Thread and Memory Placement on NUMA Systems: Asymmetry Matters

Baptiste Lepers
Simon Fraser University

Vivien Quéma
Grenoble INP

Alexandra Fedorova
Simon Fraser University

Abstract

It is well known that the placement of threads and memory plays a crucial role for performance on NUMA (Non-Uniform Memory-Access) systems. The conventional wisdom is to place threads close to their memory, to collocate on the same node threads that share data, and to segregate on different nodes threads that compete for memory bandwidth or cache resources. While many studies addressed thread and data placement, none of them considered a crucial property of modern NUMA systems that is likely to prevail in the future: *asymmetric interconnect*. When the nodes are connected by links of different bandwidth, we must consider not only whether the threads and data are placed on the same or different nodes, but how these nodes are connected.

We study the effects of asymmetry on a widely available x86 system and find that performance can vary by more than 2× under the same distribution of thread and data across the nodes but different inter-node connectivity. The key new insight is that the best-performing connectivity is the one with the greatest total bandwidth as opposed to the smallest number of hops. Based on our findings we designed and implemented a dynamic thread and memory placement algorithm in Linux that delivers similar or better performance than the best static placement and up to 218% better performance than when the placement is chosen randomly.

1 Introduction

Typical modern CPU systems are structured as several CPU/memory nodes connected via an interconnect. These architectures are usually characterized by non-uniform memory access times (NUMA), meaning that the latency of data access depends on *where* (which CPU-cache or memory node) the data is located. For this reason, the placement of threads and memory plays a crucial role in performance. This property inspired many NUMA-aware algorithms for operating systems. Their insight is to place threads close to their memory [19, 12, 9], spread the memory pages across the system to avoid the overload on memory controllers and interconnect links [12], to collocate data-sharing threads on the same node [30, 31] while avoiding memory controller contention [7, 31, 10], and to segregate threads competing for cache and memory bandwidth on different nodes [34].

Further, modern operating systems aim to reduce the number of hops used for thread-to-thread and thread-to-memory communication. When balancing the load across CPUs, Linux first uses CPUs on the same node, then those one hop apart and lastly two or more hops apart. These techniques assume that the interconnect between nodes is symmetric: given any pair of nodes connected via a direct link, the links have the same bandwidth and the same latency. *On modern NUMA systems this is not the case*.

Figure 1 depicts an AMD Bulldozer NUMA machine with eight nodes (each hosting eight cores). Interconnect links exhibit many disparities: (i) Links have different bandwidths: some are 16-bit wide, some are 8-bit wide; (ii) Some links can send data faster in one direction than in the other (i.e., one side sends data at 3/4 the speed of a 16-bit link, while the other side can only send data at the speed of an 8-bit link). We call these links *16/8-bit links*; (iii) Links are shared differently. For instance the link between nodes 4 and 3 is only used by these two nodes, while the link between nodes 2 and 3 is shared by nodes 0, 1, 2, 3, 6 and 7; (iv) Some links are unidirectional. For instance node 7 sends requests directly to node 3, but node 3 routes its answers via node 2. This creates an asymmetry in read/write bandwidth: node 7 can write at 4GB/s to node 3, but can only read at 2GB/s.

The asymmetry of interconnect links has dramatic and at times surprising effects on performance. Figure 2 shows the performance of 20 different applications on

Machine A (Figure 1)[1]. Each application runs with 24 threads and so it needs three nodes to run on. We vary *which* three nodes are assigned to the application and hence the *connectivity* between the nodes. The relative placement of threads and memory on those nodes is *identical* in all configurations. The only difference is *how the chosen nodes are connected*. The figure shows the performance on the best-performing and the worst-performing subset of nodes for that application compared to the average (obtained by measuring the performance on all 336 unique subsets of nodes and computing the mean). We make several observations. First, the performance on the best subset is up to 88% faster than the average, and the performance on the worst subset is up to 44% slower. Second, the maximum performance difference between the best and the worst subsets is 237% (for facerec). Finally, the mean difference between the best and worst subsets is 40% and the median 14%. In the following section we demonstrate that these performance differences are caused by the asymmetry of the interconnect between the nodes.

This work makes the following contributions:

- We quantify and characterize the effects of asymmetric interconnect on a commercial x86 NUMA system. The key insight is that the best-performing connectivity is the one with the greatest total bandwidth as opposed to the smallest number of hops.
- We design, implement and evaluate a new algorithm that dynamically picks the best subset of nodes for applications requiring more than one node. This algorithm places the clusters of threads and their memory to ensure that the most intensive CPU-to-CPU or CPU-to-memory communication occurs between the best-connected nodes. Our evaluation shows that this algorithm performs as well as or better than the best set of nodes chosen statically.
- Our implementation revealed a limitation in hardware counters, which prevented us from having certain flexibilities in the algorithm. We discuss them and make suggestions for improvements.

The paper is structured as follows. Section 2 studies the impact of interconnect asymmetry and discusses challenges in catering to this phenomenon in an operating system. Section 3 discusses current architectural trends and shows that machines are becoming increasingly asymmetric. Section 4 presents our algorithm, and Section 5 reports on the evaluation. Section 6 discusses related work, and Section 7 provides a summary.

[1]Additional details about the applications and the machine are provided in Section 5.

