Contention-aware Application Performance Prediction for Disaggregated Memory Systems

Felippe Vieira Zacarias
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
fvieira@bsc.es

Rajiv Nishtala
Norwegian University of Science and Technology
rajiv.nishtala@ntnu.no

Paul Carpenter
Barcelona Supercomputing Center
paul.carpenter@bsc.es

ABSTRACT

Disaggregated memory has recently been proposed as a way to allow flexible and fine-grained allocation of memory capacity to compute jobs. This paper makes an important step towards effective resource allocation on disaggregated memory systems. Specifically, we propose a generic approach to predict the performance degradation due to sharing of disaggregated memory. In contrast to prior work, cache capacity is not shared among multiple applications, which removes a major contributor to application performance. For this reason, our analysis is driven by the demand for memory bandwidth, which has been shown to have an important effect on application performance. We show that profiling the application slowdown often involves significant experimental error and noise, and to this end, we improve the accuracy by linear smoothing of the sensitivity curves. We also show that contention is sensitive to the ratio between read and write memory accesses, and we address this sensitivity by building a family of sensitivity curves according to the read/write ratios.

Our results show that the methodology predicts the slowdown in application performance subject to memory contention with an average error of 1.19% and max error of 14.6%. Compared with state-of-the-art, the relative improvements are almost 24% on average and 33% for the worst case.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**.

KEYWORDS

Performance degradation, Performance counters, Memory subsystem, Memory bandwidth, Performance prediction

ACM Reference Format:
Felippe Vieira Zacarias, Rajiv Nishtala, and Paul Carpenter. 2020. Contention-aware Application Performance Prediction for Disaggregated Memory Systems. In *17th ACM International Conference on Computing Frontiers (CF '20), May 11–13, 2020, Catania, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387902.3392625

1 INTRODUCTION

High performance computing infrastructures are built from a large number of servers, each of which has a fixed set of computing resources such as processor, memory and storage. Important ratios including memory capacity per core are fixed at design time [19, 42]. Large HPC systems typically follow a rule of thumb that couples 2–3 GB of main memory per x86 core or 1 GB per Blue Gene PowerPC core. Nevertheless, HPC application memory requirements vary dramatically due to intrinsic differences in application memory requirements and the effects of strong scaling [29, 48]. Since applications must run on a number of self-contained servers, there is often a mismatch between fixed resource proportionalities and the needs of the workload [30], leading to underutilized resources [45].

More recently, disaggregated memory has been proposed in order to allow a flexible and finer-grained allocation of memory capacity to compute jobs [18, 19, 41]. In the disaggregated design, individual components such as processor, memory and storage are interconnected over a network [14, 21, 28, 36, 40] rather than being restricted to a bus on a single board [33]. The EuroEXA family of projects [14, 34], for instance, provides a global physical address space in which cores can access remote memory via RDMA and direct load–store instructions. In comparison with traditional shared memory processors, the focus is on sharing of memory capacity, rather than coherent sharing of data.

Each application requests a number of CPU cores and a given memory capacity per core, and memory capacity is allocated as a common resource [20]. Since applications running on different nodes can share memory devices and interfaces, performance may be affected by contention [37]. For this reason, coscheduling and resource allocation decisions for disaggregated memory require contention awareness in order to optimize application performance and overall system throughput. Slowdown based methods have been successful for single node coscheduling [12, 16, 23], but they have not been applied to disaggregated memory.

This paper presents a slowdown based method for disaggregated memory. It creates a family of sensitivity curves to relate computing demands for memory bandwidth to application degradation. These sensitivity curves are built using a carefully curated set of performance counters that correlate well with the memory bandwidth of the application. Since HPC applications are generally batch applications (rather than interactive services), performance is inversely proportional to runtime, which for a given application is itself proportional to memory bandwidth. Our results show that our slowdown methodology is a good approximation for predicting the performance of applications under remote memory access interference with an average error of 1.19% and max error of 14.6%.

In summary, we make the following contributions:

(1) A general approach to predict performance degradation due to sharing of disaggregated memory.
(2) We propose a slowdown based method involving smoothed sensitivity curves in order to increase prediction accuracy.

CF '20, May 11–13, 2020, Catania, Italy

(3) We propose the use of a family of sensitivity curves to account for varying ratios between read and write memory accesses.
(4) We demonstrate through experimental analysis that the proposed approach delivers higher prediction accuracy than the state-of-the-art, with relative improvements of almost 24% on average and 33% for the worst case.

The rest of the paper is organized as follows. Section 2 provides a brief background and distinguishes our approach from the large body of related work. Next, Section 3 presents our developed approach in detail. Section 4 provides the experimental validation on a specific platform, and finally Section 5 concludes the paper.

2 BACKGROUND AND RELATED WORK

This section briefly describes the background on shared memory architectures, and it distinguishes our work from the large body of related work.

2.1 Background

Enabled by the advances in network technologies, several approaches have been proposed towards a general-purpose architecture to disaggregate resources, as an alternative to the monolithic server-centric approach. They aim to solve the problem of imbalance in memory usage and expand memory capacity by exposing the global memory to all machines.

Gu *et al.* [18] implement a scalable and decentralized remote memory paging solution to enable memory disaggregation. It divides the swap space of each machine and distributes the pages across many remote machines using RDMA operations for all remote I/O operations.

Lim *et al.* [20, 21] propose a remote memory blade that can be used for memory capacity expansion to improve performance and for sharing memory across servers. They developed a software-based prototype by extending the Xen hypervisor to emulate a disaggregated memory design wherein remote pages are swapped into local memory through on-demand explicit direct memory access transfers.

Shan *et al.* [37] propose a split kernel OS architecture to manage disaggregated systems. It breaks the OS into pieces with different functionalities, each running on and managing a hardware component. All components communicate using a customized RDMA-based network stack.

The EUROSERVER [14] project built a disaggregated system architecture, which provides a global physical address space and the ability for cores to access remote memory via RDMA or direct load–store instructions. By appropriately configuring the cache policy each remote memory access can be cached locally. This work was continued in the ExaNoDe [34] and EuroEXA [31] projects. The dReDBox project [4] also proposes a customizable low-power architecture comprised of hot-pluggable modules that provide compute, memory and accelerators resources. Its software-defined global memory and peripheral resource management offers fine-grained resource allocation on-demand to improve utilization.

2.2 Related work

The most straightforward way to approach the problem of characterizing performance degradation of an application in contention is to perform a brute-force sweep. However, due to the $O(N^2)$ cost it is not possible to be employed in a production environment. A suitable alternative is to apply a method that analyzes each program once and scales linearly with the number of applications.

