Congestion-Aware Memory Management on NUMA Platforms: A VMware ESXi case study

Jagadish B. Kotra, *Seongbeom Kim, Kamesh Madduri, Mahmut T. Kandemir
*The Pennsylvania State University, *Google Inc.*
{jbk5155, kamesh, kandemir}@cse.psu.edu, *kimsbeom@gmail.com

***Abstract*—he VMware ESXi hypervisor attracts a wide range of customers and is deployed in domains ranging from desktop computing to server computing. While the software systems are increasingly moving towards consolidation, hardware has already transitioned into multi-socket Non-Uniform Memory Access (NUMA)-based systems. The marriage of increasing consolidation and the multi-socket based systems warrants low-overhead, simple and practical mechanisms to detect and address performance bottlenecks, without causing additional contention for shared resources such as performance counters. In this paper, we propose a simple, practical and highly accurate, dynamic memory latency probing mechanism to detect memory congestion in a NUMA system. Using these dynamic probed latencies, we propose congestion-aware memory allocation, congestion-aware memory migration, and a combination of these two techniques. These proposals, evaluated on Intel Westmere (8 nodes) and Intel Haswell (2 nodes) using various workloads, improve the overall performance on an average by 7.2% and 9.5% respectively.**

I. Introduction

Multi-socket architectures have become quite common over the past decade in the server domain, and many existing systems today employ them. One of the fundamental characteristics of these architectures is the "non-uniformity" they impose on data access latencies. More specifically, data accesses made from a core to the memory of the same socket is much faster compared to accesses made to remote sockets' memory. This local/remote dichotomy introduced by such Non-Uniform Memory Access (NUMA) architectures have important consequences spanning thread and data placement.

The conventional wisdom in dealing with NUMA systems has always been maximizing local accesses through techniques such as careful partitioning of data by OS or compiler across the sockets [1]–[4], co-locating data and computations, and data migration to adapt to dynamic changes in working sets and data access patterns. Increasing number of sockets (from 2 to 16 [5]) in recent architectures further emphasizes this local memory-centric view of NUMA optimization since data access latencies are now exhibiting even larger variances compared to the past – as in some cases a data access may now require several hops to reach the target memory. However, there are two complementary trends against this local memory-centric view. First, with transistor scaling, each socket itself can now host numerous hardware threads which contend for shared resources like last-level caches and memory resources. Second, application dataset sizes keep increasing [6]–[18]. Although the capacities of the last-level caches (LLC) also increase correspondingly, the dominant increase in the applications' footprint saturate memory bandwidth, thereby causing the memory wall to grow higher and higher with each generation of processors. Consequently, the performance of a node can degrade with increasing number of hardware threads and increasing working set sizes. This performance degradation is more pronounced in the virtualized environments where a large number of VMs can be consolidated on a single socket. Recent work [19] showed that the performance degrades by 22.1% on an average when 2 VMs are consolidated on the same socket, and further degrades by an additional 18.4% when 8 VMs are consolidated due to memory controller contention for cloud applications. Motivated by this, researchers [1, 2] proposed congestion-aware task/memory management techniques to mitigate contention. In this work, we propose and evaluate a strategy, which, instead of evading the remote accesses, tries to use remote memory bandwidth dynamically.

Our specific contributions include:

- We present detailed experimental data showing that the performance degradation due to increasing consolidation. For example, going from 18 VMs to 36 VMs can degrade the performance by up to 92%. Observing that the first step in any data allocation strategy that intends to use remote memory is to gauge the dynamic congestion on different nodes, we propose a dynamic latency probing mechanism.
- Taking the end-to-end congestion information as a feedback and observing that in many applications data allocations spread over the entire execution, we propose congestion-aware memory allocation (CAMA). CAMA evaluated in VMware ESXi hypervisor improved the overall system performance on an average, by 8.6% and 5.1% in Intel Haswell and Intel Westmere, respectively.
- To optimize for other types of applications where the memory allocation is done initially and accessed later, we propose congestion-aware page migration (CAPM) optimization. The results collected on our Intel Haswell and Intel Westmere based systems indicate average performance improvements of 8.2% and 6.2% respectively. Furthermore, combining CAMA and CAPM, we propose CAMM which further take these savings to 9.5% and 7.2%.

II. Background and Experimental Setup

A. NUMA architecture

Figure 1 shows the basic block diagram for Intel Haswell system. The nodes[1] in Haswell (similar to Westmere) processor are connected to one another through an Intel Quick Path Interconnect (QPI) [20]–[22]. A socket consists of a local memory which is managed by a local memory controller (MC)

[1]We use *socket* and *node* interchangeably in the rest of this paper.

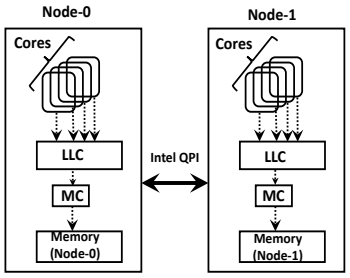


Fig. 1: Haswell block diagram.

TABLE I: Haswell config.

Nodes	2
Cores/socket	18
Core Freq.	2.3GHz
LLC Size	45MB
QPI Speed	9.6GT/sec
DRAM (Capacity)	DDR4-2133 (512GB)

Nodes	8
Cores/socket	10
Core Freq.	2.27GHz
LLC Size	24MB
QPI Speed	6.4GT/sec
DRAM (Capacity)	DDR3-1333 (1TB)

TABLE II: Westmere config.

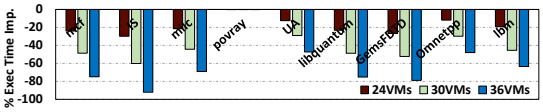


Fig. 2: Performance degradation with varying consolidation scenarios on Intel Haswell.

From/To Cycles	N-0	N-1	N-2	N-3	N-4	N-5	N-6	N-7
N-0	**290**	454	735	736	840	835	839	843
N-1	454	**290**	734	748	865	860	864	868
N-2	735	734	**290**	452	839	863	888	863
N-3	736	748	452	**290**	840	861	862	864
N-4	839	862	839	840	**290**	451	734	750
N-5	835	860	863	861	451	**290**	741	739
N-6	839	863	885	863	734	729	**290**	454
N-7	843	868	863	864	748	739	454	**290**

TABLE III: Westmere access latencies (in CPU cycles).

From/To Cycles	N-0	N-1
N-0	**229**	319
N-1	319	**229**

TABLE IV: Haswell access latencies (in CPU cycles).

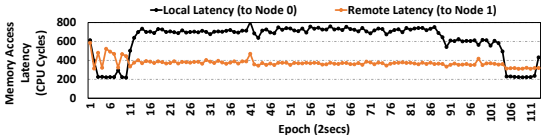


Fig. 3: Memory access latencies noted from Node-0 in a congested environment on Haswell for npb_is workload.

as shown in the Figure 1, and is a Chip Multi-Processor (CMP) containing cores; all cores share a last-level cache, represented by LLC in the figure. A local memory access incurs a DRAM access delay and, if the DRAM banks are busy, an additional MC queuing delay as well. However, since the remote memory access involves moving data from the remote socket over the QPI, an additional interconnect latency is incurred. Tables III and IV show the latencies in CPU cycles after a system bootup without any guest VMs running inside the ESXi for Westmere and Haswell systems, respectively. In these tables, the value in row-x and column-y represent memory access latency in CPU cycles observed from Node-x when accessing the data allocated in the local memory of Node-y. Hence, values in the diagonal (bolded) represent the local memory access latencies. From these tables, it can be observed that the local latency incurred is always lower than the remote latency.