2 The Impact of Interconnect Asymmetry on Performance

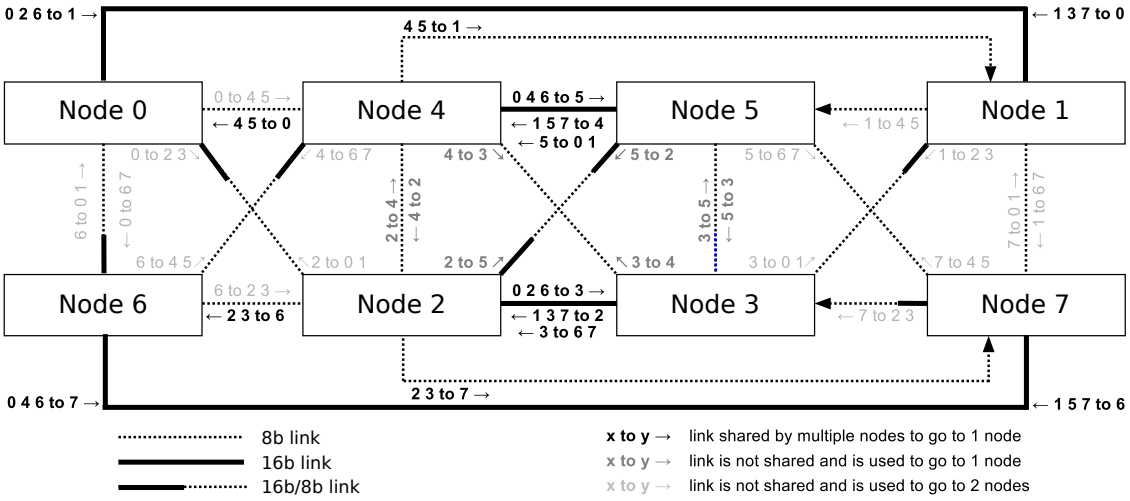
To explain the reasons behind the performance reported in Figure 2, Figure 3 shows the memory latency measured when the application runs on the best and worst node subsets relative to the latency averaged across all 336 possible subsets. We can see that memory accesses performed by facerec are approximately 600 cycles faster when running on the best subset of nodes relative to the average, and 1400 cycles faster relative to the worst. We can see that the latency differences are tightly correlated with the performance difference between configurations. The applications that are the most affected by the choice of nodes on which to run are also those with the highest difference in the memory latencies.

To further understand the cause of very high latencies on "bad" configurations we analyzed streamcluster – an application from the Parsec [26] benchmark suite, which was among the most affected by the placement of its threads and memory. In the following experiment we run streamcluster with 16 threads on two nodes. Table 1 presents the salient metrics for each possible two-node subset. Depending on which two nodes we chose, we observe large (up to 133%) disparities in performance. The data in Table 1 leads to several crucial observations:

- As shown earlier, performance is correlated with the latency of memory accesses.
- Surprisingly, the latency of memory accesses is not correlated with the number of hops between the nodes: some two-hop configurations (shown in bold) are faster than one-hop configurations.
- The latency of memory accesses is actually correlated with the *bandwidth* between the nodes. Note that this makes sense: the difference between one-hop vs. two-hop latency is only 80 cycles when the interconnect is nearly idle. So a higher number of hops alone cannot explain the latency differences of thousands of cycles.

Bandwidth between the nodes matters more than the distance between them.

So the problem of choosing a "good" subset of nodes is essentially the problem of ***the placement of threads and memory pages on a well-connected subset of nodes***. When an application executes on only two nodes on a machine similar to the one used in the aforementioned experiments, the placement on the nodes connected with the widest (16-bit) link is always the best because it maximizes the bandwidth and minimizes the latency between the nodes. However, when an application needs more than two nodes to run, no configuration exists with 16-bit links between every pair of nodes, so we must decide



(Machines A and B)

Figure 1: Modern NUMA systems, with eight nodes. The width of links varies, some paths are unidirectional (e.g., between 7 and 3) and links may be shared by multiple nodes. Machine A has 64 cores (8 cores per node - not represented in the picture) and machine B has 48 cores (6 cores per node). Not shown in the picture: the links between nodes 4 and 1 and between nodes 2 and 7 are bidirectional on machine B. This changes the routing of requests from node 7 to 2 and node 1 to 4.

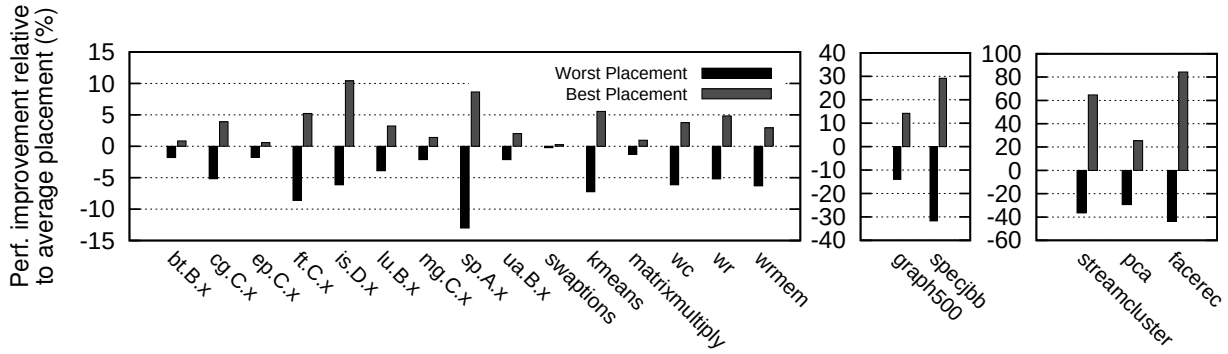


Figure 2: Performance difference between the best, and worst thread placement with respect to the average thread placement on Machine A. Applications run with 24 threads on three nodes. Graph500, specjbb, streamcluster, pca and facerec are highly affected by the choice of nodes and are shown separately with a different y-axis range.

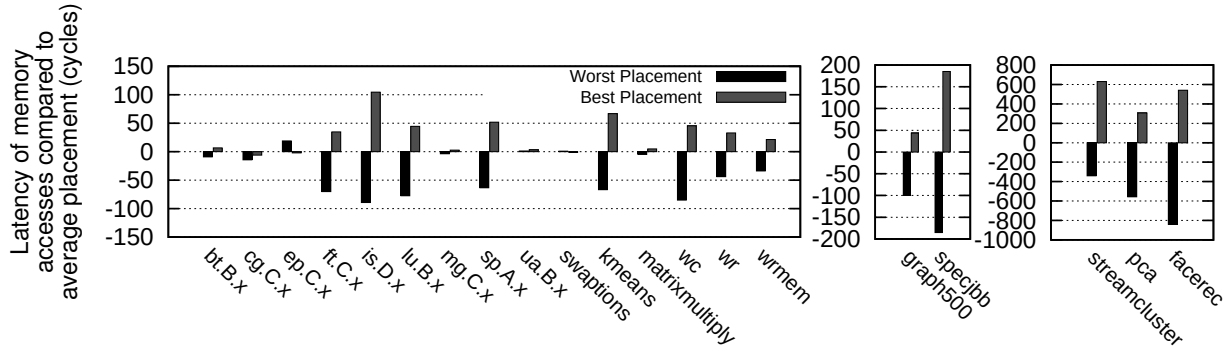
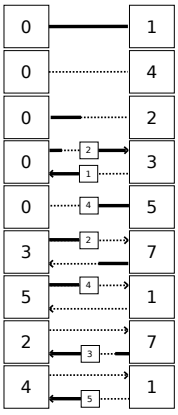


Figure 3: Difference in latency of memory accesses between the best, and worst thread placement with respect to the average node configuration on Machine A. Positive numbers mean that memory accesses are faster than the average.



