running in disjoint worker nodes.

In the simplest co-scheduling scenario, we can consider one high-priority workload, A, that has a low memory intensity; and a best-effort workload, B, that is memory-intensive. It is desirable that B places some of its pages on the nodes where A runs (i.e., B's non-worker nodes), so B can benefit from the spare BW of those nodes. However, such memory consolidation strategy should not degrade A's memory performance. To support this scenario, the DWP tuner supports a co-scheduling variant, where the iterative search comprises two stages, both coordinated by an external process that monitors the stall rates of both applications. The rationale of this 2-stage approach is that, below a given *DWP* of B, the demand that A will pose on the local nodes of A will start having a noticeable degradation

of A's performance. Hence, the search should only consider *DWP* values that are above that lower bound.

The first stage determines an approximation of such upper bound. To achieve this, the monitoring process measures the stall rate of A, increasing the *DWP* of B as long as the stall rate of A keeps decreasing. As soon as A's stall rate stabilizes, the search has found a local maximum of A's performance, which is probably a good approximation of the lower bound on the *DWP* of B. At that point on, the second stage starts, which proceeds as we described for the stand-alone (i.e., now guided by the stall rate of B).

As a final remark, we note that there are two aspects that the DWP tuner does not currently handle automatically. First, in the co-scheduled variant, we assume that some external tool/hint [24] has classified each workload as memory-intensive or not. Second, the programmer is expected to call BWAP-init when the program is about to enter its stable phase. The first limitation can be addressed by using the number of memory accesses per instruction (MAPI) to classify workloads as either memory-intensive or not (like in Carrefour [24]). As for the second limitation, one may consider looking at the periodic variation of the MAPI metric and only trigger the DWP tuner when such variation is below a given threshold.

IV. Evaluation

Our evaluation answers two key questions: *1. What performance advantage does BWAP bring to memory-intensive applications on NUMA systems compared to state-of-the-art page placement policies like Carrefour [24] or Asymsched [9], which rely on uniform interleave to place shared pages? 2. Considering each main component of BWAP separately, how effective is it and what overheads does it introduce?*

We consider several state-of-the-art page placement policies, i.e., the Linux's default policy (*first-touch*), uniform interleaving across workers (*uniform-workers*), uniform interleaving across all nodes (*uniform-all*), and *autonuma* [6].

Among the different policies evaluated, *uniform-workers* is especially relevant since it is the core strategy that state-of-the-art proposals adopt when placing pages across memory nodes of a NUMA system. Among others, this includes proposals like Carrefour [24], Asymsched [9], and the BW-aware policies proposed by Baek et al. [35]. Since other recent BW-aware page placement proposals do not support NUMA (e.g. [10], [26]) they are not represented here.

Although Carrefour and Asymsched resort to *uniform-workers*, they complement it with two main optimizations: namely the detection and co-location of private pages, and the replication of read-only pages. We do not evaluate these features since they require kernel modifications and no patch was available for our operating system versions. Still, we note that such optimizations are orthogonal to our paper, since they can directly complement the mechanisms proposed in BWAP.

Besides these alternatives, we evaluate complete BWAP and an incomplete variant, denoted *BWAP-uniform*, which disables the canonical tuner. This variant departs from *uniform-all* as its initial weight distribution and only runs the DWP tuner.

Benchmark	BW Requirements		Memory Access Pattern	
	Reads (MB/s)	Writes (MB/s)	Private Accesses (%)	Shared Accesses (%)
Ocean_cp (OC) [30]	17576	6492	79.3%	20.7%
Ocean_ncp (ON) [30]	16053	5578	86.7%	13.3%
SP.B [12]	11962	5352	19.9%	80.1%
Streamcluster (SC)	10055	70	0.2%	99.8%
FT.C [12]	5585	4715	95.0%	5.0%

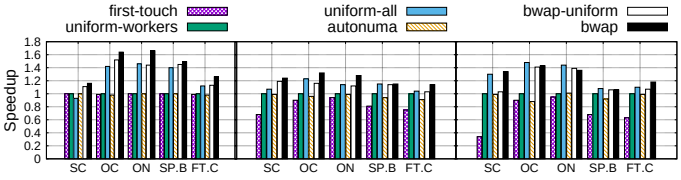
TABLE I: Memory access characterization of the evaluated benchmarks, as obtained by the NumaMMA [15] tool on machine B running each benchmark on a full worker node. The trends of each benchmark do not change significantly for higher number of workers (results omitted for space limitations).