B. Experimental setup

We used two Intel NUMA based systems, Haswell and Westmere to conduct our experiments. Tables I and II summarize the configurations of these machines. The Intel Haswell system shown in Table I contains 2 NUMA sockets. In each socket, there are total 36 hardware (2-way hyper-threaded) threads sharing an LLC of size 45MB. Intel Westmere system[2] shown in Table II contains 8 NUMA sockets and each socket contains 20 hardware threads (2-way hyper-threaded) per socket. The applications presented in Table V are run inside the guest VMs running on ESXi hypervisor. Each guest VM runs a RHEL 6.0 Operating System. For simplicity, we assume each guest VM to be running a single application, though our analysis and evaluations hold equally well for multiple applications running inside a guest VM. For our evaluations, we used multi-threaded and single-threaded applications from various benchmarks suites from HPC domain viz., NAS [24], SPEC CPU2006 suite [25], SPECOMP [26], MANTEVO [27], also the SPECJBB [28] suite and finally the graph parallel-processing multistep [29] suite. The specific applications from these suites are further categorized in to high (represented by H), medium (M) and low (L) based on their memory-intensities, measured by LLC Misses Per Kilo Instruction (LLC-MPKI) as depicted in Table V. Applications with LLC-MPKI greater than 15 are categorized as H, while those with LLC-MPKIs between 5-15 (inclusive) are categorized as M and the ones with LLC-MPKIs below 5 are categorized as L.

We further used homogeneous and heterogeneous workloads formed using the applications in Table V to quantify the benefits of our optimizations. The homogeneous workloads execute same applications while the heterogeneous workloads execute different applications on a given socket. To simulate real-world scenarios, where different nodes on the same machine exhibit varying degrees of congestion, in some workloads, we map high/medium/low-memory intensive applications on different nodes. However, to replicate scenarios where all the nodes experience the same degree of memory congestion, we map different instances of the same workload on all the nodes in a machine, there by covering the entire spectrum of the real-world scenarios.

In the rest of the paper, the % Execution Time Improvement (or degradation) results reported in various figures for each workload is the geometric mean of individual applications' execution time improvement (or degradation) over the corresponding applications' baseline execution time.

III. Motivation

As can be observed from Tables III and IV, the access latency to local node is around 40% lower in both the systems compared to access to the neighboring remote nodes. ESXi is NUMA-aware [30] and hence tries to allocate memory from the local node, whenever possible, for better performance. However, when a high number of memory-intensive VMs are consolidated on a single node, the MC queuing and DRAM access latencies can dominate the additional interconnect latency incurred by the remote access.

A. Effect of increasing consolidation

Figure 2 shows how performance changes with increasing number of VMs consolidated on a socket for our homogeneous workloads on Intel Haswell system. In this experiment, each VM is pinned to a hardware thread on a processor socket. The performance degradation reported is normalized to the basecase which executes 18 VMs. As can be observed from

[2]In the interest of space, we could not present the Westmere block diagram. Please find it in slide-12 of [23].

Benchmark	Suite	Multi-threaded ?	LLC-MPKI	Category	Benchmark	Suite	Multi-threaded ?	LLC-MPKI	Category
mcf	SPEC2K6	No	66.9	H	GemsFDTD	SPEC2K6	No	13.33	M
npb_is	NPB	No	43.51	H	HPCCG	mantevo	No	11.81	M
npb_ua	NPB	No	42.72	H	equake	SPECOMP	Yes	7.24	M
Omnetpp	SPEC2K6	No	23.01	H	specjbb	SPECJBB	Yes	6.48	M
swim	SPECOMP	Yes	22.162	H	CC	MULTISTEP	Yes	2.45	L
lbm	SPEC2K6	No	21.62	H	mgrid	SPECOMP	Yes	1.95	L
libquantum	SPEC2K6	No	19.52	H	SCC	MULTISTEP	Yes	2.45	L
milc	SPEC2K6	No	18.28	H	povray	SPEC2K6	No	0.07	L

TABLE V: Benchmarks used in our evaluation.

Figure 2, performance degrades as the number of VMs consolidated increases and is as high as 92% for npb_is [24] which is highly memory-intensive, while it is almost 0% for low memory-intensive applications like povray as such applications rarely access memory.

To further explain the degradation in Figure 2, we present the access latencies for npb_is homogeneous workload. Figure 3 shows the memory access latencies incurred from Node-0 to both Node-0 and Node-1 on our Intel Haswell system running homogeneous npb_is workload. As can be observed, initially when there is no congestion in the system the local node (Node-0) access latency is lower than the remote node (Node-1) access latency. However, around the 11th epoch (23 secs), the local latency to Node-0 starts increasing and becomes greater than the remote nodes' access latency indicating that the Node-0's memory bandwidth is saturated resulting in congestion. Note that, after the 11th epoch, the Node-0 latency continues to dominate the latency to Node-1 till the end of the workload execution in the 105th epoch. In such scenarios, the overall system performance would improve if the neighboring remote nodes' bandwidth could be utilized. However, to use the remote memory bandwidth effectively, one needs to answer the following questions:

1) When to allocate data in the remote node ?
2) What percentage of data needs to be allocated in the remote node for a specific topology NUMA system?
3) In which remote node (number of hops from the source node) should the data be allocated ?

These questions are important, because if data is allocated on the remote node when there is no congestion, system performance can degrade significantly as extra cycles are spent on the interconnect traversal while accessing data. Consequently, it is important to detect the congestion in the system *dynamically* so that the data can be allocated in the most appropriate node. Once the congestion is detected and the decision to allocate data in the remote node is taken, it is important to determine how much data needs to be allocated in the remote node. This decision is also important because if too less data is allocated in the remote node, system performance further degrades as the local node will still be congested. However, if a high percentage of data is allocated on the remote node, local node memory bandwidth will be underutilized, thereby resulting in performance degradation.

B. Static allocation results

Figure 4 plots the percentage execution time improvement when a fixed fraction of data is allocated in the remote node on our Intel Haswell 2 NUMA node system. All the results presented are *normalized* to the basecase where all the data are allocated on the local NUMA node, Node-0. We present results for different allocation ratios, for example, the bars

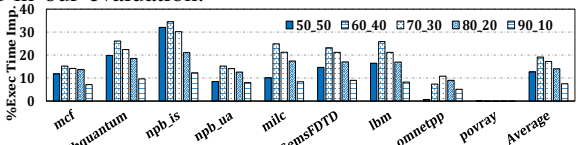


Fig. 4: Execution time improvements with different static allocation ratios on Intel Haswell.