Prior work predicted performance degradation when two applications run together using the concepts of *sensitivity* and *contentiousness*. These methods are decoupled into two steps: (1) to measure the sensitivity curve, which quantifies the impact of different levels of shared resource contention on the application's performance; (2) to measure the contentiousness value, which quantifies how much shared resource contention is generated by the application. Then, to predict application's performance the contention is applied to the sensitivity curve. In both cases, pressure may be quantified in various ways, for example, in terms of cache capacity and/or memory bandwidth. The rest of this section presents prior efforts which identify, quantify or model the applications' performance due to shared resource contention.

Slowdown based methods — De Blanche *et al.* [12, 13] propose a slowdown based characterization method to estimate the applications' slowdown when sharing the memory bus. Their sensitivity curve is obtained using synthesized memory traffic and the contentiousness is found using performance counters. De Blanche *et al.* use a fixed read or write ratio to create the sensitivity curve, while, for better accuracy, we calculate the performance degradation using the sensitivity curve that best represents the interfering application.

Bubble-Up proposes a generalisable methodology for predicting performance degradation from contention for shared resources in the memory subsystem [23]. For Bubble-Up, sensitivity and contentiousness quantify occupancy of the last-level cache (LLC), but it is pointed out that the methodology can be applied to other metrics. Since Bubble-Up only considers cache occupancy, it cannot be applied in its original form to our problem, which has separate caches.

Bandwidth Bandit [16] proposes a quantitative profiling method for analyzing the performance impact of contention for shared memory resources to determine the application's sensitivity to latency and bandwidth. For performance prediction it uses a bandwidth graph, which is a quantitative description of the application's sensitivity to bandwidth contention, then it finds the throughput of a given number of co-running instances. Rather than stealing cache capacity as it is done in the author's previous work [15] (see below), Bandwidth Bandit executes the applications in the same socket using n instances of a single-threaded application.

Cache contention — Several works have been proposed to model how applications' performance is affected by changes in the amount of available shared resources, especially cache capacity. Our work does not consider using cache interference, as it is misleading when coscheduling applications using separate cache hierarchies [12].

Eklov *et al.* [15] propose a methodology to measure application performance and bandwidth as a function of the cache capacity occupied by an interfering application. Zhao *et al.* [46, 47] capture the aggregate cache and bandwidth utilizations of all cores and characterize the performance degradation using a regression approach, which admits different sub-functions depending on which contention factors are dominant. Casas *et al.* [8] present a methodology which quantifies an application's utilization of the memory

hierarchy, specifically capacity and bandwidth of shared caches to predict the application's performance when the required memory resources are not available. Casas *et al.* qualitatively demonstrate that their validation methodology has higher accuracy than prior work [15, 16].

Online mechanisms — The following works provide online methods to estimate performance of applications already running in contention and using simulation. Subramanian *et al.* [38, 39] and Xiong *et al.* [43] use, respectively, the memory request rate, cache access rate and IPC to estimate the slowdown for individual applications. Their method involves giving high priority to the target application for a short time to estimate the value of its respective metric if the application were running alone, and then using this value to calculate the experienced interference.

3 PREDICTING APPLICATION PERFORMANCE

This section describes our solution to predict the relative slowdown of the application due to remote memory access interference.

3.1 Problem definition

The model disaggregated architecture is shown in Figure 1, which is inspired by the UNIMEM approach [14]. In this design, independent computing units execute their own Operating System image, and can access memory attached to it (local memory access) as well as memory attached to another computing unit through a global interconnect (remote memory access). The remote memory access is performed through a common Global Address Space, either using ordinary load–store instructions or using Remote Direct Memory Access (RDMA) operations. The design supports two cache policy configurations: caching the memory (locally) at the unit that requested the access or (remotely) at the unit attached to the memory.

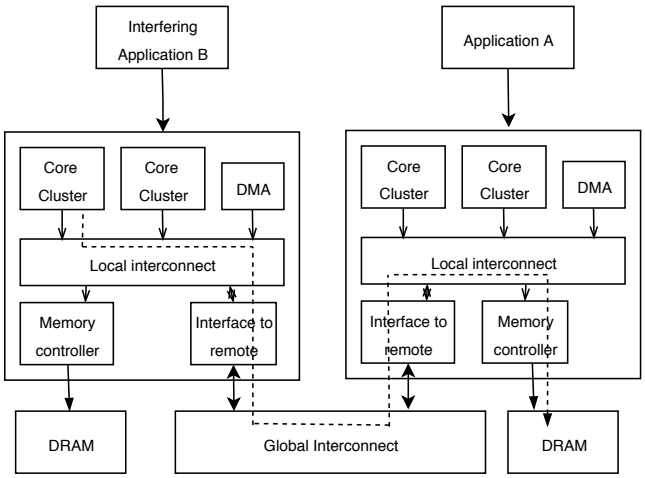

Figure 1: Model architecture for memory disaggregation based on UNIMEM [14]. Remote memory accesses are routed to the appropriate node through a global interconnect.

Since the concept of remote memory decoupled from processor is not easy to implement in real prototypes and due to the absence of an available prototype [7], we emulate a disaggregated shared memory architecture, without the need for real hardware, using a conventional multi-socket server, as shown in Figure 2. This approach takes advantage of a two-socket server and its separated LLC to create pressure only in the desired shared resource. According to Molka *et al.* [26], cache coherence traffic is not a significant bottleneck in a two-socket system. Thus, in our approach the processor and cache resources are isolated from interference while the effects of memory bandwidth contention can be observed on the shared memory resource. In addition, the latency of the cross-socket memory access for our experimental platform detailed in Section 4.1, is similar to those presented in disaggregated works such as [1, 45].

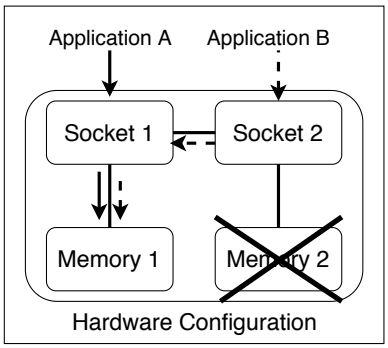

Figure 2: Emulated disaggregated scenario applied in this work.

As shown in Figure 2, all threads of the interfering application (B) execute on the socket 2 while issuing memory requests exclusively to the memory bank attached to socket 1 (remote access). On the other hand, all threads of the target application (A) use only its memory bank (local access), thus contending for memory controller and memory bandwidth. Applying this scheme, the impact of application B on application A can be modeled based in B's bandwidth interference. This model predicts the slowdown experienced by application A in the face of diverse remote bandwidth requirements.