For space limitations, the weighted page interleaving in BWAP is enforced with the portable, user-level option. By enabling the kernel-level variant, we observed only marginal gains (at most 3%) and reach the same main conclusions. The parameters of the DWP tuner of BWAP (Section III-B1) are set as follows: $n = 20$, $c = 5$, $t = 0.2$, and $x = 10\%$. We chose these values by optimizing the parameters for a particular setting (benchmarks Ocean_* on machine A), then used those values for every other benchmark/machine/experiment.

We have evaluated memory-intensive benchmarks from several benchmark suites, namely, NAS [12], PARSEC [7] and SPLASH [30]. We intentionally omit benchmarks with low memory intensity, since page placement has marginal impact on their performance. Although BWAP's design assumes read-only and shared-only pages, we evaluate how it performs as a *best-effort* approach with workloads that do not fit those assumptions. Therefore, our selection of benchmarks promotes diversity with regard to the ratios of read vs. write accesses, and thread-private vs. shared accesses. Table I details the memory access characteristics of each benchmark, which confirms that most benchmarks are far from the assumptions underlying BWAP's design. More specifically, 3 of the benchmarks, namely OC, ON and FT.C have more than 79% of memory accesses that are thread-private. Another aspect where applications differ is their scalability, as Table II summarizes.

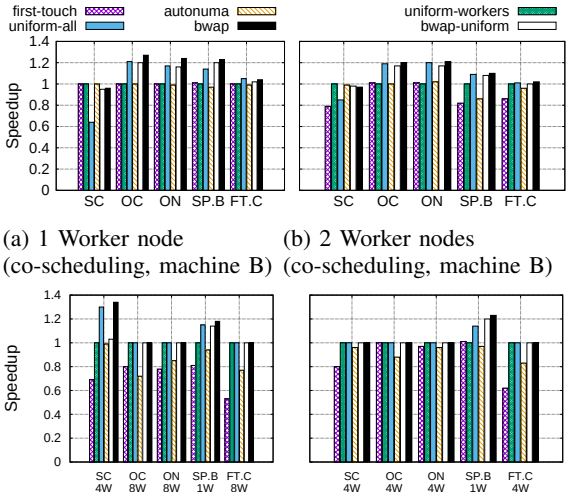
As for the datasets, we used the largest available datasets, i.e., the native inputs for PARSEC and SPLASH and the CLASS B and C datasets for NAS. All these datasets fit in the memory of each node. We use execution time, averaged over 5 runs, as the performance metric.

All the benchmarks are multithreaded applications with a malleable thread pool. For thread placement, we used the usual rule of thumb that is adopted, for instance, by AsymSched [9]: threads are grouped together on the subset of worker nodes with the highest aggregate inter-worker BW. To minimize scheduling-related overheads (e.g., core over-subscription, simultaneous multithreading, thread migration), we pin each thread to a distinct core. We leave the interaction of page placement and OS-guided thread scheduling as future work. The page size used in our experiments and BWAP is the Linux default page size (4KB). Since handling large pages is orthogonal to NUMA handling [14], we disabled large pages. Integrating BWAP with solutions to this problem is left as



(a) 1 Worker node (b) 2 Worker nodes (c) 4 Worker nodes

Fig. 2: Speedup vs *uniform workers* (co-scheduling, mach.A)



(a) 1 Worker node (co-scheduling, machine B) (b) 2 Worker nodes (co-scheduling, machine B)

(c) Optimal num. of workers (stand-alone, machine A) (d) Optimal num. of workers (stand-alone, machine B)

Fig. 3: Speedup vs *uniform workers*

future work.

We ran our experiments on 2 NUMA systems of different scales and asymmetry. Machine A is a 4-socket AMD Opteron Processor 6272, with 8 memory nodes, 8 cores per node, 64GB DRAM, running Linux 4.17. Machine A is representative of a high-end NUMA system with a strongly asymmetric interconnect topology. As it is evident in Figure 1a, interconnect links exhibit ample disparities in link BW, sometimes affecting even different directions of the same link. Machine B is a 2-socket Intel Xeon CPU E5-2660 v4. In contrast, Machine B is a smaller-scale machine with a simpler topology. It has 4 NUMA nodes (using Cluster-on-Die mode), 7 cores per node, 32GB DRAM, running Linux 4.4. In this case, memory BW still exhibits asymmetries, however less pronounced than in machine A. While the lowest BW in machine A was $5.8\times$ lower than the highest BW (i.e., local memory BW), that amplitude drops to $2.3\times$ in machine B.