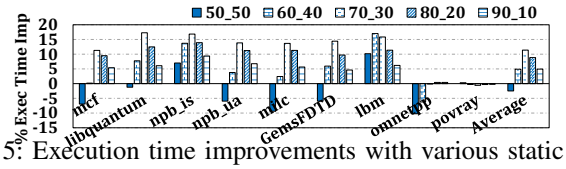


Fig. 5: Execution time improvements with various static allocation ratios on Westmere only using 2 of 8 NUMA nodes.

marked using 60_40 in Figure 4 represents the execution time improvement when 60% of memory is allocated on the local node, and the remaining 40% is allocated on the neighboring remote node. As can be observed, in this system, allocating 10% of data in the remote node (90_10 ratio) does not alleviate congestion and incurs considerably lower performance improvement of around 7.5% on an average. On the other hand, allocating 50% of data on the remote node alleviates congestion; however, it results in under-utilizing the local node memory bandwidth and yields a performance improvement of 12.6% on an average, which is still not very high. Allocating 60% on the local node and 40% on the remote node is optimal in this case, giving a performance improvement of 19.1% on an average. Figure 5 shows the performance improvement when only 2 NUMA nodes out of 8 are used in our Intel Westmere system. In these experiments, memory is allocated only on Node-0 and Node-1. Node-0 is the source node, and Node-1 is the neighboring remote destination node for VMs running on Node-0. As can be observed from Figure 5, 70% local node allocation and 30% remote node allocation result in the maximum performance improvement of 11.4% on an average. Similarly 70_30 yielded better performance considering 4 NUMA nodes out of 8 NUMA nodes, i.e, 30% data is spread the rest three other remote nodes. Beyond 4 NUMA nodes, latency to the remote node dominates the extra bandwidth provided by the remote node. Consequently, we do not see any performance improvement when data is allocated in the remote node beyond 4 NUMA nodes in Westmere.

Summarizing the results from Figures 4 and 5, one can conclude that memory congestion can be alleviated by using the extra memory bandwidth from the target remote nodes. There is no *one common ratio* in distributing the data across the local and remote nodes to achieve maximum performance. That is, different workloads prefer different ratios depending on the hardware configuration. Thus, we need a mechanism which can dynamically identify how congested different shared resources in a NUMA system are oblivious to the

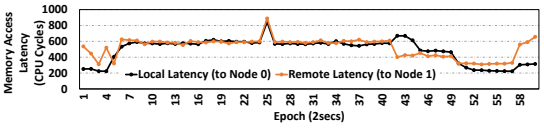


Fig. 6: CAMA latencies for NPB IS on Haswell.

hardware configuration. These shared resources include local memory bandwidth, interconnect bandwidth, and the remote memory bandwidth. The proposed scheme should be successful in identifying congestion oblivious to the underlying hardware configuration. Hence, such a scheme should estimate congestion accurately in diversified environments employing:

- homogeneous/heterogeneous link bandwidth interconnects like AMD Bulldozer [20] and
- homogeneous/heterogeneous bandwidth memories including DRAM [31, 32], AMD's HBM [33], Intel's MCDRAM [34], non-volatile memories like Intel 3D XPoint [35] and futuristic phase-change memory (PCM) [36]

IV. Dynamic Latency Probing

Past congestion detection techniques [1, 3, 4, 19] relied on reading the performance counters to estimate the per-socket memory intensity. However, since performance counters are *shared resources*, these mechanisms limit the number of performance counters [37] that can be used for other runtime optimizations. We implemented our dynamic probing mechanism in VMware's ESXi[3] on a real system, and observed that it gives the end-to-end information on how congested different shared resources in a NUMA system are very accurately.

Algorithm 1 Pseudo-Code for Dynamic Latency Probing.

```
1: procedure NUMA_ProbeDynLatencies(my_numa_node)
2:     for each numa_node do
3:         NUMA_ProbeDynLatency(my_numa_node, numa_node);
4:     end for
5: end procedure
6:
7: procedure NUMA_ProbeDynLatency(srcNode, tgtNode)
8:     totalAccessCycles = 0
9:     for each page in NUM_PROBED_PAGES do
10:        startCycle = RDTSC();
11:        for each probe in NUM_PROBES_PER_PAGE do
12:            /* Access the tgtNode pages (by issuing processor-loads) in a non-
       cacheable manner using volatile constructs. */
13:        end for
14:        endCycle = RDTSC();
15:        /* Let pageAccCycles represent the elapsed cycles */
16:        totalAccessCycles + = pageAccCycles;
17:    end for
18:    /* Measure and record the avgAccessLatency in the table corresponding to row
       srcNode and column tgtNode */
19:    currNodeLatArray[tgtNode] = average;
20: end procedure
```

Our dynamic latency probing mechanism is based on periodically accessing the *non-cacheable pages* in each node from every node. Algorithm 1 presents the pseudo-code for our dynamic probing mechanism. *NUMA_ProbeDynLatencies* procedure is invoked periodically on each NUMA node through timer callbacks already implemented in ESXi. From every source node, represented by *srcNode* in the pseudo-code, *NUMA_ProbeDynLatency* routine is invoked by ESXi passing

[3]We would like to emphasize again that though we implemented and evaluated the mechanism in ESXi, this mechanism is **general and can be adapted by any runtime or OS** and is **not** limited to ESXi.

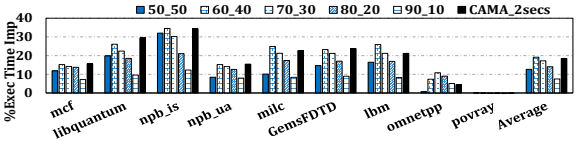


Fig. 8: CAMA results for Intel Haswell processor.

the target node as an argument. To ensure only one Physical CPU (PCPU) per node probes pages from the target nodes, *NUMA_ProbeDynLatencies* callback code is only scheduled on the first PCPU of every NUMA node, except for the node where the current Timing Loop code is being executed. On this node, instead of scheduling the probing code on first PCPU, our dynamic probing latency code is executed on the same PCPU. Such an optimization prevents the pre-emption of the code currently being executed on the first PCPU for the current node. This can be observed in the pseudo-code for the timing loop presented in Algorithm 2. This timing loop in Algorithm 2 is responsible for scheduling the dynamic latency probing code in Algorithm 1 on each node periodically.

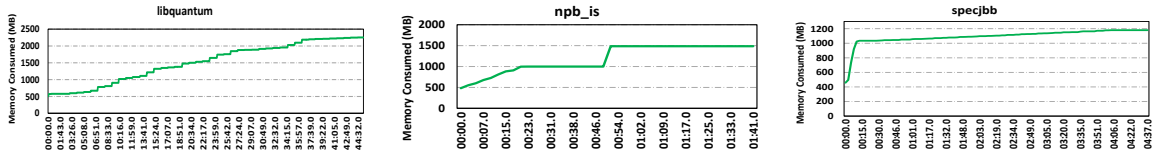
Algorithm 2 Pseudo-Code for Dynamic Latency Probing.

```
    for each numa_node in TOTAL_NUMA_NODES do
2:      currNumaNode = NUMA_GetNumaNodeNum(MY_CPU);
        if numa_node == currNumaNode then
4:          NUMA_ProbeDynLatencies(currNumaNode);
        else
6:          fCPU=NUMA_GetFirstCPUOnNode(numa_node);
            Timer_Add(fCPU, NUMA_ProbeDynLatencies, numa_node);
8:      end if
    end for
```

As can be observed in lines 2-4 for procedure *NUMA_ProbeDynLatencies* in Algorithm 1, the for-loop code passes each node as the target node for procedure *NUMA_ProbeDynLatency*. Consequently, the source node can itself be the destination target node. In such a scenario, the PCPU running on source node accesses pages in local node.