3.2 Overview

Figure 3 shows a high-level view of our slowdown methodology, which is comprised of three components:

(1) **Interference phase** — In the interference phase, a set of interfering applications execute concurrently with the target Application A, using only the remote bandwidth as the measure of pressure. The interfering applications differ as they account for different read/write ratios to create a family of distinct sensitivity curves (see Section 3.3). In the model disaggregated memory architecture, remote memory accesses do not create cache contention in the local node as their cache hierarchies are separate (see Figure 1). Contention induced by cache capacity can misrepresent the degradation experienced by the applications in such scenario, so the metric of interest is bandwidth usage and contention on memory controller.

(2) **Bandwidth calculator** — The bandwidth calculator is responsible for collecting hardware counters to calculate Application B's

remote bandwidth (bw_b) and its read/write ratio (R/W Ratio). The calculated values will be used to select the appropriate target's sensitivity curve to predict the performance degradation (see Section 3.4).

(3) **Prediction methodology** — In the prediction methodology, we start by selecting application A's sensitivity curve taking into account Application's B read/write ratio. Then, to mitigate the measurement noise from the benchmark process and improve the prediction accuracy, a smoothing function is applied to the sensitivity curve. Linear and polynomial smoothing functions are considered, as they are straightforward and commonly employed. This gives the function (f_a) that relates Application A's slowdown to the total interfering remote bandwidth. Finally, the Application's B bandwidth (bw_b) is applied to the function to predict Application A's performance (y) when Application B contends for remote memory bandwidth (see Section 3.5). The prediction methodology has $O(cN)$ complexity (N as the number of applications and c the number of distinct read/write ratios), as it requires the application to be characterized only once instead of every pairwise execution that would be required in a brute force characterization scenario.

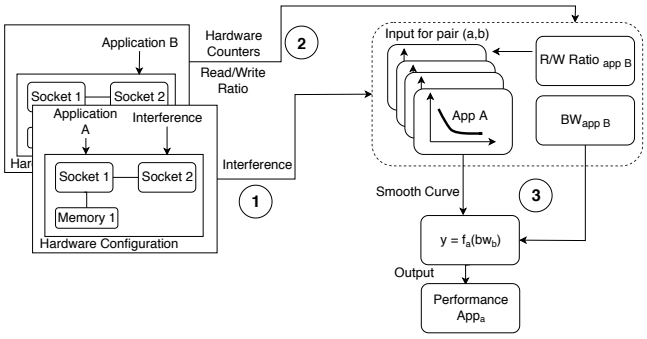

Figure 3: High-level view of the slowdown methodology.

3.3 Creating sensitivity curves

To create the sensitivity curves, we quantify the memory pressure using the total remote bandwidth rate, and calculate the performance degradation the application suffers when executing with an interfering application. For this purpose, we use a synthetic benchmark, which is an adaptation of the STREAM benchmark [24], modified to generate a variable requested bandwidth for the remote memory. Then, we create a curve of the target application's performance, normalized to its performance running alone, on the y-axis, versus the remote bandwidth generated by the interfering application, on the x-axis. An example of the sensitivity curve for the Triad workload from the STREAM benchmark is presented in Figure 4.

Radulovic *et al.* [32] argue that the total bandwidth is misleading because it combines into a single metric the aggregate bandwidth of reads and writes, even though they are fundamentally different operations. Distinguishing read and write bandwidths when creating the sensitivity curve not only accounts for the particularities of

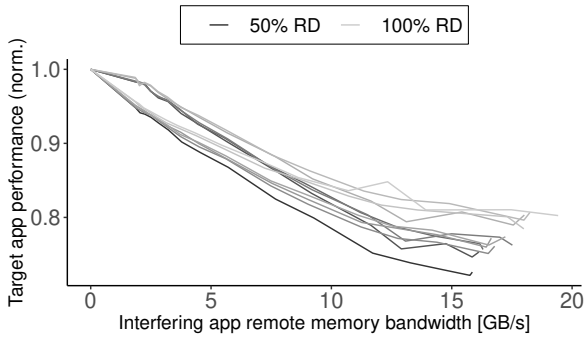

Figure 4: Family of sensitivity curves for Triad workload.

the memory subsystem, but also more accurately represents the behavior of real applications. Variable read/write ratios are supported by the synthetic interfering benchmark [6].

We applied the slowdown based methodology by creating a family of sensitivity curves that depend on the interfering application's remote memory bandwidth and read/write ratio. Previous works rely on one single sensitivity curve, however in Figure 4 we show the measured sensitivity curves for the Triad workload on our experimental platform (more detail in Section 4) while the proportion of reads varies between 50% and 100%. In all our experiments, curves created using a higher percentage of reads generally achieved a higher sustainable bandwidth. As pointed out in [32], writes have additional delays caused by the write recovery time, which is the delay between a write and the next precharge command, and the write-to-read delay time, which is the interval between a memory write and the consecutive read.

An important aspect when creating a sensitivity curve is the representation of pressure in its x-axis. When executing the synthetic interfering benchmark and the target application concurrently, both applications will be interfered. Intuitively, the reported interfering bandwidth will be lower than the expected bandwidth of its execution alone.

In our methodology, we use all curves for Section 4 results with the x-axis representing the bandwidth that the interfering application would have had if it were executing alone instead of under contention. This is similar to the Figure 4. We chose this methodology because it is generic to any interference study, and it is appropriate for the scenario where the known value is the actual bandwidth usage that the interference application applies.

3.4 Measuring Contentiousness

In the second part, we characterize the application in terms of how much remote bandwidth it requires throughout its execution by running it solo. The value of remote memory bandwidth will be applied as the contention pressure. For this step, we use performance counters to measure the read and write bandwidths.

Memory bandwidth is typically measured using one of the three approaches: (a) Last Level Cache (LLC) miss [8, 44], (b) Off-core response [17] and (c) uncore Integrated Memory Controller (IMC) counters [3]. While the LLC miss counter accounts for neither the prefetcher read memory traffic nor write memory traffic, off-core response counters cannot detect write memory traffic due cache line evictions, which is the major portion of the overall write traffic.