A. Overall performance evaluation

To quantify the performance of BWAP relative to the page placement alternatives, we measure the performance of the different benchmarks when using the different placement policies. Our initial focus goes to two representative execution scenarios: *co-scheduled* and *stand-alone*.

Co-scheduled scenario. In this scenario, two benchmarks share the same NUMA machine (for instance, hosted by a Cloud provider), as assumed in Section III-B3. We assume that one benchmark, A, is not memory-intensive and the other, B, is. Hence, while A will place pages locally for improved latency, B will resort to BWAP to scatter its pages across all nodes (including the non-worker nodes where A resides) in order to optimize B's throughput, while not degrading A's.

Figures 2, 3b and 3a compare the performance of each application B on the co-scheduled scenario. We consider different allocations of worker nodes to application B: 1, 2, and 4 worker node(s) in machine A; 1, and 2 worker node(s) in machine B. In all cases, A runs on the remaining nodes. As for application A, we run Swaptions [7]. In all our co-scheduled experiments, we did not observe relevant changes to the performance of Swaptions when application B placed some of its pages on the nodes of A, so we omit the performance results of this application in this analysis.

Stand-alone scenario. In the stand-alone scenario, the NUMA machine is entirely available for application A, which can be deployed with any number of threads/workers. Hence, a rational user will run the application with its optimal parallelism level (assumed to be tuned *a priori*). Note that, for each application, the corresponding optimal parallelism level may differ depending on the page placement policy. Figures 3c and 3d show how each application performs when running with the different page placement alternatives in this scenario. For now, let us skip the analysis of the *BWAP-uniform* variant, whose performance is discussed in detail in the next section.

Analysis of the results. The results for both scenarios confirm that, as expected, the best performing solutions are those that fully exploit the available memory BW by placing shared pages across all nodes (*uniform-all* and BWAP), rather than restricting placement within the boundaries of the worker node set (*uniform-workers* and *autonuma*).

Not surprisingly, *first-touch* is usually the worst alternative for multi-worker scenarios by substantial margins, since it tends to centralize many shared pages on a single node (where the initializing threads run) as studied before [19], [24], [33]. Our results show that, even in a single worker scenario, binding the entire application's memory locally to the only worker node is suboptimal for memory-intensive applications. Among the solutions that spread pages across all nodes, BWAP consistently achieves the best performance or, with less favourable applications, performs comparably to the best solution (*uniform-all*). More precisely, BWAP is able to outperform both *uniform-workers* and *autonuma* by up to $1.66\times$, and *uniform-all* by up to $1.50\times$.

As expected, the largest speedups of BWAP are observed on machine A, which has the most asymmetric topology. This confirms our main claim that, for some types of workloads, trivial uniform interleaving policies are not appropriate to exploit the full BW of complex NUMA systems.

One important trend is that the benefits of BWAP over the uniform interleaving alternatives drop when more workers are involved. This is evident in the co-scheduled scenarios (as the number of workers increases). Further, in the stand-alone

scenario, this explains why BWAP only achieves relevant gains for those applications whose optimal parallelism level is lower than the total number of nodes. This trend can be explained by two factors. First, as applications use an increasingly larger worker node set, the worker vs. non-worker dichotomy fades away. Thus, the gains from tuning *DWP* decrease. A second, less intuitive factor, is that, as one enlarges the worker node set, the inter-worker canonical weight distributions (as devised by our canonical tuner) tend to uniformity.

Let us now focus on the subset of benchmarks with non-negligible thread-private memory BW demand, namely OC, ON and FT.C. For such benchmarks, consider the set of thread-private pages that belong to the threads running in a given worker node. In theory, the optimal placement of such pages should consider that worker node only (and regard every other node as non-worker). However, both components of BWAP are, by design, based on the simplifying assumption that every page is accessed from *every* worker node. This approximation allows BWAP to decide the placement of every page similarly, independently of each page's actual worker node set (i.e., the full worker node set in the case of shared pages; a specific worker node in the case of thread-private pages).