Having understood how the dynamic probing latency code is triggered on each NUMA node, we can now look at the latency probing algorithm itself which is covered in procedure *NUMA_ProbeDynLatency* in Algorithm 1. We modified the ESXi code to allocate extra specified set of 'probe pages' on each NUMA node. Each 'probed page' is 4KB[4] in size and is used only by ESXi for probing periodically. In the dynamic probing algorithm, these probed pages allocated per NUMA node are accessed using "volatile" construct so that the processor loads generated by accessing these probed pages do not get cached in the on-chip cache hierarchy. Hence, accesses to these probed pages always result in memory access to corresponding NUMA nodes. Since processor stores going to the memory are buffered in a separate write-queue and are drained based on the low/high watermarks set at the MC, non-cacheable stores do not truely reflect the congestion of the memory subsystem. As a result, our probing code only issues non-cacheable processor-loads to to estimate congestion. The for-loop in lines 9-19 of Algorithm 1 represents the PCPU on the source NUMA node accessing the probed pages from the target NUMA node. We use RDTSC() function to read the time-stamp counter (TSC) provided by the hardware to record the processor cycles lapsed in accessing these probed pages.

[4]We do not consider large-pages in our evaluation, but our mechanism holds equally well for large pages.



(a) (b) (c)

Fig. 7: Esxtop output showing the memory allocation behavior of applications over time (in minutes).

Note that the dynamic probing code itself is an additional overhead incurred by ESXi. There is a clear *trade-off* on how frequently the probing code can be triggered versus the accuracy of the memory latency information provided by probing. Probing code triggered very frequently not just preempts the execution of VM(s) running on the corresponding PCPUs but also causes additional memory traffic. However, if the probing code is sampled very infrequently, the memory latencies reported by the dynamic latency probing might become stale and may not be useful in alleviating memory congestion and sometimes could result in a proposed optimization to degrade the overall system performance. Two parameters in our algorithm effect the accuracy vs performance trade-off, viz, *sampling interval* and *number of memory probes*.

In the next three sections, we show how dynamic probing is used to guide memory allocation and migration strategies. Though our mechanism can be used in migrating VMs, we do *not* consider VM migration for the following reasons:

- VM migration is often an expensive operation, since migrating VM not only involves migrating the execution context, but also migrating the entire memory footprint of the corresponding VM to the remote node. Migrating the entire memory footprint is an expensive operation in terms of both energy and performance considering how quickly the memory footprint is growing in the emerging workloads [6, 17]. Migrating just the VM without it's corresponding memory results in overwhelming number of remote node memory accesses thereby degrading performance.
- VM migration from one node to another in highly consolidated environments often necessitates swapping the VMs between the source and destination nodes. This is due to the unavailability of a free PCPU that can excute the migrated VM on the target node. This VM swapping is an expensive operation as it involves not just migrating the memory, but also the cost involved in other hardware structures including: flushing the deep processor pipelines, disrupting the branch predictors, flushing TLB entries, and the locality lost in private L1D/L1I and last-level caches. And such a cost increases with increase in number of VMs to be migrated (swapped) and also by the frequency of migration.

V. Congestion-Aware Memory Allocation (CAMA)

Before we look into the optimizations based on the dynamic latency probing, Figures 7a, 7b and 7c plot the memory allocation behavior of libquantum, npb_is and specjbb applications, respectively, collected using *esxtop* utility. As can be observed, in these applications, the memory footprint grows over time and hence provides an opportunity for the memory allocator to allocate some of the data on the remote nodes dynamically when the local node is congested. In this section, we discuss Congestion-Aware Memory Allocation (CAMA), which guides allocation of data on the remote node dynamically.

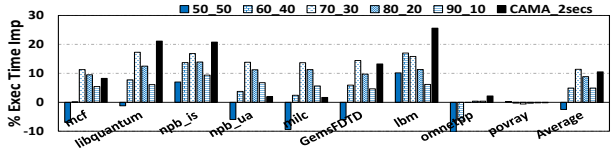


Fig. 9: CAMA results for Intel Westmere processor.

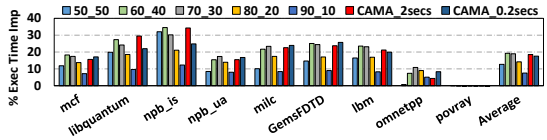
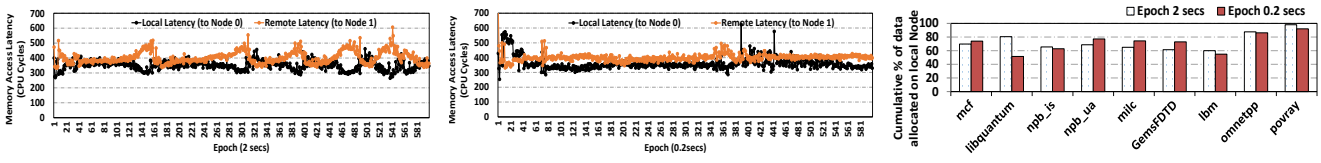


Fig. 10: 0.2 secs epoch CAMA Improvements on Haswell.

Upon receiving a memory allocation request from a VM running on a source node, CAMA decides the target node based on the latencies incurred from the source node to the other nodes in the previous epoch. The target node chosen for allocation in CAMA is the one which incurred the lowest probed latency in the previous epoch. The intuition is that, the probed latencies recorded at the end of previous epoch gives an approximate idea of how congested different shared resources would be in the current epoch. This simple modification in the ESXi memory allocator can automatically consider the end-to-end access latencies and can account for the congestion on different shared resources. The shared resources include the local nodes' memory controllers, the inter-socket interconnect (Intel QPI/AMD HT interconnect), and the remote nodes' memory controllers. Also, this simple change in the memory allocator is flexible enough to pick the local node automatically when the access latency to the local node is lower.

In Figure 8, the last bar for each workload shows the performance improvement for the homogeneous workloads with CAMA. All the results presented are *normalized* to the basecase, where all the memory is allocated in the local node. The other bars representing the static allocation are shown there for comparison. The epoch duration used in these experiments is 2 secs, that is, the dynamic latencies are updated every 2 secs. As can be observed, CAMA improves the performance by 18.49% on an average and a maximum of up to 34.2% for npb_is, and also on an average CAMA is within 1% compared to the best in static allocation schemes. Figure 6 shows the dynamic latencies measured on our Haswell system from Node-0 to both Node-0 and Node-1 for CAMA in ESXi. One can see from this figure that the local and remote node latencies are almost the same at every epoch for CAMA unlike in Figure 3 where the local node latency is much higher than the remote node latency. Figure 9 shows the performance improvements in for CAMA on Intel Westmere processor; the static allocation results are reproduced for comparison. It can be observed that on an average CAMA improves the



(a) (b) (c)

Fig. 11: Dynamic latencies for VMs running mcf with epoch duration of (a) 2 secs, (b) 0.2 secs and (c) Cumulative percentage of data allocated on local node in Haswell by CAMA.

performance by 10% for Westmere. Also, from Figures 8, 9 and 11c, comparing the static allocation and CAMA results, we note that same percentage of data allocated on local node could result in varying performance improvements.

Impact of epoch duration on CAMA: Since we use the probed latencies from the previous epoch to govern the memory allocations in the current epoch, one crucial parameter in our setup which governs the performance improvements is the epoch duration itself. Since our epoch of 2 secs translates to billions of core cycles (with a processor frequency in the order of GHz), it may be beneficial to trigger the probed latencies more frequently. Probing more frequently can give us the more up-to-date congestion information and can help the memory allocator to respond to the dynamic modulations in the application phase behavior more rapidly. Figure 10 shows the performance benefits when we sample the probing code more frequently at 0.2secs epoch on the Haswell processor. For high memory-intensive applications like libquantum, npb_is, lbm we can see that the performance degrades with 0.2 secs epoch compared to 2 secs epoch. This degradation in performance is due to the fact that the probed memory accesses interfere with applications' on-demand memory accesses resulting in increased memory access latencies for the on-demand memory requests. The performance degradation in these applications is as high as 10% for npb_is compared to the case with epoch duration of 2 secs. However, applications like mcf, npb_ua, milc, GemsFDTD and omnetpp benefit from 0.2 secs epoch duration. This improvement in performance is because CAMA could adjust to subtle changes in congestion rapidly.