Master thread node	Execution Time (s)	Diff with 0-1 (%)	Latency of memory accesses (cycles) (compared to 0-1(%))		% accesses via 2-hop links	Bandwidth to the "master" node (MB/s)
-	148	0%	750		0	5598
-	228	56%	1169	(56%)	0	2999
0	228	56%	1179	(57%)	0	2973
2	168	15%	855	(14%)	0	4329
0	340	133%	1527	(104%)	98	1915
3	**185**	27%	1040	(39%)	98	3741
0	340	133%	1601	(113%)	98	1903
5	**228**	56%	1206	(61%)	98	2884
3	185	27%	1020	(36%)	0	3748
7	338	132%	1614	(115%)	98	1928
1	338	132%	1612	(115%)	98	1891
5	230	58%	1200	(60%)	0	2880
2	**167**	15%	867	(16%)	98	3748
7	225	54%	1220	(63%)	0	3014
4	230	58%	1205	(60%)	0	2959
1	**226**	55%	1203	(60%)	98	2880

Table 1: Performance of streamcluster executing with 16 threads on 2 nodes on machine A. The performance depends on the connectivity between the nodes on which streamcluster is executing and on the node on which the master thread is executing. Numbers in bold indicate 2-hops configurations that are as fast or faster than some 1-hop configurations.

which nodes to pick. When there is more than one application running, we need to decide how to allocate the nodes among multiple applications.

Nodes	% perf. relative to best subset	
	streamcluster	SPECjbb
0, 1, 3, 4, and 7	-64%	0% (best)
2, 3, 4, 5, and 6	0% (best)	-9.4%

Table 2: Performance of streamcluster and SPECjbb on two different set of nodes on machine A, relative to the best set of nodes for the respective application.

In this paper, we present a new thread and memory placement algorithm. Designing such an algorithm for asymmetrically connected NUMA systems is challenging for the following reasons:

Efficient online measurement of communication patterns is challenging: The algorithm must measure the volume of CPU-to-CPU and CPU-to-memory communication for different threads in order to determine the best placement when we cannot run the entire application on the best connected nodes. This measurement process must be very efficient, because it must be done continuously in order to adapt to phase changes.

Changing the placement of threads and memory may incur high overhead: Frequent migration of threads may be costly, because of the associated CPU overhead, but most importantly because cache affinity is not preserved. Moreover, when threads are migrated to "better" nodes, it might be necessary to migrate their memory in order to avoid the overhead of remote accesses and overloaded memory controllers. Migrating large amounts of memory can be extremely costly. Thus, thread migration must be done in a way that minimizes memory migration.

Accomodating multiple applications simultaneously is challenging: Applications have different communication patterns and are thus differently impacted by the connectivity between the nodes they run on. As an illustration, Table 2 presents the performance of streamcluster and SPECjbb executing on two different sets of five nodes (the best set of nodes for the two applications, respectively). The two applications behave differently on these two sets of nodes: streamcluster is 64% slower on the best set of nodes for SPECjbb than on its own best set. The algorithm must, therefore, determine the best set of nodes for every application. Furthermore, it cannot always place each application on its best set of nodes, because applications may have conflicting preferences.

Selecting the best placement is combinatorially difficult: The number of possible application placements on an eight-node machine is very large (e.g., 5040 possible configurations for four applications executing on two nodes). So, (i) it is not possible to try all configurations *online* by migrating threads and then choosing the best configurations, and (ii) doing even the simplest computation involving "all possible placements" can still add a significant overhead to a placement algorithm.

Before describing how we addressed these challenges, we briefly discuss architectural trends and the increasing impact of interconnect asymmetry.

3 Architectural trends

Asymmetric interconnect is not a new phenomenon. Nevertheless, we show in this section that its effects on

performance are increasing as machines are built with more nodes and cores. For that purpose, we measured the performance of streamcluster on four different asymmetric machines: two recent machines with 64 and 48 cores respectively, and 8 nodes (Machines A and B, Figure 1), and two older machines with 24 and 16 cores respectively, and 4 nodes (Machines C and D, not depicted). Machines A and B have highly asymmetric interconnect; Table 1 lists all possible interconnect configurations between 2 nodes. Machines C and D have a less pronounced asymmetry. Machine C has full connectivity, but two of the links are slower than the rest. Machine D has links with equal bandwidth, but two nodes do not have a link between them.

Table 3 shows the performance of streamcluster with 16 threads on the best-performing and the worst-performing set of nodes on each machine. The performance difference between the best and worst configurations increases with the number of cores in the machine: from 3% for the 16-core machine to 133% for the 64-core machine. We explain this as follows: (i) On the 16-core Machine D, the only difference between configurations is the longer wire delay between the nodes that are not connected via a direct link. This delay is not significant compared to the extra latency induced by bandwidth contention on the interconnect. (ii) The CPUs on 24-core Machine C have a low frequency compared to the other machines. As a result, the impact of longer memory latency is not as pronounced. More importantly, the network on this machine is still a fully connected mesh, so there is less asymmetry than on Machines A and B. (iii) The 48- and 64-core Machines B and A offer a wider range of bandwidth configurations, which increases the difference between the best and the worst placements. The 64-core machine is more affected than the 48-core machine because it has more cores per node, which increases the effects of bandwidth contention.

If this trend holds across different machines and architectures, then it is clear that the effects of asymmetry can no longer be ignored.

Machine	Best time	Worst time	Difference
A (64 cores)	148s	340s	133%
B (48 cores)	149s	277s	85%
C (24 cores)	171s	229s	33%
D (16 cores)	255s	262s	3%

Table 3: Performance of streamcluster executing on 2 nodes on machine A, B, C, and D. The performance of streamcluster depends on the placement of its threads. The impact of thread placement is more important on recent machines (A and B) than on older ones (C and D).