Despite this approximation, BWAP is still able to obtain important performance gains with those benchmarks whose ratios of thread-private accesses are at odds with the above assumption (OC, ON and FT.C). The performance benefits obtained with these benchmarks follow the same trend as discussed above: the performance advantage of BWAP over the remaining alternatives is especially evident when the set of worker nodes is smaller; it decays as we enlarge the worker node set, becoming comparable to the best-performing alternative when worker nodes span across every node in the machine.

A more careful analysis allows us to shed some light on these results. We start by observing that the memory demand that these benchmarks place on thread-private pages is sufficiently high to saturate the local memory BW. We support this observation by the fact that, when we place thread-private pages exclusively locally to the thread that accesses each page (with the *first-touch* policy), their performance degrades relatively to the opposite extreme of *uniform-all*. Therefore, the thread-private access demand of these benchmarks calls for page placement strategies that interleave pages across multiple nodes to maximize the available thread-private BW.

On top of this observation, we note that, among the evaluated strategies that resort to page interleaving to meet the previous requirement, BWAP's *best-effort* approach is the most effective one in scenarios where the set of worker nodes is small – namely, either one or two nodes in our experiments. More precisely, when there is only a single worker node, BWAP is trivially accurate for thread-private pages. However, when there are two worker nodes, then BWAP will wrongly decide to place some thread-private pages in the nearest node instead of the local node (when compared to an optimal placement that took thread-private pages into account). Still, our results suggest that the impact of such an inherent inaccuracy is modest. This can be explained by the fact that the BW ratio between the local node and the nearest remote node is much lower ($1.7\times$ in machine A, $1.8\times$ in machine B) than the BW ratio between the local node and the farthest nodes ($5.1\times$ in machine A, $2.3\times$ in machine B). Consequently, despite its inaccurate modelling of thread-private accesses, BWAP still manages to outperform alternative approaches whose placement decisions lead to even larger inaccuracies: those that rely on uniform interleaving do not take BW heterogeneity into account at all (thus it places high numbers of pages in the farthest nodes); while those that simply place thread-private pages locally fail to exploit the additional BW of the non-worker nodes. Overall, these results suggest that BWAP's approximated approach is accurate enough to be advantageous with a wide set of memory-intensive applications.

B. Detailed analysis of BWAP's components

Let us now study the gains and overheads of the various components of BWAP.

Canonical tuner. Looking back at Figures 2 and 3, we can compare the performance of the full BWAP with *BWAP-uniform*. The differences between each alternative denote the effective contribution that the canonical tuner has. Our results show that, for most cases where BWAP is advantageous over *uniform-workers* and *uniform-all*, the larger speedup slice is due to the canonical tuner (which is evident by observing how the results of BWAP-uniform and BWAP differ). Concretely, enabling the canonical tuner attains speedups of up to $1.32\times$ relatively to the uniform variant of BWAP. Further, the highest relative gains happen in machine A, which can be explained by its highest asymmetry. In contrast, the simpler and less asymmetric topology of machine B makes BWAP essentially perform similarly to *BWAP-uniform*. This highlights that, in machines with pronounced BW asymmetries, simple approaches based on uniform interleaving or 2-level weighted interleaving (like BWAP-uniform) are substantially suboptimal; however, in more symmetrical machines, simpler solutions may be an acceptable choice.

DWP tuner. Continuing the previous analysis, we can compare *uniform-all* with *BWAP-uniform* to infer the actual benefits that the DWP tuner of BWAP attains by itself (i.e., departing from *uniform-all* rather than from the canonical distribution). Figures 2 and 3 show that, while BWAP-uniform outperforms *uniform-all* in many scenarios, that is not always the case. This can be explained by taking into account two considerations. First, for some scenarios, the optimal *DWP* is 0. In this case, BWAP-uniform produces the same interleaving as *uniform-all*; however, with the overheads of the online iterative search. Table II details the optimal *DWP* for each application in each co-scheduled scenario. This table yields a high correlation between the null *DWP* cases and the cases where BWAP-uniform does not outperform *uniform-all*. On the positive side, Table II also shows many cases where adapting *DWP* allowed *BWAP-uniform* to choose intermediate values that clearly outperform *uniform-all*. This results in performance gains of up to $1.49\times$ relatively to *uniform-all*

	Machine A			Machine B	
Application	1 Worker	2 Workers	4 Workers	1 Worker	2 Workers
SC	48.00%	0%	23.80%	100%	100%
OC	14.10%	0%	0%	0%	0%
ON	14.10%	16%	0%	0%	0%
SP.B	0%	0%	0%	15.20%	22.20%
FT.C	0%	16.30%	0%	30.30%	0%

TABLE II: Ideal number of worker nodes and DWP values computed via BWAP iterative search (Co-scheduled scenario).