The uncore IMC counters, on the other hand, measure events in the memory controller, including read and write Column Access Strobe (CAS) commands [10]. These commands are sent from the memory controller to the DRAM at every memory column access and they include prefetching and eviction events. Each memory access consists of a cache line of 64 bytes. As the counter measures memory traffic at the memory channel, only the total read/write traffic per channel is measured, which prevents further segmentation of requests. Its broad range makes it suitable to compute application bandwidth with higher accuracy than LLC misses, providing a complete bandwidth profile. Following the measurements, the per-channel read and write application bandwidths are:

$$\begin{aligned} BW_{\text{read}} &= CAS\_COUNT_{\text{read}} \times 64\,\text{B}/elapsed\_time \\ BW_{\text{write}} &= CAS\_COUNT_{\text{write}} \times 64\,\text{B}/elapsed\_time \end{aligned} \quad (1)$$

and the total bandwidth is therefore:

$$BW_{\text{tot}} = BW_{\text{read}} + BW_{\text{write}} \quad (2)$$

3.5 Prediction Methodology

The penultimate step in predicting the application's performance is to calculate the read/write ratio of the interfering application. This is done using Equation 1. As the last step, we select the target application's sensitivity curve corresponding to the read/write ratio of the interfering application. The selected curve is smoothed using a linear function so as to predict the degradation in the case of missing points with higher accuracy and also to decrease the influence of outliers in the collected data. Finally, we apply the interfering remote memory bandwidth to the target interference curve to obtain the expected target performance when running under interference.

3.6 Key differences compared with state-of-the-art

Our work is differentiated from prior work [12, 16, 23], and provides improvements for a singular reason: Our approach targets performance prediction due to sharing of disaggregated memory, while the prior works use application working set size or local bandwidth as their measure of pressure to create the sensitivity curve. As noted in Section 3.4, cache contention characterization method is misleading for predicting the performance of applications using separated cache hierarchies. For this reason, we proposed using a family of smoothed sensitivity curves to account for varying ratios between read and write memory accesses to increase accuracy and decrease the effect of outliers.

4 EXPERIMENTAL EVALUATION

4.1 Experimentation methodology

This section introduces the applications and hardware resources used to evaluate the accuracy of the proposed approach. We simulated the global shared memory architecture using a conventional node (see Section 3.1). We normalize our results against a brute-force sweep performed against the pair of applications.

Benchmarks — We used applications from several known benchmarks suites, including a total of 44 applications from Parsec (8 applications) [5], Rodinia (5) [9], NAS Parallel Benchmarks (NPB) (8) [2], Splash (5) [35] and another 15 diverse publically available applications. We selected applications to cover a variety of computational patterns found in multithreaded and high performance codes. All applications were compiled with GNU/Linux GCC 7.2 and multithreading enabled. We also used the Linux utility *numactl* to apply affinity settings for both threads and memory placement, and the *Perf* tool to collect the performance counters.

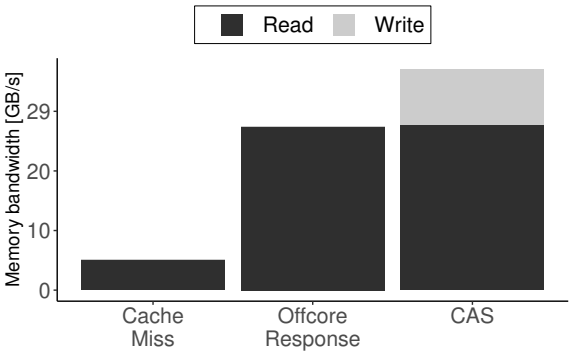

Figure 5: Calculated memory bandwidth for Triad workload using three different hardware performance counter.

Hardware resource — We carried out the experiments on a server equipped with two Intel Xeon SandyBridge-EP E5-2670 that together comprise 16 cores operating at 2.6 GHz. Each socket has 20 MB L3 cache (LLC) shared among all cores, single memory controller, and two Quick Path Interconnect (QPI) links version 1.1 operating at 8.0 GT/s. It implements the home snoop cache coherence with MESIF protocol [10]. The node has 64 GB of DDR3-1600 DIMMs main memory, theoretical bandwidth of 51 GB/s (37 GB/s sustained) for local access and 38 GB/s (20 GB/s sustained) for remote memory access. Its latency is 81 ns and 133 ns for local and remote accesses respectively [26].

Performance Counters — To compare the accuracy for representing the bandwidth of an application, we collected the performance counters listed in Section 3.4 during the execution of the Triad workload. Performance counters can track the occurrence of events with negligible overhead, and there are commonly employed in many related works [8, 22, 27] to measure resource utilization and demonstrate effectiveness of any proposed method. Figure 5 summarizes the calculated bandwidth derived from the collected performance counters. As can be seen, LLC miss counter represents only a fraction of the sustainable memory bandwidth (5 GB/s), confirming that not only write traffic but part of the read traffic is neglected by this performance counter. The off-core response counter displays a bandwidth of about 27 GB/s which is roughly equal the read traffic calculated using CAS counter. However, the difference between both counters emerges in the write traffic captured by the later, which differs by 30% of the overall bandwidth.

The bandwidth calculated by Triad during the tests was 30 GB/s on average. According to McCalpin [25], the bandwidth values assumed by Triad is based on the minimum data traffic that each iteration will perform. For Triad workload this number is 24 B (two floats read and one float write), however it does not account for bytes transferred during write-allocate operations. As Triad kernel requires 4/3 as much bandwidth as the benchmark generates when

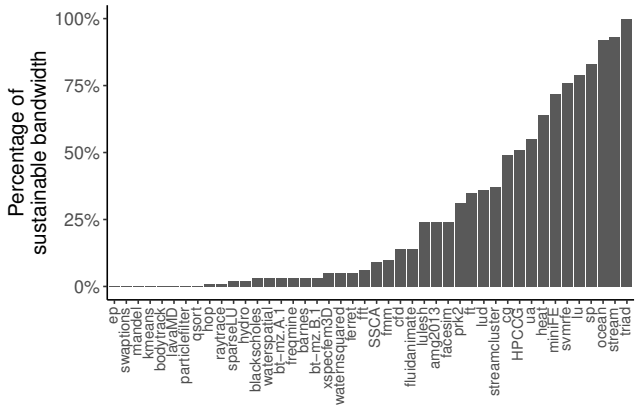

Figure 6: Percentage of sustained memory bandwidth utilization for each application in our study. Memory bandwidth usage calculated using CAS performance counter.