Figures 11a and 11b show the memory access latencies for homogeneous mcf workload for 2 secs and 0.2 secs epochs, respectively. For an epoch duration of 2 secs in Figure 11a, we can see the bubbles formed due to gap in latencies between the local and remote nodes around epochs 141, 301, 381, 481 and 541. This result indicates that there is still some scope for improvement when using an epoch duration of 2 secs. For epoch duration of 0.2 secs, one could see that the latencies to local and remote nodes match at every instant. This is because, all the shared resources viz., local memory bandwidth, interconnect bandwidth and remote bandwidth are utilized optimally making the end-to-end latencies to match at every instant. Hence, in workloads that benefit from smaller epoch of 0.2 secs, the performance improvement is as high as 4.5% for omnetpp, compared to an epoch duration of 2 secs. On an average, an epoch duration of 0.2 secs yields 17.5% increase in performance compared to the basecase and the corresponding performance improvement for an epoch duration of 2 secs is 18.4%. Figure 11c gives the cumulative percentage of data allocated on the local node for epochs of 2 secs and 0.2 secs using CAMA. For both the epochs, high memory-intensive applications like mcf and npb_is have more than 30% of their data in the remote node.

VI. Congestion-Aware Page Migration (CAPM)

CAMA is ineffective in the following scenarios:

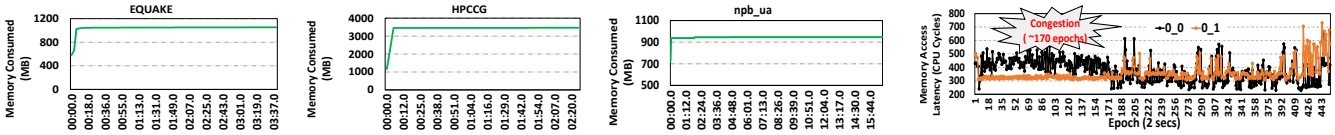
1) Memory system is congested during data allocation, but it is no longer congested when the data is being accessed.
2) Memory subsystem is not congested during data allocation, however, it gets congested during execution of workload.

Figures 12a, 12b and 12c, show the memory allocation behavior of different classes of applications over time. As can be observed, all of the data is allocated in the first few seconds of the program execution. In such scenarios, as local node memory will not be congested initially, CAMA allocates all the data in the local node it cannot alleviate congestion.

Therefore, we need a dynamic mechanism which can fix the incorrect decisions made by CAMA at runtime or augment CAMA to reap maximum overall system performance. Motivated by this, we propose our second optimization, Congestion-Aware Page Migration (CAPM) which dynamically *migrates* pages across nodes based on the probed latencies described in Section IV. In this section, we elaborate on the intricacies in designing CAPM in isolation in the absence of our first proposed optimization CAMA. From Figures 6 and 11b, in presence of congestion, it can be observed that maximum overall performance is obtained when the shared resources are utilized optimally, i.e, when the gap between end-to-end memory latencies of the local node and the remote node is minimum in every epoch. Hence, based on dynamic probed latency in the previous epoch, CAPM tries to migrate pages such that the latencies to the local node and remote node(s) match in every epoch. When local node becomes congested, CAPM migrates the data from local node to the remote node, but when the latency to the remote node increases later, data is migrated back from remote node to local node in next epoch to balance end-to-end latencies. The key factors that effect improvements in CAPM are:

- Candidate pages chosen to migrate from source to destination nodes.
- Number of such candidate pages to be migrated per epoch, referred to as *migration rate*.

Candidate Pages for Migration: Since programs exhibit locality [38] during execution, not all the pages allocated are accessed in all the epochs. In VMware ESXi and traditional OS's like Linux, such pages accessed can be identified by poisoning the page table entry (PTE) by setting its reserved bit similar to [38]. Such a poisoned page is flushed from the TLB and a corresponding processor load/store will incur a TLB miss. Upon a miss, a hardware pagetable walk is triggered



(a) (b) (c) (d)

Fig. 12: Esxtop output showing memory allocation behavior of the applications over time (in minutes) in (a), (b), (c) and (d) Dynamic latencies without any migration.

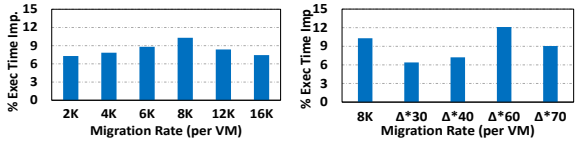
Algorithm 3 Pseudo Code for setting CAPM migration rate.

```
1: currNode = NUMA_GetNumaNodeNumFromCPU(MY_CPU);
2: currNodeLatArray = NUMA_GetDynLatCyclesArray(currNode);
3: /* currNodeLatArray –> Latencies from the current Node to all the other nodes.
   minLatencyNode –> Node incurring minimum latency from current Node. */
4: if currNode ! = minLatencyNode then
5:     /* Local node is congested, pages are to be migrated from local current node to
   the neighboring remote node. */
6:     lat_gap = currNodeLatArray[currNode] - avgLatency;
7:     if lat_gap>(η_thresh*avgLatency) and freespace_available_in_remote_node
   then
8:         Set migration rate here
9:     else
10:        migrationrate = 0;
11:    end if
12: else
13:    /* Neighboring remote NUMA node is congested, pages are to be migrated from
   neighboring remote NUMA node to local current NUMA node. */
14:    lat_gap = currNodeLatArray[neighboringNode] - avgLatency;
15:    if lat_gap>(η_thresh*avgLatency) and freespace_available_in_local_node then
16:        Set migration rate here
17:    else
18:        migrationrate = 0;
19:    end if
20: end if
```

following which a BadgerTrap routine [39] is executed accounting for the page access. BadgerTrap handler further resets (unpoisons) the reserved bit in PTE and caches the translation in TLB and later re-poisons the PTE. Hence the number of badgertraps accounted for each page can be used to distinguish an accessed page from a non-accessed page in a given epoch. From our offline-analysis of each application, we identified that in every epoch, total memory footprint of the accessed pages is in the order of several Mega Bytes (MBs) which span over few thousands of 4KB pages. In CAPM, the candidate pages to be migrated are only chosen from the accessed set of pages. Hence the migration rates used in CAPM will be in the order of few thousands per epoch to alleviate congestion.

Target Migration Rate per epoch: Migrating a candidate page from source node to destination node incurs following steps: (a) allocating a new page in the destination node, (b) copying the entire page contents from the source node memory to the target node memory, (c) shooting down the cached translations inside the TLB and (d) updating the page table entry. Prior work [40] accounted for the TLB shootdown overheads to be as high as 11000 CPU cycles for 16 threads. Considering the above overheads, the target migration rate per epoch, plays a significant role in the overall system performance. Since our CAPM supports migration in both the directions (local to remote and remote to local), the migration rate should be chosen such that the overheads are minimal. A lower migration rate migrates fewer pages per epoch, and hence it will take more time to alleviate congestion, resulting in low improvement in performance. On the other hand, a high migration rate causes more pages to migrate per epoch, thereby alleviating the congestion faster. However, at the end of the epoch, due to migration, the latency to the neighbouring NUMA node will become higher than that to the local NUMA node. As a result, pages are now migrated from the neighbouring node to the local node in the next epoch, thereby causing the pages to migrate back-and-forth between the local and the neighboring remote nodes, causing a *ping-pong effect*. Clearly, such a ping-pong effect is not desirable warranting the migration rate to be chosen carefully.