4 Solution

4.1 Overview

We designed *AsymSched*, a thread and memory placement algorithm that takes into account the bandwidth asymmetry of asymmetric NUMA systems. *AsymSched*'s goal is to maximize the bandwidth for *CPU-to-CPU communication*, which occurs between threads that exchange data, and *CPU-to-memory communication*, which occurs between a CPU and a memory node upon a cache miss. To that end, *AsymSched* places threads that perform extensive communication on relatively well-connected nodes and places the frequently accessed memory pages such that the data requests are either local or travel across high-bandwidth paths.

AsymSched is implemented as a user level process and interacts with the kernel and the hardware using system calls and `/proc` file system, but could also be easily integrated with the kernel scheduler if needed.

AsymSched continuously monitors hardware counters to detect opportunities for better thread placements. The thread placement decision occurs every second, and *AsymSched* only migrates threads when the benefits of migration is expected to exceed its overhead. The placement of memory pages follows the placement of threads.

AsymSched relies on three main techniques to manage threads and memory: (i) **Thread migration**: changing the node where a thread is running. (ii) **Full memory migration**: migrating all pages of an application from one node to another. Full memory migration is performed using a new system call that we present in Section 4.3. (iii) **Dynamic memory migration**: migrating only the pages that an application actively accesses. Dynamic memory migration uses *Instruction-Based Sampling* (IBS), a profiling mechanism available in AMD processors[2], to sample memory accesses and to identify the most frequently accessed pages. Then, the pages that are not shared are migrated to the node that accesses them. Shared pages are spread across multiple nodes. We use the same algorithm and techniques described in [12]; we therefore omit further details on dynamic memory migration.

4.2 Algorithm

AsymSched relies on 3 components. The *measurement* component continuously computes salient metrics. The *decision* component uses these metrics to periodically compute the best thread placements. The *migration* component migrates threads and memory. Table 4 presents the definitions relevant to *AsymSched*. Algorithm 1 summarizes the algorithm.

[2]Intel processors have a similar mechanism called *Precise Event-Based Sampling* (PEBS).

Per cluster (C) statistics	
C_{rbw}	Remote bandwidth: the number of memory accesses performed by threads in the cluster to another node, i.e., *remote accesses*.
C_{weight}	"Weight" of the cluster. Clusters with the highest weights are scheduled on the nodes with the highest interconnect bandwidth. By default $C_{weight} = log(C_{rbw})$.
$C_{bw}(P)$	Maximum bandwidth of C threads on placement P.
Per placement (P) statistics	
P_{wbw}	Weighted total bandwidth of P. Is equal to the sum of the $C_{bw}(P) * C_{weight}$ for every placed cluster C.
P_{mm}	Amount of memory that has to be migrated to use this placement.
Per application (A) statistics	
A_{tm}	Time already spent migrating memory.
A_{tt}	Dynamic running time of the application.
$A_{mm}[node]$	Resident set size of the application, per node.
A_{oldA}	Percentage of memory accesses performed on nodes on which the application *was* scheduled but is no longer scheduled on.

Table 4: Definitions relevant to *AsymSched*.

Measurement. *AsymSched* continuously gathers the metrics characterizing the volume of CPU-to-CPU and CPU-to-memory communication. On our experimental system there is a single counter that captures both: it measures the number of *data accesses performed by a CPU to a given* node and includes both the accesses to cached data (CPU-to-CPU communication) and to the data located in RAM (CPU-to-memory communication). Ideally, we would like to measure the communication volume from every CPU to every other CPU, however the counters available on AMD systems do not offer this opportunity. One alternative is to use AMD's Instruction Based Sampling (IBS)[3]. Unfortunately, to accurately track CPU-to-CPU communication, IBS requires a high sampling rate, and that introduces too much overhead. Lightweight Profiling (LWP), a new profiling facility of AMD processors, has a smaller overhead, but is only partially implemented in current processors. Despite these limitations, we believe that it is only a matter of time until they are addressed in the mainstream hardware, so for the time being we use the following work-around.

The algorithm described below relies on detecting which threads share data. Since we can only practically measure the communication between a CPU and a remote node, but not CPU-to-CPU communication (either across or within nodes), we make the following simplifying assumptions: *(a)* a thread may share data with any other thread running on the same node, *(b)* if there is a high volume of communication between a CPU and a node, a thread running on that CPU may share data with any thread of the same application on that node. To reduce the occurrence of situations where we assume data sharing while in reality there is none, we initially collocate threads from the same application on the same node, to the extent possible. Data sharing is far more common between threads from the same application than between threads from different applications.

The downside of this simplifying assumption is that we may unnecessarily keep a group of threads collocated on the same node even if they do not share data. But there is also an important benefit: characterizing the communication in terms of CPU-to-node keeps the number of sharing relationships to consider small and reduces the complexity of the algorithm.

Decision. The following description relies on definitions in Table 4. **Step 1**: *AsymSched* groups threads of the same application that share data in virtual *clusters*. A cluster is simply a list of threads that share data. It then assigns a weight C_{weight} to each cluster; clusters with the highest weights will be scheduled on the nodes with the best connectivity. By default clusters are weighed by the logarithm of the number of remote memory accesses performed by their threads ($C_{weight} = log(C_{rbw})$). The logarithm deemphasizes small differences in C_{rbw} between the clusters, while preserving large differences. This makes it much easier for the algorithm to pick out the clusters with a relatively high C_{rbw} and place them on well-connected nodes.

Step 2: *AsymSched* computes possible *placements* for all the clusters. A placement is an array mapping clusters to nodes. It works at the node granularity, so the number of possible placements is equal to the number of node permutations (i.e., migrating all threads of node X to node Y and vice versa). As this number can be very large, it is important that *AsymSched* not test all possible placements. Section 4.3 details how *AsymSched* avoids testing all possible placements. For each placement P, *AsymSched* computes the maximum bandwidth $C_{bw}(P)$ that each cluster C would receive if it were put in this placement. Each placement is assigned a performance metric, P_{wbw}, the *weighted bandwidth* of P, defined as $P_{wbw} = \sum_{C \in clusters} C_{bw}(P) * C_{weight}$. The higher P_{wbw}, the higher the bandwidth available to clusters that perform a lot of remote communications. The definition of C_{weight} implies that our algorithm aims to optimize the overall communication bandwidth across all applications. The

[3]PEBS, Precise Event-Based Sampling, is a similar feature on Intel systems.

algorithm can be easily changed to optimize a different metric, e.g., one that takes into account application priorities, by changing the definition of C_{weight}.