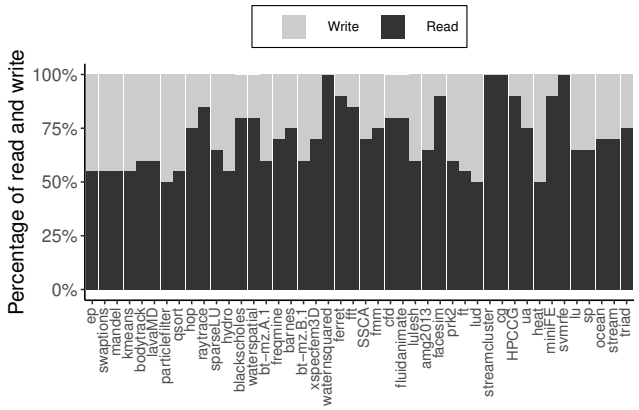

Figure 7: Percentage of read/write ratio for each application considering their sustained bandwidth utilization.

write-allocate is included, the result must be multiplied by a factor of 1.33. After applying the correction factor, the sustainable bandwidth is approximately 39 GB/s. The total bandwidth calculated for Triad workload using CAS counter deviate by only about 5% of the scaled bandwidth, which confirms its accuracy to be sufficient for the proposed methodology.

Application characteristics — Figures 6 and 7 present the characteristics of bandwidth usage for each application considered in our experiments. The applications used in this study show a wide range of memory bandwidth utilization. At least 40% of the applications use 20% or more of the sustainable bandwidth. The sustainable bandwidth for this work is calculated using the highest value achieved among 5 executions. Figure 7 shows that the benchmark suite covers a wide range of read/write memory traffic, with reads accounting for between 50% and 100%. The figure also shows that the family of curves (see Figure 4) correspond to the range that arises in practice.

Sensitivity curves — For our tests, the interfering application generates the sensitivity curve for remote memory bandwidth traffic between 10% and the maximum sustainable bandwidth achieved.

Table 1: Prediction error for linear and polynomial smoothing.

Method	% of Reads	Mean (%)	SD	Max (%)	Cost (×)
Polynomial	**right curve**	**1.15**	**1.49**	**14.5**	**44**
	50	1.29	1.86	20.6	4
	75	1.34	1.86	17.7	4
	100	1.39	2.20	21.0	4
Linear	**right curve**	**1.19**	**1.59**	**14.0**	**44**
	50	1.27	1.86	22.1	4
	75	1.38	1.94	18.9	4
	100	1.49	2.32	19.5	4

We then apply a linear function to smooth the curve before predicting the application's performance. We tried different smoothing functions, and the linear function attained satisfactory results.

When applying pressure equal to the maximum sustainable remote bandwidth, we noticed that the performance of the applications suffer considerable levels of degradation in face of an interfering application issuing memory requests from a remote node. This indicates that the resource allocation for new architectures using global memory address must account for the degradation of applications executing concurrently and sharing the memory subsystem. Also as presented in the figure 4, we can notice the difference in the performance when read/write ratio of the interference application varies. Increasing read ratio, less interference the benchmark causes to the target application, then more sustainable remote bandwidth is used by the remote application. This pattern is emphasized with high remote memory access when we can clearly see the separation between curves.

4.2 Prediction Error

We evaluate the effectiveness of our methodology by predicting the expected performance degradation of the applications in a pairwise fashion. We consider the prediction error to be the absolute value of the difference between the predicted performance and the real normalized application performance under contention. In our tests, we start both target and interfering application at the same time on different sockets. When the interfering application finishes, we restart it to keep the target application in contention during its entire execution. All pairwise combinations were executed in such way that the target application completed at least 10 times. We recorded the actual performance degradation (increasing in runtime) when the applications executed together, in order to compare with the predicted performance. Degradation for the target application is calculated using its normalized performance alone in the system without interference and its performance under contention.

Table 1 summarizes the predicted error for all applications using two different functions to smooth the interference curve. The smoothing functions applied to create the curves are: linear and polynomial function with degree two. It also shows the prediction error using the smoothed specific curve for the interfering application read/write ratio (called right curve) and the smoothed sensitivity curves produced with the static values of 50%, 75% and 100% read/write ratio. The column *Cost* indicates the profiling cost

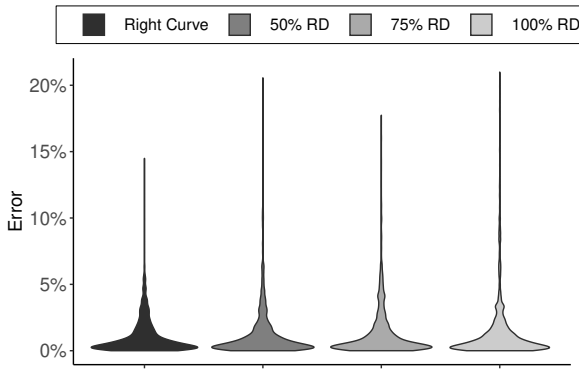

Figure 8: Distribution of error using polynomial smoothing function.

needed by the method compared to previous works that rely on 4 different levels of pressure and a static read/write ratio to create their sensitivity curve.

We achieve the lowest mean and variance prediction errors using the curve corresponding to the actual read/write ratio of the interfering application. In the best case, the right curve using polynomial smoothing attains a mean error of 1.15%, and a relative improvement of 18% compared with predictions using static read/write ratio. In addition, it also has the lowest worst case with around 14% max error and relative improvement between 19% and 31%. The linear smoothing method achieves mean error of 1.19%, which is similar to the polynomial smoothing results, and relative improvements between 26% and 37% for the worst case.

Figure 8 presents the distribution of prediction error using the polynomial smooth function (best result so far). In the Figure we can note that the density of the prediction error using the correct read/write ratio is more concentrated in the base of our chart and the majority of the values is below 5%. Using static curves we note predictions with high variance and the presence of a longer tail, denoting the occurrence of higher values for errors. Using fixed read/write ratio curves the max error is higher than 15%, however the max error using the right curve is lower. The results confirm that the read/write ratio play an important role to predict the performance degradation.

To further evaluate the influence of noise and outliers on our predictions, we compared the smoothing results against the non-smoothed interpolation method used in previous works. We assessed the method using all data points (16 distinct values) collected during the interference benchmark and a sampled version using four distinct interference values (25%, 50%, 75% and 100%) to create the interference curve. Table 2 summarizes the overall result. We do not see an overall improvement compared to the smoothing values when using all data points and right curves. Even though it has mean error similar to linear smoothing and better results than the static curves, its worst case prediction increases. This can be viewed as a side effect of the noise intrinsic to the collected data for all sensitivity curves calculated.

Sampling the data points and using the right curve, the interpolation maintains the lowest mean error compared to its static counterparts. However, it increases the worst case prediction error compared to the smoothing methods and decreases compared to

Table 2: Prediction error for smoothing functions and sampled data.