When the local node is congested, it is clear from Figure 6 that, the maximum performance is achieved when the measured end-to-end latencies between local and neighboring nodes are close to each other in every epoch. Hence, the desired probed latency at every epoch to both the local and remote nodes should be the average of the observed latencies. The pseudo-code for setting the migration rate in CAPM is presented in Algorithm 3. At the end of every epoch, for every node, the algorithm involves identifying the congested node by comparing it's local NUMA node latency with the neighboring remote node. Once a congested node is identified, the lat_gap, defined as difference in the latency to the congested node and the average expected latency is computed, as shown in lines 10 and 18. If the lat_gap exceeds the expected average by a threshold, denoted by η_{thresh}, CAPM migrates pages from the congested node to the neighboring node[5], else not. If η_{thresh} is too low, less latency-gap can be tolerated and thus too many pages could be migrated which could possibly trigger the ping-pong effect. If η_{thresh} is too high, too much latency gap will be tolerated and consequently not enough pages are migrated, resulting in low overall performance. Based on our experimentation on all our workloads, we identified 5% to be a reasonable value for η_{thresh}(used as threshold in this paper).



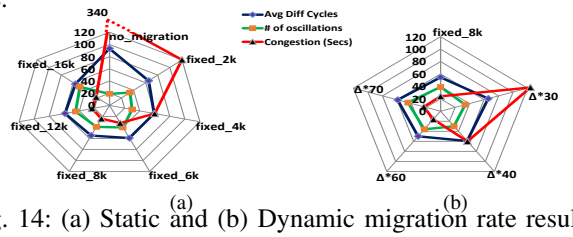
(a) (b)

Fig. 13: (a) Static, (b) Dynamic migration rate results.

To explain various results in CAPM, we use the following heterogeneous workload: equake(2), povray(2), mgrid(2), mcf(2), swim(4). This heterogeneous workload contains multi-programmed and multi-threaded applications with varied memory intensities. Figure 12d represents the dynamic latencies measured in the basecase where all the data is allocated on the local node. It can be observed that the local node is congested for around 170 epochs (340 secs) since the latency

[5]Please note from lines 7 and 15 of Algorithm 3, migration rate will be 0 if there is no free space available in the destination node. Hence, CAPM tries to alleviate congestion opportunistically as long as there is freespace available in the destination node.

to the local node is higher compared to that to the remote node.



(a) (b)

Fig. 14: (a) Static and (b) Dynamic migration rate results.

Once CAPM decides to migrate candidate pages based on the latency gap and η_{thresh}, the next important question is how many candidate pages to migrate per VM per epoch to get the best overall system performance. Migration rates per VM can be set per epoch either statically or dynamically. As covered in "Candidate Pages for Migration" discussion, since the working set sizes of the accessed pages are in the order of several MBs, few thousands of candidate pages need to be migrated per epoch to alleviate congestion. Figure 13a shows the performance improvements for the heterogeneous workloads when migration rate is varied from 2000 to 16000 candidate pages per VM. As the migration rate is increased from 2000 to 8000, performance improvement over the baseline increases from 7.2% to 10.3% and beyond 8000, the performance degrades from 10.3% to 7.45% as the migration rate is further increased from 8000 till 16000. Hence, 8000 pages per epoch per VM is the best migration rate.

This variation can be quantified using following parameters:
1) Congestion duration in seconds,
2) Number of oscillations in the probed latencies, and
3) Average difference between probed latencies per epoch.

The congestion duration is defined as the time in seconds during which the probed latency to the local node is higher than that to the remote node over the entire execution. From Figures 12d and 14a, the congestion duration is around 340 secs for the basecase. Smaller congestion duration indicates that a specific migration rate could mitigate the congestion faster. As the migration rate is changed from 2000 to 16000, the congestion duration changes from 120 secs to 22 secs.

The number of oscillations in probed latencies represents the ping-pong effect in observed latencies due to back-and-forth migrations of pages between NUMA nodes. With its value initialized to zero, once the migration is triggered due to congestion, the value is incremented every time the probed latencies to local and remote nodes cross each other. To get an understanding on how its value is calculated, for example, for the basecase in Figure 12d, initially as there is no congestion, its value remains zero for few epochs as the local node is not congested. As the local node gets congested over time, the first epoch when the latency to local node crosses the remote node latency, its value is incremented to 1 and the value keeps incrementing every time the latencies cross each other. As can be observed in Figure 14a, the number of oscillations for basecase is 19, which is the lowest among the other values for different static migration rates as the data is not migrated in the basecase. The only oscillations that happen in latencies in basecase is because of the variations in application behavior triggering changes in the overall memory intensity of the socket. As the static migration rate increases from 2000 to 16000, the number of oscillations increases from 19 to 49. With the higher migration rate, the pages keep migrating back-and-forth between the local and remote nodes, thereby resulting in higher number of oscillations. For a migration rate of 8000, the number of oscillations suffered is 39.

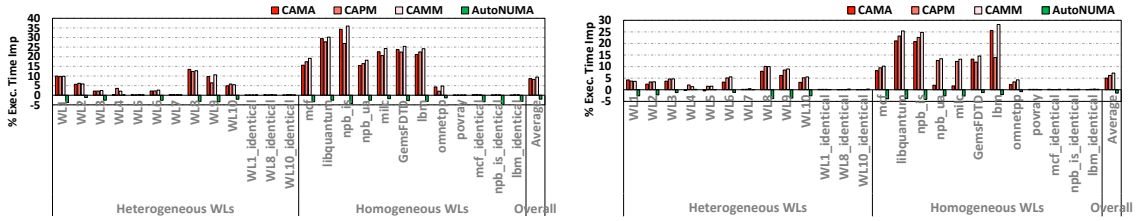
The average difference in CPU cycles between the probed latencies per epoch is defined as the average of absolute difference in probed latencies between the local node and remote nodes at every epoch. Figure 14a shows how the average difference in CPU cycles is effected by the static migration rate. For the basecase, as the data is not migrated, the local node is congested, and hence, the absolute difference in probed latencies per epoch is as high as 93 CPU cycles. As the migration rate is varied from 2000 to 16000, average value changes from 65 cycles to 56 cycles with a minimum value of 54 cycles for a migration rate of 8000.

The maximum performance improvements occur when all the above mentioned parameters experience lower values, which seems to be the case for a migration rate of 8000. Though the static migration rates can improve performance, being agnostic to the gap in probed latencies might not yield the maximum performance. Observing this, we propose using the current absolute latency gap at the end of the epoch, represented by Δ_{cycles}, to calculate the dynamic migration rates so that we migrate more pages if the latency gap is high and fewer pages if the latency gap is not too high. Figure 13b shows the improvement in performance for different dynamic migration rates, and the corresponding values for the three parameters discussed above are plotted in Figure 14b (results for static migration rate 8000 is just shown for comparison). As can be observed, ($\Delta_{cycles} \times 60$) yields maximum system performance with 12.1% in overall system performance[6]. Hence, CAPM is successful in alleviating congestion in scenarios where CAMA is ineffective.