Step 3: *AsymSched* filters placements to keep only those that have a *weighted bandwidth* value at least equal to 90% of the maximum weighted bandwidth. Among these remaining placements *AsymSched* chooses those that will minimize the number of page migrations.

Step 4: For each application, *AsymSched* estimates the overhead of memory migration assuming the cost of 0.3s per GB, which was derived on our system using simple experiments. If the overhead is deemed too high, the new placement will not be applied. Another goal here is to avoid migrating the applications back and forth because of recurring changes in communication patterns and accumulating a high overhead. To that end, *AsymSched* keeps track of the total time already spent doing memory migration for the application: A_{tm}. If that time plus the estimated cost of additional migration ($\sum_{n \in migrated\_nodes} A_{mm}[n] * 0.3$) exceeds 5% of the running time of the application (A_{tt}), then *AsymSched* does not apply the new thread placement. We chose 5% as a reasonable maximum overhead value. In practice, the highest overhead we observed was around 3%.

Migration. Step 1: *AsymSched* migrates threads using system calls that are available in the Linux kernel.

Step 2: *AsymSched* relies on *dynamic migration* to migrate the subset of pages that the application uses. If, after two seconds, the application still performs more than 90% of its memory accesses on the nodes where it was previously running ($A_{oldA} > 90\%$), then *AsymSched* concludes that *dynamic migration* was not able to migrate the working set of the application and performs a *full memory migration*.

The Measurement, Decision and Migration phases described above are performed continuosly to account for phase changes in applications and other dynamics.

Algorithm 1 *AsymSched* algorithm

1: **if** Threads of nodes N_1 and N_2 access a common memory controller and threads of N_1 and N_2 have the same pid **then**
2: Put all threads running on N_1 and N_2 in a cluster C and Increase C_{rbw}
3: **end if**
4: Compute relevant cluster placements
5: $Max_{wbw} = 0$
6: **for all** P $\in$ computed placements **do**
7: $P_{wbw} = \sum_{C \in clusters} C_{bw}(P) * C_{weight}$
8: $Max_{wbw} = max(Max_{wbw}, P_{wbw})$
9: **end for**
10: **for all** P $\in$ computed placements **do**
11: Skip if $P_{wbw} < 90\% * Max_{wbw}$
12: Compute P_{mm}
13: **end for**
14: Choose the placement with the lowest P_{mm}
15: **for all** A $\in$ migrated applications **do**
16: **if** $A_{tm} + \sum_{n \in migrated\_nodes} A_{mm}[n] * 0.3 > 0.05 * A_{tt}$ **then**
17: Do not change thread placement
18: **end if**
19: **end for**
20: Migrate threads
21: Use dynamic memory migration
22: After 2 seconds:
23: **for all** A $\in$ migrated applications **do**
24: **if** $A_{oldA} > 90\%$ **then**
25: Fully migrate memory of A
26: **end if**
27: **end for**

4.3 Optimizations and tricks

We integrated several optimizations within *AsymSched* to ensure that it runs accurately and with low overhead.

Fast memory migration. When *AsymSched* performs full memory migration, all the pages located on one node are migrated to another node. The applications we tested have large working sets (up to 15GB per node), and migrating pages is costly. We measured that migrating 10GB of data using the standard `migrate_pages` system call takes 51 seconds on average, making migration of large applications impractical.

Therefore, we designed a new system call for memory migration. This system call performs memory migration without locks in most cases, and exploits the parallelism available on multicore machines. Using our system call, migrating memory between two nodes is on average 17× faster than using the default Linux system call and is only limited by the bandwidth available on interconnect links. Unlike the Linux system call, our system call can migrate memory from multiple nodes simultaneously. So if we are migrating the memory simultaneously between two pairs of nodes that do not use the same interconnect path, our system call will run about 34 times faster.

Fast migration works as follows. (i) First, we "freeze" the application by sending SIGSTOP to all its threads. Freezing the application is done to ensure that the application does not allocate or free pages during migration. This allows removing many locks taken by the Linux memory migration mechanism, and since up to 80% of migration time can be wasted waiting on locks, the resulting performance improvements are significant. (ii) Second, we parse the memory map of the application and store all pages in an array. We then launch worker

threads on the node(s) on which the application is scheduled. Worker threads process pages stored in the array in chunks of 30 thousand. The old page is unmapped, data is copied to a new page, the new page is remapped, and the old page is freed. Shared pages or pages that are currently swapped are ignored.

Avoiding evaluation of all possible placements. The number of all possible thread placements on a machine can be very large. We use two techniques to avoid computing all thread placements: (i) A lot of thread placement configurations are "obviously" bad. For instance, when a communication-intensive application uses two nodes, we only consider configurations with nodes connected with a 16-bit link. (ii) Several configurations are equivalent (e.g., the bandwidth between nodes 0 and 1 and between nodes 2 and 3 is the same). To avoid estimating the bandwidth of *all* placements, we create a hash for each placement. The hash is computed so that equivalent configurations have the same hash. Using simple dynamic programming techniques, we only perform computations on non-equivalent configurations.

These two techniques allow skipping between 67% and 99% of computations in all tested configurations with clusters of 2, 3 or 5 nodes (e.g., with 4 clusters of 2 nodes, we only evaluate 20 configurations out of 5040).

5 Evaluation

Our goal is to evaluate the impact of asymmetry-aware thread placement *in isolation* from other effects, such as those stemming purely from collocating threads that share data on the same node. Performance benefits of sharing-aware thread clustering are well known [30]. *AsymSched* clusters threads that share data as described in the Section 4; the Linux thread scheduler, however, does not. We experimentally observed that Linux performed worse than clustered configurations. E.g., when graph500 and specjbb are scheduled simultaneously, both run 23% slower on Linux than on an average clustered placement. Since comparing Linux to *AsymSched* would not be meaningful because of that, we instead compare *AsymSched*[4] to the best and the worst static placements of data-sharing thread clusters. We also compare the average performance achieved under all static placements that are unique in terms of connectivity. We obtain all unique static placements with respect to connectivity by examining the topology of the machine. There are 336 placements for single-application scenarios and 560 placements for multi-application scenarios.

Further, we want to isolate the effects of thread placement with *AsymSched* from the effects of dynamic memory migration. To that end, we compare *AsymSched* to the subset of our algorithm that performs the dynamic placement of memory only, turning off the parts performing thread placement.