Method	# of Points	% of Reads	Mean (%)	SD	Max (%)	Cost (×)
Polynomial	All points	**right curve**	**1.15**	**1.49**	**14.5**	**44**
Linear	All points	**right curve**	**1.19**	**1.59**	**14.0**	**44**
Interpolation	All points	**right curve**	**1.19**	**1.63**	**24.2**	**44**
		50	1.34	1.89	18.3	4
		75	1.34	1.82	19.4	4
		100	1.43	2.19	21.0	4
Interpolation (**Memgen**)	Sampled	**right curve**	**1.35**	**1.62**	**18.1**	**11**
		50	1.47	1.91	18.1	1
		75	**1.56**	**1.94**	**21.6**	**1**
		100	1.58	2.18	24.7	1

using all data points. A positive side effect of sampling the data for the right curve results is that part of the noise is removed and the interpolated values provide a more stable result, which explain its results compared with the other methods. This aspect illustrates that the overall accuracy is affected by the data points and smoothing function, which decreases the effects of outliers and improves the worst case predictions.

Another important point to be considered is the effective cost to achieve the results. In Table 2 for our initial approach the high cost of using the right curve can be broken down into two components: the number of read/write curves and the number of data points collected. In our tests we analyzed 11 different read/write ratio curves and we also sampled 12 supplementary points in addition to the default 4 points used in previous works. Even though the polynomial smoothing has the best results so far, it also has the highest cost to compute. The cost is 44 times higher than using a single sampled static curve due to the additional curves collected and the number of points.

To investigate the effects of the trade-off between accuracy and cost, we reduced the cost of our methodology by estimating the values for the right curve based in two sampled curves. For this test we used the values of 50% and 100% sampled sensitivity curves collected for the previous test, and we estimated the performance of the target application based on the read/write ratio proximity of the interfering application to both curves. The process is shown in equation 3. First we calculate the proximity (*scale*) of the interfering application read/write ratio to 100% ratio. Then, we predict the performance of the target application for 50% (*P 50%*) and 100% (*P 100%*). In the end, we apply a weighted mean estimation to determine the estimated performance (*P_est*).

$$\begin{aligned} scale &= (interf\_ratio - 50)/(100 - 50) \\ P\_est &= ((P50\%) \times (1 - scale) + (P100\%) \times (scale)) \end{aligned} \tag{3}$$

Table 3 summarizes the overall result reducing the cost of the methodology. Sampling the data points decreases to 11 times the cost of using the right curve for the smoothing methods and also keeps the mean and maximum prediction errors lower than the sampled static curve. However, we achieve the best trade-off between cost and error estimating the values for the right curve. We reduce the cost to only 2 times and the results are similar to those using a higher number of curves and data points. The linear method

is better than the polynomial one with 1% of difference for max error, thus we chose it to compare with the state of the art.

Table 3: Prediction error for estimated curves.

Method	# of Points	Mean (%)	SD	Max (%)	Cost (×)
Polynomial	All data	1.15	1.49	14.5	44
	Sampled	1.21	1.43	15.4	11
	Estimated	**1.19**	**1.48**	**15.6**	**2**
Linear	All data	1.19	1.59	14.0	44
	Sampled	1.21	1.48	13.7	11
	Estimated	**1.19**	**1.50**	**14.6**	**2**
Interpolation	(**Memgen**) 75	**1.56**	**1.94**	**21.6**	**1**

4.3 Comparison with Memgen

In this section, we compare our methodology to the state-of-the-art slowdown based methodology, Memgen [12]. Memgen uses a modified version of the Triad workload to generate a specific amount of traffic, then creating contention for the target application. Memgen creates four reference points which are 25, 50, 75 and 100% of the sustainable bandwidth as its interference pressure. To create the sensitivity curve, Memgen relies on linear interpolation between the reference points in order to estimate the performance, and defines the contentious pressure as the memory bandwidth usage of an application. To evaluate the applications' memory bandwidth usage, Memgen uses the bus_trans_mem.self performance counter.

To apply the Memgen methodology to our environment, we adapted some unavailable features that will be discussed hereafter. Their triad version code is not available on-line, so we could not use the same interference application to create the sensitivity curve. However, as their version is based on the triad algorithm from STREAM, we applied our interfering application that generates 75% read/write ratio. Using this specific configuration to create the sensitivity curve, we maintain the same static read/write ratio and computing characteristics used by the authors in their work. Another point regarding the sensitivity curve is that [12] and [13] do not specify whether the values of the x-axis are the maximum bandwidth or the interfering bandwidth in contention. We assumed the same approach applied in our work, using the maximum bandwidth achieved during the interfering test.

Another adapted feature concerns using the bus_trans_mem.self performance counter [11] to compute the application's memory bandwidth. This counter is unavailable in the Sandy Bridge processor architecture [10] which is used in our experiments. As consequence, to calculate the contention pressure we applied the *CAS* performance counter to measure the application's memory bandwidth. This performance counters calculate with high precision the actual bandwidth for applications keeping the comparison fair. The result of the comparison with Memgen is shown in Figure 9.

In the Figure we can see that the error for the Memgen methodology is higher than using our methodology. For our approach the distribution of errors are more concentrated in the base, which means that it has the majority of its predictions close to the true value. The Memgen methodology has a relative mean error increase

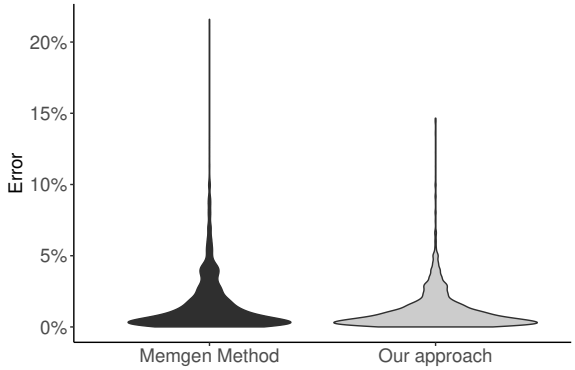

Figure 9: Distribution of error for Memgen methodology and our approach.

of almost 24% (see Table 3 Interpolation method). This is also true for the worst case, where the max error is 21.6% using the Memgen approach and 14.6% using our approach. These results highlights that distinguishing the interference caused by different ratios of read/write and increasing the number of points collected, the slowdown method can be improved.