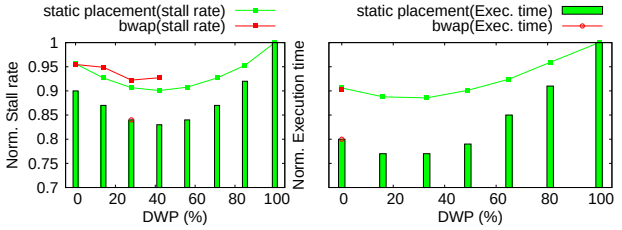


Fig. 4: Evaluating the *DWP* iterative search using Streamcluster on machine A, with 1 (left) and 2 worker nodes (right).

(which are further increased if we consider the combination of both BWAP components).

Overhead and accuracy of DWP tuner. Finally, we study the accuracy and overheads of the DWP tuner. To evaluate both aspects, we have manually deployed each application (at a given machine and scenario) with different *static* values of *DWP* and measure the corresponding execution times and average stall rates. This allows us to understand, what the optimal *DWP* is, what is the corresponding (maximum) performance, and the stall rate curve effectively guides us towards finding the optimal *DWP*. By comparing with the value that our DWP tuner chooses and the resulting execution time, we can assess how close to optimal our search gets and at which overhead.

Our complete experiments with the entire selection of applications confirmed that stall rate is effectively correlated to execution time and its variation with *DWP* is essentially convex. Furthermore, the DWP tuner was able to successfully find the optimal *DWP* by a maximum error margin of 1 iterative step for all cases we evaluated. Figure 4 illustrates this with the Streamcluster application on machine A.

Regarding execution overhead, in the set of experiments discussed in Section IV-A, we measured a maximum overhead of 4% over all the applications. We found that the overhead of DWP tuner can be increased by two main factors: i) higher optimal values of *DWP*, since the search starts at the opposite extreme and each iteration costs time and page migrations; and ii) lower total execution times, which do not allow to amortize the cost of the search performed by the DWP tuner and benefit from resulting optimized page placement.

V. Related Work

When deploying an application on a NUMA system, a number of complex questions arise, from how many threads, to where to place threads and pages. As NUMA systems increase their prominence, these problems receive increasing attention from the research community. Optimizing thread and memory placement on NUMA systems has been extensively studied [8], [9], [11], [18], [21], [23]–[25], [28], [34], [38], [40]–[43].

Linux provides several extensions [1], [6] to improve data access locality in NUMA systems. However, these extensions do not take into account the interconnect asymmetry and do not improve the BW for communicating threads. For instance, autonuma [6] migrates threads closer to the memory they are accessing and/or migrates application data to memory closer to the threads that reference it, thereby implementing locality-driven optimization. Linux also provides an option to uniformly interleave part of address space across memory nodes. BWAP complements these strategies by providing a novel BW-aware alternative to the uniform interleaving.

Among the proposals that address page placement, we can distinguish between proposals that aim at minimizing memory latency (e.g., [6], [25]) and proposals that consider memory BW as the main bottleneck (e.g., [9], [24]). Depending on the application, the main bottleneck can be latency or BW, or somewhere in-between those extremes. In contrast to these proposals that focus on each extreme, BWAP takes this spectrum into account with its DWP tuner.

Apart from page placement, other approaches have been proposed to improve memory performance in NUMA systems. Carrefour [24] replicates read-only pages accessed from multiple nodes. It also detects private pages and places them close to the corresponding thread. Shoal [33] and Smart Arrays [19] introduce programming abstractions that enable a runtime layer to choose the most appropriate NUMA page placement, data layout and/or replication, while taking into account the underlying memory topology and the run-time behaviour of the application. Shoal and Smart Arrays adopt uniform interleaving as one of their NUMA-aware page placements. BWAP is less intrusive, since the application only needs to be linked with and activate DWP tuner. Still, the design principles behind BWAP's placement policy can also be applied to improve these systems.