[4]When running *AsymSched*, thread clusters are initially placed on a randomly chosen set of nodes.

5.1 Experimental platform

We evaluate *AsymSched* on machine A. It is equipped with four AMD Opteron 6272 processors, each with two NUMA nodes and 8 cores per node (64 cores in total). The machine has 256GB of RAM and uses HyperTransport 3.0. It runs Linux 3.9.

We used several benchmark suites: the NAS Parallel Benchmarks suite [6] which is composed of numeric kernels, MapReduce benchmarks from Metis [25], parallel applications from Parsec [26], Graph500 [1], a graph processing application with a problem size of 21, FaceRec from the ALPBench benchmark suite [11], and SPECjbb [2] running on OpenJDK7. From the NAS and Parsec benchmark suites we picked the benchmarks that run for at least 15 seconds, and that can be executed with arbitrary numbers of threads. The memory usage of the benchmarks ranges from 518MB for EP from the NAS suite to 34,291MB for IS from NAS. Except for SPECjbb, we use the execution time of applications as performance indicator. SPECjbb runs during a fixed amount of time; we use the throughput (measured in SPECjbb bops) as performance indicator.

5.2 Single application workloads

The results are presented in Figure 4. *AsymSched* always performs close to the best static thread placement. In a few cases where it does not, the difference is not statistically significant. For applications that produce the highest degree of contention on the interconnect links (streamcluster, pca, and facerec), *AsymSched* achieves much better performance than the best thread placement, because the dynamic memory migration component balances memory accesses across nodes, thus reducing contention on interconnect links and memory controllers.

We also observe that dynamic memory migration without the migration of threads is not sufficient to achieve the best performance. More precisely, dynamic memory migration alone often achieves performance close to average. Moreover, it produces a high standard deviation for many benchmarks: the minimum and maximum performance often being the same as that of the best and worst static thread placement. For instance, on SPECjbb, the difference between the minimum and maximum performance with dynamic memory migration alone is 91%.

In contrast, *AsymSched* produces a very low standard deviation for most benchmarks. Two exceptions are is.D

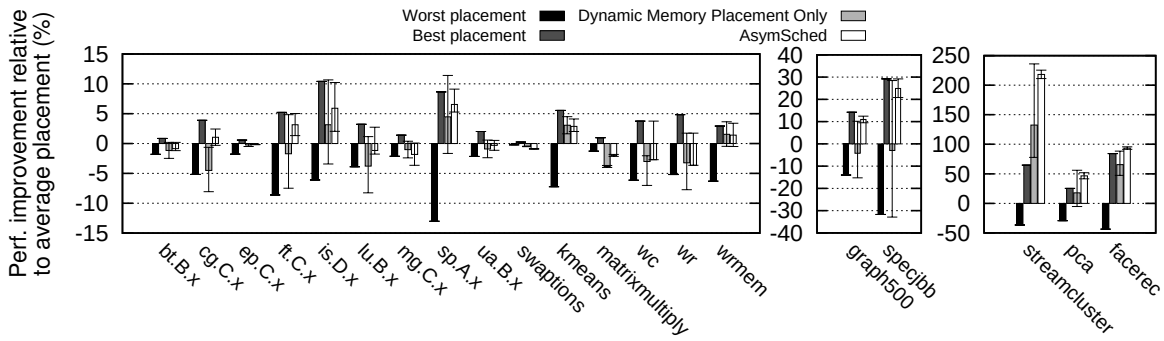


Figure 4: Performance difference between the best and worst static thread placement, dynamic memory placement, *AsymSched* and the average thread placement on machine A. Applications run with 24 threads on 3 nodes.

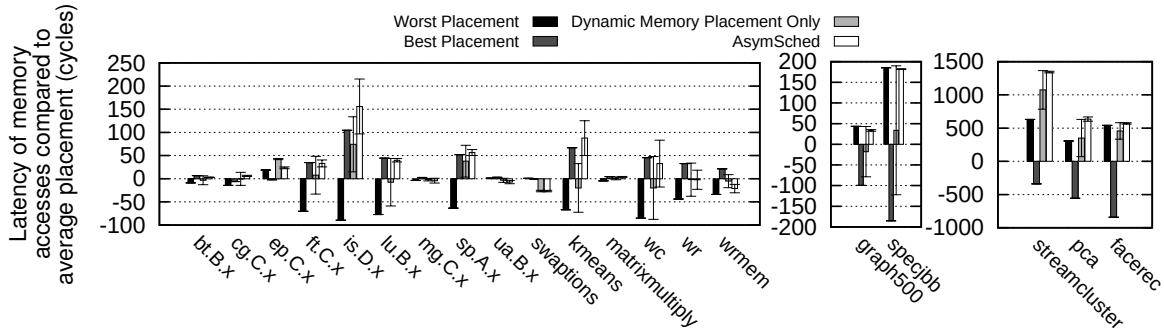


Figure 5: Memory latency under the best and worst static thread placement, dynamic memory placement, *AsymSched* and the average thread placement on machine A. Applications run with 24 threads on 3 nodes.

and SPECjbb. This is because in both cases, *AsymSched* migrates a large amount of memory. Both applications become memory intensive after an initialization phase, and *AsymSched* starts migrating memory only after the entire working set has been allocated. For instance, in the case of is.D, *AsymSched* migrates between 0GB and 20GB, depending on the initial placement of threads.

Figure 5 shows the latency of memory accesses compared to the average. For most applications, the dynamics of latency closely matches that of the performance. A few exceptions are is.D, lu.B and kmeans. For is.D, the latency is drastically improved by *AsymSched* but the impact on performance is not visible because of the time lost performing memory migrations. Lu.B is extremely memory intensive during its first seconds of execution, but performs very few memory accesses thereafter; *AsymSched* improves this initial phase but has no impact on the rest of the running time. Kmeans is very bursty; placing its threads has a huge impact on the latency of memory accesses performed during bursts of memory accesses but not on the rest of the execution.

5.3 Multi application workloads

We evaluate several multi-application workloads using the applications studied in section 5.2. We chose four applications that benefit to various degrees from *AsymSched*: streamcluster (benefits to a high degree), SPECjbb (benefits to a moderate degree), graph500 (benefits to a small degree), and matrixmultiply (does not benefit). Some of these applications have different phases during their execution; for instance, streamcluster processes its input set in five distinct rounds, and SPECjbb spends significant amount of time initializing data before emulating a three-tier client/server system.