5 CONCLUSION

In this work, we have proposed a slowdown based methodology to predict the performance degradation that results from contention in remote memory access. We also added to the methodology the concepts of smoothing and read/write memory access ratio to create the correct sensitivity curve, in order to increase the accuracy and similarity with real executions. Using the characterization of an application's sensitivity to contentious pressure from remote access to the memory subsystem, we were able to predict an application's performance in a pairwise execution with 1.19% prediction error on average and 14.6% in the worst case. Compared with the state of the art, the relative improvements are almost 24% on average and 33% for the worst case. Interesting avenues to build on this work in future including taking account of multiple execution phases in a single run and reducing the need to profile the applications in advance. We believe that the approach in this paper is of particular importance for novel and upcoming global shared memory architectures and future resource allocation decisions for such platforms.

ACKNOWLEDGMENTS

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 754337 (EuroEXA); it has been supported by the Spanish Ministry of Science and Innovation (project TIN2015-65316-P and Ramon y Cajal fellowship RYC2018-025628-I), Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), and the Severo Ochoa Programme (SEV-2015-0493).

REFERENCES

[1] Bulent Abali, Richard J Eickemeyer, Hubertus Franke, Chung-Sheng Li, and Marc A Taubenblatt. 2015. Disaggregated and optically interconnected memory: when will it be cost effective? *arXiv preprint arXiv:1503.01416* (2015).

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V.

Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks: Summary and Preliminary Results. In *SC*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[3] Tirtha Pratim Bhattacharjee. 2013. *Data Movement and Workload characterization: Intel Sandy Bridge Core and Uncore PMU features.*

[4] Maciej Bielski, Ilias Syrigos, Kostas Katrinis, Dimitris Syrivelis, Andrea Reale, Dimitris Theodoropoulos, Nikolaos Alachiotis, D Pnevmatikatos, EH Pap, George Zervas, et al. 2018. dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1093–1098.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*. ACM, 72–81. https://doi.org/10.1145/1454115.1454128

[6] BSC. 2009. PROFET: Analytical model that quantifies the impact of the main memory on application performance and system power and energy consumption. https://github.com/bsc-mem/PROFET Accessed: 2019-10-16.

[7] Dhantu Buragohain, Abhishek Ghogare, Trishal Patel, Mythili Vutukuru, and Purushottam Kulkarni. 2017. DiME: A performance emulator for disaggregated memory architectures. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. 1–8.

[8] Marc Casas and Greg Bronevetsky. 2015. Evaluation of HPC applications' memory resource consumption via active measurement. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (2015), 2560–2573.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*. IEEE, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[10] Intel Corporation. 2012. Intel® Xeon® Processor E5-2600 Product Family Uncore Performance Monitoring Guide. *tech. rep.* (March 2012).

[11] Intel Corporation. 2018. *Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.*

[12] Andreas De Blanche and Thomas Lundqvist. 2014. A methodology for estimating co-scheduling slowdowns due to memory bus contention on multicore nodes. In *International conference on parallel and distributed computing and networks*.

[13] Andreas De Blanche and Thomas Lundqvist. 2015. Addressing characterization methods for memory contention aware co-scheduling. *The Journal of Supercomputing* 71, 4 (2015), 1451–1483.

[14] Yves Durand, Paul M Carpenter, Stefano Adami, Angelos Bilas, Denis Dutoit, Alexis Farcy, Georgi Gaydadjiev, John Goodacre, Manolis Katevenis, Manolis Marazakis, et al. 2014. Euroserver: Energy efficient node for european micro-servers. In *2014 17th Euromicro Conference on Digital System Design*. IEEE, 206–213.

[15] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. 2011. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*. IEEE, 165–175.

[16] David Eklöv, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. 2013. Bandwidth Bandit: Quantitative Characterization of Memory Contention. In *CGO 2013, 23-27 February, Shenzhen, China*. IEEE Computer Society, 99–108.

[17] Josué Feliu, Julio Sahuquillo, Salvador Petit, and Jose Duato. 2016. Perf&Fair: A Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Trans. Comput.* 66, 5 (2016), 905–911.

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[19] Vamsee Reddy Kommareddy, Amro Awad, Clayton Hughes, and Simon David Hammond. 2018. Exploring Allocation Policies in Disaggregated Non-Volatile Memories. In *Proceedings of the Workshop on Memory Centric High Performance Computing*. ACM, 58–66.

[20] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 267–278.

[21] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.

[22] 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*. ACM, 12.

[23] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*. ACM, 248–259.

[24] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[25] John D. McCalpin. 2019. SC16 Invited Talk: Memory Bandwidth and System Balance in HPC Systems. https://sites.utexas.edu/jdm4372/tag/stream-benchmark/. Accessed: 2019-09-18.

[26] Daniel Molka, Daniel Hackenberg, and Robert Schöne. 2014. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the workshop on Memory Systems Performance and Correctness*. 1–10.

[27] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E Nagel. 2017. Detecting memory-boundedness with hardware performance counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 27–38.

[28] Héctor Montaner, Federico Silla, Holger Froning, and José Duato. 2011. Memscale™: A scalable environment for databases. In *2011 IEEE International Conference on High Performance Computing and Communications*. IEEE, 339–346.

[29] Rajiv Nishtala, Paul Carpenter, and Xavier Martorell. 2019. Performance effects on HPC workloads of global memory capacity sharing. In *MULTIPROG*.

[30] Antonios D Papaioannou, Reza Nejabati, and Dimitra Simeonidou. 2016. The benefits of a disaggregated data centre: A resource allocation approach. In *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–7.

[31] EuroEXA project. 2009. H2020 project number 754337. https://euroexa.eu/ Accessed: 2019-10-16.

[32] Milan Radulovic, Rommel Sánchez Verdejo, Paul Carpenter, Petar Radojkovic, Bruce Jacob, and Eduard Ayguadé. 2019. PROFET: Modeling System Performance and Energy Without Simulating the CPU. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '19)*. ACM, New York, NY, USA, 71–72. https://doi.org/10.1145/3309697.3331502

[33] Pramod Subba Rao and George Porter. 2016. Is memory disaggregation feasible?: A case study with Spark SQL. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*. ACM, 75–80.

[34] Alvise Rigo, Christian Pinto, Kevin Pouget, Daniel Raho, Denis Dutoit, Pierre-Yves Martinez, Chris Doran, Luca Benini, Iakovos Mavroidis, Manolis Marazakis, et al. 2017. Paving the way towards a highly energy-efficient and highly integrated compute node for the Exascale revolution: the ExaNoDe approach. In *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE, 486–493.

[35] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *ISPASS*. IEEE, 101–111.

[36] A Saljoghei, V Mishra, M Bielski, I Syrigos, K Katrinis, D Syrivelis, A Reale, DN Pnevmatikatos, D Theodoropoulos, M Enrico, et al. 2018. dReDbox: Demonstrating Disaggregated Memory in an Optical Data Centre. In *2018 Optical Fiber Communications Conference and Exposition (OFC)*. IEEE, 1–3.