VII. Congestion-Aware Memory Management (CAMM)

In this section, we discuss and evaluate how CAMA and CAPM interact with each other. We refer to this combined scheme, which includes both CAMA and CAPM, as Congestion-Aware Memory Management (CAMM). CAPM employed a dynamic migration rate of $\Delta_{cycles} \times 60$ and the epoch duration used is 2 secs. To give an overall picture of how the overall system performance is impacted by our schemes, we present the full evaluation results for CAMA, CAPM and CAMM using homogeneous and heterogeneous workloads. Tables VI and VII show heterogeneous workloads for Westmere and Haswell respectively. Apart from WL1 to WL10 heterogeneous workloads, we have three additional workloads, WL1_identical, WL8_identical, WL10_identical where all the nodes in a machine execute the same workloads thereby causing equal amount of congestion to their corresponding local memories. Similarly we use mcf_identical, npb_is_identical and lbm_identical homogeneous workloads that result in same amount of congestion in all the nodes of a machine. In

[6]In all our workloads, we observed that this heuristic yields maximum performance.



(a) (b)

Fig. 15: Overall execution time improvements for (a) Intel Haswell and (b) Intel Westmere.

the heterogeneous workloads, since some applications finish execution earlier, a node may not be congested allthrough the workload execution which might not depict some real-world scenarios. To cover such scenarios, we restart applications which finish earlier till the point where each application in the workload finishes execution at least once.

	Workloads	MPKI Category	Restart Enabled?
WL-1	mcf(2), libquantum(3), is(3), milc(2), lbm(2), swim(1)	H	No
WL-2	hpccg(2), milc(1), is(1), libquantum(1), equake(1), specjbb(1), lbm(1)	H+M	No
WL-3	GemsFDTD(4), specjbb(1), equake(1), hpccg(2)	M	No
WL-4	hpccg, povray(2), cc(1), equake(1), mgrid(1), GemsFDTD(1)	M+L	No
WL-5	povray(4), scc(2), mgrid(2)	L	No
WL-6	povray(1), mcf(2), libquantum(2), cc(1), is(1), ua(1)	H + L	No
WL-7	GemsFDTD(1), mcf(1), povray(2), lbm(1), milc(1), specjbb(1), libquantum(1), ua(1), swim(1), mgrid(1)	H + M + L	No
WL-8	mcf(2), libquantum(3), is(3), milc(2), lbm(2), swim(1)	H	Yes
WL-9	hpccg(2), milc(1), is(1), libquantum(1), equake(1), specjbb(1), lbm(1)	H + M	Yes
WL-10	povray(1), mcf(2), libquantum(2), cc(1), is(1), ua(1)	H+ L	Yes

TABLE VI: Heterogeneous workloads for Intel Westmere.

	Workloads	MPKI Category	Restart Enabled?
WL-1	is(6), ua(4), mcf(4), libquantum(5), milc(4), swim(2), omnetpp(2)	H	No
WL-2	GemsFDTD(3), lbm(3), libquantum(2), mcf(3), omnetpp(3), is(4), ua(1), equake(2), swim(1)	M+ H	No
WL-3	GemsFDTD(4), hpccg(3), equake(2), specjbb(3)	M	No
WL-4	equake(3), mgrid(2), GemsFDTD(4), specjbb(1), hpccg(1)	M + L	No
WL-5	povray(8), cc(3), scc(3), mgrid(1)	L	No
WL-6	lbm(2), libquantum(3), mcf(2), milc(3), omnetpp(2),cc(2), scc, is(2), npb_ua(2), mgrid(1), swim(1),	H+L	No
WL-7	GemsFDTD(2), lbm(2), mcf(2), milc(1), omnetpp(2), hpccg(1), cc(1), scc(2), is(1), ua(1), equake(1), swim(1)	H+M+L	No
WL-8	is(6), ua(4), mcf(4), libquantum(5), milc(4), swim(2), omnetpp(2)	H	Yes
WL-9	GemsFDTD(3), lbm(3), libquantum(2), mcf(3), omnetpp(3), is(4), ua(1), equake(2), swim(1)	H+M	Yes
WL-10	lbm(2), libquantum(3), mcf(2), ua(2), mgrid(1) milc(3), omnetpp(2), cc(2), scc, is(2),	H+L	Yes

TABLE VII: Heterogeneous workloads for Intel Haswell.

Overall CAMM results: In Haswell, CAMM improves the overall performance on an averge by 9.5%, while CAMA and CAPM imrove the performance by 8.6% and 8.2% respectively. Similarly, on Westmere, CAMM improved the performance on an average by 7.2%, while CAMA and CAPM improved the performance by 5.1% and 6.2% over the basecase, respectively. Since all the nodes are equally congested in the heterogeneous WL1_identical, WL8_identical, WL10_identical workloads and homogeneous mcf_identical, npb_is_identical, lbm_identical workloads, the CAMM does not allocate/migrate any data to the remote node. This is because the remote node probed latencies are greater than the local node probed latencies in every epoch owing to the additional interconnect traversal latency thereby resulting in 100% local accesses. As a result, there is no improvement in performance in these workloads over the baseline as can be observed in Figures 15a and 15b.

Homogeneous-vs-Heterogeneous WL Results: From figures 15a and 15b it can be observed that homogeneous workloads yield better improvements compared to the heterogeneous ones. This is because, homogeneous workloads have high overall memory intensity compared to the heterogeneous ones. For example, comparing workloads WL8 and npb_is, which are highly memory-intensive in their respective (heterogeneous/homogeneous) categories, the probed latencies look as follows: For WL8: Node0→Node0: 434 cycles; Node0→Node1: 340 cycles; Delta: 94 cycles. For npb_is: Node0→Node0: 630 cycles; Node0→Node1: 375 cycles; Delta: 255 cycles. The homogeneous npb_is workload is around $2.7x$ more congested compared to heterogenous WL8, resulting in larger improvements for npb_is. This variation in memory intensities between homogeneous and heterogeneous workloads is primarily due to application "phase behavior". In a homogeneous workload since the same applications are run on all the cores, at any given moment all the applications are in the same phase, a node is either highly congested or not. For heterogeneous-workloads, since different applications are running concurrently, they are in different phases at any given epoch, thereby causing lower congestion relatively.

Haswell-vs-Westmere Improvements: From Figures 15a and 15b, it can be observed that the improvements on Haswell are relatively higher compared to the Westmere processor. This difference is because Haswell system runs 36 VMs per socket, while Westmere only runs 20VMs per socket. Hence, in our experiments, we observed that Haswell system is more congested overall compared to the Westmere system. Hence, CAMM yields better improvements in Haswell over Westmere.

Dynamic Probing Overhead Analysis: In our dynamic probing code presented in Algorithm 1, NUM_PROBED_PAGES and NUM_PROBES_PER_PAGE play an important role in the overall performance improvement. Too few requests might not give the accurate congestion information, and too many probe-requests will interfere with on-demand requests issued by VMs. Based on experimentation with all our workloads, we determined NUM_PROBED_PAGES and NUM_PROBES_PER_PAGE with values 20 (per-node) and 8 respectively, could capture the end-to-end congestion information accurately. For this configuration, our probing-code accesses 160 cache lines from a node to determine the average end-to-end access latency for every 2 secs epoch. The npb_is (homogeneous) workload (which is the most memory-intensive workload among our workloads) running on Haswell, experienced an average memory access latency of 630 CPU cycles (per memory request) to the congested node (Node-0) in the baseline. In such scenarios, our default probing overhead is 48 μsecs for every 2 secs probing. However, with our proposed CAMM optimizations, average memory access latency reduces significantly, further reducing probing overheads to few μsecs, which is significantly better over prior proposals [3].