Figure 6 presents the performance on multi application workloads. We chose two different clustering configurations: (i) Three applications executing on three, three and two nodes, respectively; (ii) Two applications executing on five and three nodes respectively.

In all workloads, *AsymSched* achieves performance that is close or better than the best static thread placement on Linux. Furthermore, it produces a very low standard deviation. In constrast, dynamic memory migration alone exhibits high standard deviation and, like with single application workloads, is unable to improve performance for Graph500 and SPECjbb.

AsymSched significantly improves the latency of applications that benefit from thread and memory migration (Figure 7), in particular for streamcluster. This is because *AsymSched* chooses configurations in which the links used by streamcluster are not shared with any other application.

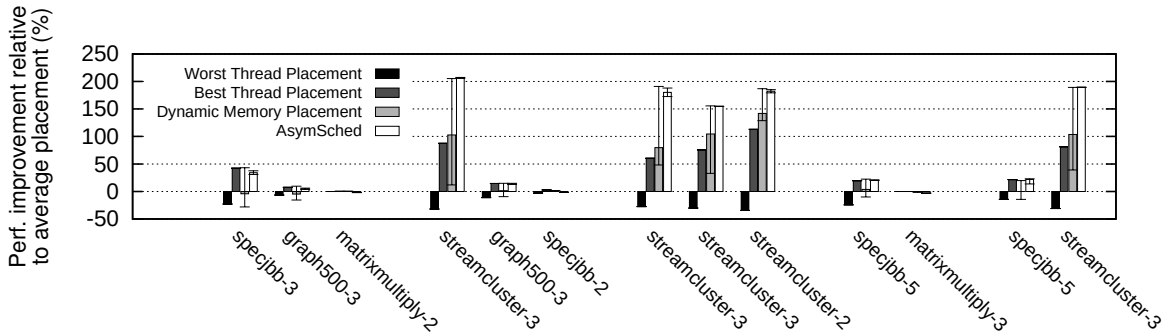


Figure 6: Performance difference between the best thread placement, the worst thread placement, dynamic memory placement, *AsymSched* and the average thread placement of applications on machine A. The numbers appended to the name of applications specify the number of nodes on which the application runs.

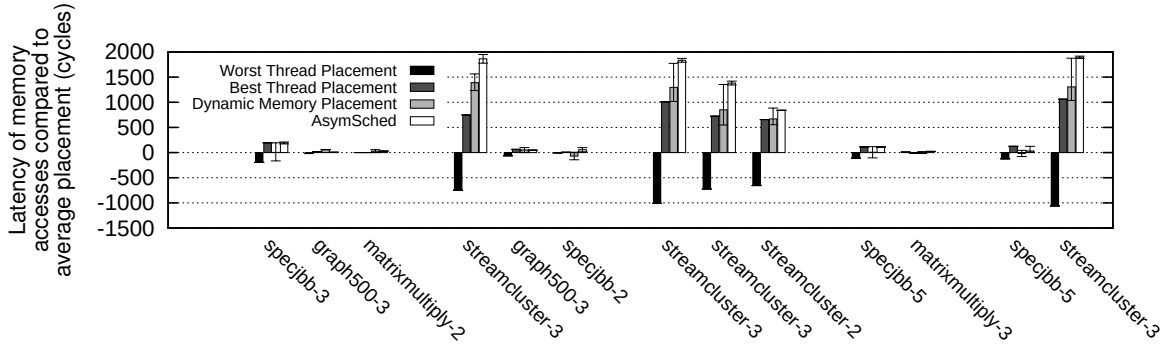


Figure 7: Performance difference between the best thread placement, the worst thread placement, dynamic memory placement, *AsymSched* and the average thread placement of applications on machine A.

	cg.B	ft.C	is.D	sp.A	streamcluster	graph500	specJBB
Migrated memory (GB)	0.17	2.5	20	0.1	0.15	0.3	10
Average time - Linux syscall (ms)	860	12700	101000	490	750	1500	50500
Average time - fast migration (ms)	51	380	3050	30	45	90	1500

Table 5: Average amount of migrated memory for various applications running on 3 nodes and required time to perform the migration using the standard Linux system call and using fast memory migration.

5.4 Overhead

The main overhead of *AsymSched* is due to memory migration. This explains why we implemented a custom system call (see Section 4.3). Table 5 compares the migration time when running the standard Linux system call and when running our custom system call. For instance, for is.D, migration takes 101 seconds using the Linux system call (50% overhead), but only 3 seconds using our custom system call (1.5% overhead). To keep the overhead low, *AsymSched* performs migrations only if the predicted overhead is below 5%. In practice, the maximum migration overhead we observed was 3%.

The cost of collecting metrics and computing cluster placement is below 0.5% on all studied applications. Moreover, *AsymSched* requires less than 2MB of RAM.

The overhead of thread migration is negligible and we did not observe any noticeable effect of thread migrations on cache misses.

Finally, when dynamic memory placement is used, IBS sampling incurs a light overhead (within 2% in our experiments) and statistics on memory accesses are stored in about 20MB of RAM.

5.5 Discussion - Applicability on future NUMA machines

We believe that the findings and the solution presented in this paper are likely to be applicable on future NUMA systems. First, we believe that the clustering and placement techniques used in *AsymSched* can scale on machines with a much larger number of nodes. With very simple heuristics we were able to avoid computing up to 99% of the possible thread placements. Such optimizations will still likely be possible on future machines, as machines are usually made of multiple identical cores/sockets (e.g., our 64-core machine has 4 identical sockets). On machines that offer a wider diversity

of thread placements, a possibility is to use statistical approaches, such as that of Radojkovic et al. [27] to find good thread placements with a bounded overhead.

Furthermore, *AsymSched* can easily be adapted to different optimization goals. On current NUMA machines, maximizing the bandwidth between threads was the key to achieving good performance, but our solution could be easily adapted to take other metrics into account.