Recently, some authors have studied the problem of bandwidth-aware page placement for different heterogeneous memory systems [10], [26], [35]. All these works consider a tiered memory architecture consisting of two (or more) types of memory nodes with heterogeneous performance characteristics (BW, latency, etc). Moreover, all works share the fundamental insight that, for BW-intensive applications, their throughput improves if memory accesses are distributed across all memories, proportionally to the BW of each memory. Multi-node NUMA systems, as addressed by BWAP, are also characterized by heterogeneous BWs, even when every NUMA node relies on the same memory technology (e.g., DRAM). Hence, BWAP shares the same BW-aware placement principle as the previous works. However, the problem of BW-aware page placement in NUMA systems has crucial differences (as Section I highlights), with new challenges that either render the above-mentioned proposals inappropriate or strongly suboptimal if applied directly to NUMA systems. Among the previously mentioned proposals for BW-aware page placement in heterogeneous memory systems, they either

do not support multi-node NUMA systems [10] or, in order to distribute pages across (hybrid memory) NUMA nodes, simply use the `uniform-workers` policy. Hence, BWAP can complement or extend these techniques to support NUMA systems whose nodes comprise heterogeneous hybrid memory hierarchies.

Our work is also related to workload consolidation techniques for cloud computing systems and datacenters. Recent works have focused on optimizing memory throughput in these scenarios by employing last-level cache and memory bandwidth partitioning [20], [39] across co-scheduled applications. While these works focus on single-node systems where two or more applications share the same CPU, BWAP targets multi-node NUMA systems where each node is exclusively allocated to one application at most. Works such as [16], [32] propose effective tools to characterize (either through an analytical model or through an empirical procedure) the NUMA topology. These can be integrated into BWAP to allow BWAP to devise a more accurate canonical distribution.

VI. Conclusions

Although new thread placement approaches for asymmetric NUMA systems have recently emerged, today's usual techniques for page placement still rely on the obsolete assumption of a symmetric architecture. This paper proposes BWAP, a novel approach for asymmetric bandwidth-aware placement of shared pages in NUMA systems. Our experimental evaluation shows that BWAP improves the gains of state-of-the-art policies by up to 66%, on commodity NUMA machines. The gains of BWAP are especially evident in co-scheduled scenarios and when the application does not scale up to the available hardware parallelism.

While far from trivial, the design of BWAP is inherently *best-effort*, since it relies on important simplifying assumptions about the workloads and the underlying system. Therefore, our contributions can be seen as first step that opens avenues for follow-up research on BW-aware page placement for NUMA systems. Among future work directions, we plan to more accurately model workloads with relevant write and/or thread-private access volumes, as well as workloads with non-uniform access distributions to the shared address space. Since these scenarios are characterized by inherently asymmetric memory access patterns, the different sets of pages may have distinct optimal placements (e.g., depending on whether a page is thread-private or shared, read or write-dominated, hot or cold). Hence, accurately determining the optimal page placement in such scenarios requires devising different canonical weight distributions and *DWP* values, as well as physically mapping pages according to different placement configurations. The key challenge here lies in achieving these goals while retaining BWAP's key virtues of transparency and portability.

As further future work, we intend to extend BWAP to dynamically adjust its weight distribution throughout the application's execution time, in order to obtain improved performance for applications whose access patterns change over time or for co-scheduling scenarios with dynamic sets of applications. This extension would enable integrating BWAP at the core of dynamic runtime support systems like Callisto [36] or Asymsched [9]. Finally, we plan to extend BWAP to support NUMA systems whose nodes have hybrid memory subsystems (e.g. DRAM and NVRAM) or where the computational power is heterogeneous between different nodes.

VII. Acknowledgements

The research leading to these results has received funding from the EU Horizon2020 programme via the EPEEC project (grant agreement No 801051); from the Fundação para a Ciência e a Tecnologia (FCT) via the projects UIDB/50021/2020 and PTDC/EEISCR/1743/2014; and from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under FPA 2012-0030.