Comparison with AutoNUMA: The current version of Linux is NUMA-aware and supports AutoNUMA [41] feature

to minimize remote node accesses. AutoNUMA migrates threads/memory to co-locate threads and data to minimize remote node accesses. It tracks local-vs-remote faults by invalidating few TLB entries, generating page-faults when those pages are accessed. However, unlike CAMM, autoNUMA does not migrate memory pages anymore if a workload incurs 100% local accesses irrespective of the congestion. As can be observed in Figures 15a and 15b (green-bar), congestion-agnostic autoNUMA implemented in ESXi infact degraded the performance over the baseline on an average by 2.1% and 1.4% on Haswell and Westmere respectively, as the additional time is spent servicing the artificially induced page-faults.

VIII. Related Work

NUMA-aware proposals: The numactl [42] feature in Linux provides the user with a flexbility to choose the node on which a task can execute and a node on which data can be allocated statically based on various policies like localalloc, interleaved, etc., which govern where the tasks' data can be allocated. This numactl [42] feature abides by the policy specified by the user and hence is congestion-agnostic. All NUMA-aware proposals [2, 30, 41]–[43] target co-locating tasks and memory on the same node to enhance locality and are agnostic to congestion.
Congestion-aware proposals: The prior congestion-aware proposals [1, 3, 4, 19] employ reading performance counters to estimate the congestion information. Sharanyan et al [3, 4] read various counters to estimate number of memory accesses and use this information to isloate the inter-node coherence traffic and the main memory traffic by reading the performance counters. Liu et al [19] proposed a NUMA overhead-aware buddy allocator. In this work, authors infer congestion by reading performance counters that count the LLC hitrates, cycle loss due to LLC misses, and IPC ratio between local and remote nodes. Sergey et al [1] proposed a dynamic scheduling algorithm, DINO, which migrates threads and the corresponding memory between sockets to mitigate the memory congestion based on profiled information from *perfmon*. Perfmon internally reads performance counters [44]. Almost all the commerical processors available today impose constraints on how many performance counters can be monitored simultaneously (Intel Ivybrige allows only four [3, 37]). Reading these counters itself result in overheads in the order of several μsecs for every 1sec sampling interval [3]. Also, inferring the shared interconnect contention information is not straightforward in processors like AMD as demonstrated by Lepers et al [2], and hence the past proposals do not have one generic mechanism which works on all the machines oblivious to the hardware unlike our CAMM.

IX. Conclusion

We proposed a generic, novel, accurate and practical dynamic probing mechanism which can aid in detecting congestion in a NUMA-based system. Our proposed mechanism does not necessitate reading performing counters to detect and mitigate congestion. We proposed and evaluated CAMA, CAPM and CAMM in ESXi. We observed that CAMM, a combination of CAMA and CAPM improved the overall performance, on an average, by 9.5% and 7.2% in Haswell and Westmere systems, respectively.

Acknowledgment

The authors would like to acknowledge the support of NSF under the grants 1439021, 1439057, 1409095, 1626251, 1629915, 1629129 and 1526750.

References

[1] S. Blagodurov *et al.*, "A case for NUMA-aware contention management on multicore systems," in *USENIX ATC*, 2011.
[2] B. Lepers *et al.*, "Thread and memory placement on NUMA systems: Asymmetry matters," in *USENIX ATC*, 2015.
[3] S. Srikanthan *et al.*, "Data sharing or resource contention: Toward performance transparency on multicore systems," in *USENIX ATC*, 2015.
[4] Sharanyan *et al.*, "Coherence stalls or latency tolerance: Informed cpu scheduling for socket and core sharing," in *USENIX ATC*, 2016.
[5] "Bullion(16-socket) System," https://goo.gl/WTRoKn.
[6] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS*, 2012.
[7] J. B. Kotra *et al.*, "Re-NUCA: A practical nuca architecture for reram based last-level caches," in *IPDPS*, 2016.
[8] Kislal *et al.*, "Location-aware computation mapping for manycore processors," in *PACT*, 2017.
[9] O. Kislal *et al.*, "Cache-aware approximate computing for decision tree learning," in *IPDPSW*, 2016.
[10] H. Zhang *et al.*, "Race-to-sleep + content caching + display caching: A recipe for energy-efficient video streaming on handhelds."
[11] X. Tang *et al.*, "Improving bank-level parallelism for irregular applications."
[12] J. D. Booth *et al.*, "Phase detection with hidden markov models for dvfs on many-core processors," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, 2015.
[13] J. Liu *et al.*, "Network footprint reduction through data access and computation placement in noc-based manycores," in *DAC*, 2015.
[14] K. Swaminathan *et al.*, "Thermal-aware application scheduling on device-heterogeneous embedded architectures," in *VLSID*, 2015.
[15] P. Yedlapalli *et al.*, "Meeting midway: Improving cmp performance with memory-side prefetching," in *PACT*, 2013.
[16] J. Kotra *et al.*, "Quantifying the potential benefits of on-chip near-data computing in manycore processors," in *MASCOTS*, 2017.
[17] L. Wang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *HPCA*, 2014.
[18] J. B. Kotra *et al.*, "Hardware-software co-design to mitigate dram refresh overheads: A case for refresh-aware process scheduling," in *ASPLOS*, 2017.
[19] M. Liu *et al.*, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," in *ISCA*, 2014.
[20] "AMD Bulldozer," http://goo.gl/otbg9J.
[21] "Intel Haswell Processor," http://goo.gl/xIiqKY.
[22] "Intel Westmere processor," http://goo.gl/UGPBsI.
[23] "Westmere config," https://goo.gl/CjGO0X.
[24] "NAS," https://goo.gl/dA36i5.
[25] "SPEC CPU 2006," http://www.spec.org/cpu2006.
[26] "SPEC OMP 2001," https://www.spec.org/omp2001/,.
[27] "HPCCG," https://mantevo.org/default.php.
[28] "SPECJBB," https://www.spec.org/jbb2005/.
[29] G. Slota *et al.*, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *IPDPS*, 2014.
[30] "VMware ESX Server 2," white paper.
[31] "JEDEC," DDR4 SDRAM Standard, 2012.
[32] "JEDEC," Low Power Double Data Rate 3 (LPDDR3), 2012.
[33] "AMD HBM," http://goo.gl/iVHexT.
[34] "Intel MCDRAM," https://goo.gl/iGQG32.
[35] "Intel 3D XPoint Memory," http://goo.gl/04n7Ksl.
[36] "Phase-change Memory," https://goo.gl/xOKMDF.
[37] "Intel performance analysis guide," https://goo.gl/y21oEB,.
[38] N. Agarwal *et al.*, "Thermostat: Application-transparent page management for two-tiered main memory," in *ASPLOS*, 2017.
[39] J. Gandhi *et al.*, "Badgertrap: A tool to instrument x86-64 tlb misses," in *CAN*, 2014.
[40] C. Villavieja *et al.*, "Didi: Mitigating the performance impact of TLB shootdowns using a shared tlb directory," in *PACT*, 2011.
[41] "Linux AutoNUMA," https://lwn.net/Articles/488709/.
[42] "Linux numactl," https://goo.gl/FS9mHs.
[43] Q. Ali *et al.*, "Power aware NUMA scheduler in vmware's esxi hypervisor," in *IISWC*, 2015.
[44] "PerfMon," https://goo.gl/RvLxqE.