6 Related Work

NUMA optimizations and contention management: Optimizing thread and memory placement on NUMA systems has been extensively studied [8, 9, 20, 33, 19, 12, 9, 7, 31, 5, 23, 22, 10]. However, as shown in [19, 12, 5, 23], contention on interconnect links and memory controllers remains a major source of inefficiencies on modern NUMA machines. Our work complements these previous studies by minimizing contention on interconnect links on asymmetrically connected NUMA systems. We adopt a dynamic memory management algorithm presented in [12] for memory placement, but the key contribution of our work is the algorithm that efficiently computes the placement of threads on nodes to maximize the bandwidth between communicating threads.

Several extensions to Linux improve data-access locality on NUMA systems, but do not improve the bandwidth for communicating threads and do not address interconnect asymmetry. For example, Sched/NUMA [32] adds the notion of a "home node": when scheduling threads, Linux will try to collocate threads and data on the "home node" of the corresponding process. While this may improve communication bandwidth for applications with the number of threads not exceeding the number of cores in a node, it does not address applications spanning several nodes. Another extension, called AutoNUMA [4], implements locality optimizations by migrating pages on the nodes from which they are accessed. *AsymSched* uses a similar dynamic algorithm to migrate memory, but unlike AutoNUMA it also places threads so as to optimize communication bandwidth.

Several studies addressed contention for the memory hierarchy of UMA systems [16, 24, 34] by segregating competing threads on different nodes. None of these systems, however, addressed contention on the interconnect.

Scheduling on asymmetric architectures: Several thread schedulers catered to the asymmetry of CPUs [29, 15, 21, 17]. They optimize thread placement on processors with asymmetric characteristics (e.g., different frequencies, or different hardware features). The techniques used to address processor asymmetry are fundamentally different than those needed to address interconnect asymmetry, so there is no overlap with our study.

Thread clustering: Pusukuri et al. [18] cluster threads based on lock contention and memory access latencies. Kamali [14] and Tam [30] proposed algorithms that cluster threads that share data on the same shared cache. *AsymSched* uses a similar high-level idea: it samples hardware counters to detect communicating threads and place them onto a well-connected nodes. However, the problem addressed in *AsymSched* (asymmetric interconnect) and the specific algorithm proposed is quite different from those in the aforementioned studies.

Radojkovic et al. [28] present a scheduler that takes into account resource sharing inside a processor. They model the benefits and drawbacks of data and instruction cache sharing between threads, and they schedule threads on the the set of cores that will maximize performance. Their solution explores all possible thread placements. Their follow-up work [27] refines the solution to use a statistical approach to find the optimal placement. Our solution could benefit from a similar technique on machines with a much larger number of dissimilar nodes.

Network optimizations: Network traffic optimization is a well studied problem. Machines are often interconnected with asymmetric Ethernet links; optimizing the bandwidth on asymmetric NUMA systems shares a lot of similarities with optimizations problems found in these systems. For instance, Volley [3] proposed an algorithm to place data used by Cloud services. As *AsymSched*, this algorithm takes into account the available bandwidth between nodes (that are geographically distributed computers in their case) in order to optimize performance.

Hermenier et al. [13] present a consolidation manager for distributed systems. The goal of their system is to minimize energy consumption in a cluster. To this end, they place virtual machines on the smallest possible number of physical machines while meeting certain performance constraints. They model the problem as the multiple knapsack problem and use a constraint-satisfaction solver to find good placements. *AsymSched* could use a similar technique, but we found that on existing systems a much simpler solution was sufficient.

7 Conclusion

We showed that the asymmetry of the interconnect in modern NUMA systems drastically impacts performance. We found that the performance is more affected by the bandwidth between nodes than by the distance between them. We developed *AsymSched*, a new thread and memory placement algorithm that maximizes the bandwidth for communicating threads.

As the number of nodes in NUMA systems increases, the interconnect is less likely to remain symmetric. *AsymSched* design principles will, therefore, be of growing importance in the future.

References

[1] Graph500 reference implementation. `http://www.graph500.org/referencecode`.

[2] specjbb2005 - industry-standard server-side java benchmark (j2se 5.0). `http://www.spec.org/jbb2005/`.

[3] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[4] AutonNUMA: the other approach to NUMA scheduling. LWN.net, March 2012. `http://lwn.net/Articles/488709/`.

[5] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT*, 2010.

[6] 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. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165, Nov 1991.

[7] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[8] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Management. In *SOSP*, 1989.

[9] Timothy Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *USENIX SEDMS*, 1993.

[10] J. Mark Bull and Chris Johnson. Data distribution, migration and replication on a ccnuma architecture. In *In Proceedings of the Fourth European Workshop on OpenMP*, 2002.

[11] CSU Face Identification Evaluation System. `http://www.cs.colostate.edu/evalfacerec/index10.php`.

[12] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 381–394. ACM, 2013.

[13] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[14] Ali Kamali. Sharing aware scheduling on multicore systems. In *MSc Thesis, Simon Fraser Univ.*, 2010.

[15] Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. Aash: An asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 85–96, New York, NY, USA, 2010. ACM.

[16] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):pp. 54–66, 2008.

[17] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[18] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, January 2013.

[19] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX ATC*, 2012.

[20] Richard P. LaRowe, Jr., Carla Schlatter Ellis, and Mark A. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE TPDS*, 3:686–701, 1991.

[21] Tong Li, Dan Baumberger, David A Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Supercomputing, 2007. SC '07. Proceedings of*

the 2007 ACM/IEEE Conference on, pages 1–11, Nov 2007.

[22] Zoltan Majo and Thomas R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *ISMM*, 2011.

[23] Zoltan Majo and Thomas R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *SYSTOR*, 2011.

[24] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *EuroSys*, 2010.

[25] Metis MapReduce Library. `http://pdos.csail.mit.edu/metis/`.

[26] PARSEC Benchmark Suite. `http://parsec.cs.princeton.edu/`.

[27] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Optimal task assignment in multithreaded processors: A statistical approach. *SIGARCH Comput. Archit. News*, 40(1):235–248, March 2012.

[28] Petar Radojković, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Thread to strand binding of parallel network applications in massive multi-threaded systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 191–202, New York, NY, USA, 2010. ACM.

[29] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 139–152, New York, NY, USA, 2010. ACM.

[30] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys*, 2007.

[31] Lingjia Tang, J. Mars, Xiao Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing google's warehouse scale computers: The numa experience. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 188–197, Feb 2013.

[32] Toward better NUMA scheduling. Linux Weekly News, March 2012. `http://lwn.net/Articles/486858/`.

[33] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS*, 1996.

[34] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.