References

[1] Autonuma: the other approach to numa scheduling. https://lwn.net/Articles/488709/. Accessed: 2019-10-14.
[2] Aws cassandra: Cassandra, numa and ec2. http://cloudurable.com/blog/cassadra-numa-ec2-aws/index.html. Accessed: 2019-10-14.
[3] Intel® resource director technology (intel® rdt). https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html. Accessed: 2019-12-20.
[4] The mongodb 4.0 manual. https://docs.mongodb.com/manual/administration/production-notes/. Accessed: 2019-10-14.
[5] Mysql bugs, bug #57241. https://bugs.mysql.com/bug.php?id=57241. Accessed: 2019-10-14.
[6] Toward better numa scheduling, jonathan corbet. https://lwn.net/Articles/486858/. Accessed: 2019-10-14.
[7] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
[8] A. Collins et al. Lira: Adaptive contention-aware thread placement for parallel runtime systems. ROSS '15.
[9] B. Lepers et al. Thread and memory placement on numa systems: Asymmetry matters. USENIX ATC '15.
[10] C. Chou et al. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. MEMSYS '17.
[11] D. Goodman et al. Pandia: Comprehensive contention-sensitive thread placement. EuroSys '17.
[12] D. H. Bailey et al. The nas parallel benchmarks–summary and preliminary results. SC '91.
[13] D. Lo et al. Heracles: Improving resource efficiency at scale. ISCA '15.
[14] F. Gaud et al. Large pages may be harmful on numa systems. USENIX ATC'14.
[15] F. Trahay et al. Numamma: Numa memory analyzer. ICPP 2018.
[16] G. Chatzopoulos et al. Abstracting multi-core topologies with mctop. EuroSys '17.
[17] G. Chatzopoulos et al. Estima: Extrapolating scalability of in-memory applications. *ACM Trans. Parallel Comput.*, 2017.
[18] G. Georgakoudis et al. Scalo: Scalability-aware parallelism orchestration for multi-threaded workloads. *ACM Trans. Archit. Code Optim.*, 2017.
[19] I. Psaroudakis et al. Analytics with smart arrays: Adaptive and efficient language-independent data. EuroSys '18.
[20] J. Park et al. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. EuroSys '19.
[21] J. Rao et al. Optimizing virtual machine scheduling in numa multicore systems. HPCA '13.
[22] J. Treibig et al. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *ICPPW '10*.
[23] K. K. Pusukuri et al. Tumbler: An effective load-balancing technique for multi-cpu multicore systems. *ACM Trans. Archit. Code Optim.*, 2015.
[24] M. Dashti et al. Traffic management: A holistic approach to memory placement on numa systems. ASPLOS '13.
[25] M. Diener et al. Kernel-based thread and data mapping for improved memory affinity. *IEEE Trans. on Parallel and Distributed Systems*, 2016.
[26] N. Agarwal et al. Page placement strategies for gpus within heterogeneous memory systems. *SIGPLAN Not.*, 2015.

[27] N. Denoyelle et al. Modeling Large Compute Nodes with Heterogeneous Memories with Cache-Aware Roofline Model. In *PMBS 2017*.
[28] R. Lachaize et al. Memprof: A memory profiler for numa multicore systems. USENIX ATC'12.
[29] S. Blagodurov et al. A case for numa-aware contention management on multicore systems. USENIX ATC'11.
[30] S. C. Woo et al. The splash-2 programs: Characterization and methodological considerations. ISCA '95.
[31] S. Chen et al. Parties: Qos-aware resource partitioning for multiple interactive services. ASPLOS '19.
[32] S. Kaestle et al. Machine-aware atomic broadcast trees for multicores. OSDI'16.
[33] S. Kaestle et al. Shoal: Smart allocation and replication of memory for parallel programs. USENIX ATC '15.
[34] S. Srikanthan et al. Coherence stalls or latency tolerance: Informed cpu scheduling for socket and core sharing. USENIX ATC '16.
[35] S. Yu et al. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. ICS '17.
[36] T. Harris et al. Callisto: Co-scheduling parallel runtime systems. EuroSys '14.
[37] W. Wang et al. Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *HPCA'14*.
[38] W. Wang et al. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *HPCA'16*.
[39] Y. Xiang et al. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. ICPP 2019.
[40] Z. Majo et al. Matching memory access patterns and data placement for numa systems. CGO '12.
[41] Z. Majo et al. Memory system performance in a numa multicore multiprocessor. SYSTOR '11.
[42] Z. Majo et al. A library for portable and composable data locality optimizations for numa systems. *ACM Trans. Parallel Comput.*, 2017.
[43] T. Harris and S. Kaestle. Callisto-RTS: Fine-grain parallel loops. ATC'15.