[37] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[38] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 62–75.

[39] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. 2013. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 639–650.

[40] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, and Christian Pinto. 2018. A Software-defined SoC Memory Bus Bridge Architecture for Disaggregated Computing. In *Proceedings of the 3rd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*. ACM, 3.

[41] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, Ilias Syrigos, Maciej Bielski, Dimitris Theodoropoulos, Dionisios N Pnevmatikatos, and Georgios Zervas. 2017. A software-defined architecture and prototype for disaggregated memory rack scale systems. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 300–307.

[42] Carlos Vega, Jose Fernando Zazo, Hugo Meyer, Ferad Zyulkyarov, Sergio López-Buedo, and Javier Aracil. 2017. Diluting the Scalability Boundaries: Exploring the Use of Disaggregated Architectures for High-Level Network Data Analysis. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 340–347.

[43] Dongliang Xiong, Kai Huang, Xiaowen Jiang, and Xiaolang Yan. 2017. Providing Predictable Performance via a Slowdown Estimation Model. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 25.

[44] F.V. Zacarias, V. Petrucci, R. Nishtala, P. Carpenter, and D. Mossé. 2019. Intelligent Colocation of Workloads for Enhanced Server Efficiency. In *2019 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.

[45] Georgios Zervas, Hui Yuan, Arsalan Saljoghei, Qianqiao Chen, and Vaibhawa Mishra. 2018. Optically disaggregated data centers with minimal remote memory latency: technologies, architectures, and resource allocation. *Journal of Optical Communications and Networking* 10, 2 (2018), A270–A285.

[46] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. 2015. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2015), 1443–1456.

[47] Jiacheng Zhao, Huimin Cui, Jingling Xue, Xiaobing Feng, Youliang Yan, and Wensen Yang. 2013. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22nd international*

conference on Parallel architectures and compilation techniques. IEEE Press, 201–212.

[48] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Trans. Archit. Code Optim.* 14, 1, Article 3 (March 2017), 26 pages. https://doi.org/10.1145/3023362

A ARTIFACT APPENDIX

A.1 Abstract

This artifact is intended to demonstrate the process of using the Slowdown based method proposed in the paper.

A.2 Artifact check-list (meta-information)

- Algorithm: Slowdown Based Method to predict performance degradation from contention in remote memory access
- Program: Stream Benchmark
- Compilation: GCC compiler and Intel MPI
- Binary: Synthetic stream benchmark with different read and write ratio for the interference test. To run it is necessary Intel MPI version 2016.3.067
- Data set: Set at compiling time
- Run-time environment: Any Linux distribution with Linux performance monitoring tool Perf and Numactl installed. The Intel MPI version must have the libmpigf.so and libmpi.so libraries. To collect certain counters either root access is required or set perf_event_paranoid to 0. Additionally, all processing scripts require the R environment
- Hardware: Dual socket Intel Sandy Bridge architecture 16 cores and 32 GB memory RAM with access to CAS and OFFCORE response performance counters for benchmarking
- Metrics: Execution time in s, Bandwidth GB/s
- Output: Results in CSV files and graphs
- Experiments: The scripts provide an example of workflow to collect execution time and performance counters using native and/or interfered modes (real or synthetic). R scripts process the output data and create the final results
- How much disk space required?: ~1 GB
- How much time is needed to prepare workflow?: ~1 h
- How much time is needed to complete experiments?: The benchmark can finish in ~10 h
- Publicly available?: Yes
- DOI: 10.5281/zenodo.3749249

A.3 Description

A.3.1 How delivered — The artifact package can be download through this hyperlink. It includes all scripts to set up a small example to exemplify the methodology applied in the paper.

A.3.2 Hardware dependencies — It is necessary to use a dual socket machine with Intel Sandy Bridge processors with at least 32 GB of memory to emulate the interference in disaggregated remote memory access. It requires the presence of CAS and OFFCORE response performance counters to be used as the applications' contentiousness metric.

A.3.3 Software dependencies — The target application is compiled using GCC compiler. For the interfering application that issues remote bandwidth pressure at the sensitivity curve step, the process uses a synthetic benchmark which is an adaptation of the Stream benchmark. It is executed using the Intel MPI. After executing the benchmark, the final outputs are processed using R scripts. The environment to run this final step can be different from the one used during the benchmarking because it does not require too much processing power. In order to generate graphs and tables, it requires R software environment with the following packages: readr, plyr, dplyr, tidyr, ggplot2, zoo.

A.3.4 Data sets — Defined at compile time.

A.4 Installation

Extract the artifact package file and enter in the created artifact/ folder. The folder sources/ contains the binaries and source code of the application that will be used in the following workflow. The script start_benchmark.sh executes the workflow and compiles the application will be used.

A.5 Experiment workflow

By executing the start_benchmark.sh it will run the following steps: 1) Compile the target application, 2) Collect the application's execution time without interference using the script native.sh, 3) Collect the application's performance counter to define its contentiousness using the script perf_script.bash, 4) Execute the application in pairs to collect the application execution time in contention in order to compute its real degradation due remote memory access interference using the script degradation_pair.sh, 5) Collect data to build the application's sensitivity curve. It is carried out executing sensitivity_benchmark.sh script. The application is profiled using the interfering stream – sent within the artifact – by varying the read/write ratio and the inference intensity

After every step, a complementary script is called to format the intermediate files and generate a final csv file. If there are enough resources, all steps can be executed separately. Check inconsistencies on the output files with the verify_files.sh script. For the last step, execute the R scripts through (run_r_scripts.sh) which will use the generated files from previous steps to generate the final output. To reproduce the outputs from the paper, execute run_r_scripts.sh, located in paper_files folder. The process can take less than 30 min.

A.6 Evaluation and expected result

The scripts will create sub-folders to store intermediate files in results/ folder. The final formatted files are native_time.csv, result_pairs.csv, counter_complete_miss_off_uncore.csv and curve_final.csv. They will be stored in the artifact home folder after the execution of each step.

The R scripts will generate graphs similar to the ones presented in the paper, and tables using the previous generated csv files for the prediction step. Tables will be exported to csv files named prediction_diff.csv and estimated_curve_prediciton_diff.csv. Note that this artifact is intended to show the process of predicting degradation using the concepts of sensitiveness and contentiousness

A.7 Experiment customization

All scripts can be customized to increase the number of iterations, as well as adding new benchmarks. The places are signalized in the scripts.