Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems

Mohammad Dashti
Simon Fraser University
mdashti@sfu.ca

Alexandra Fedorova
Simon Fraser University
fedorova@sfu.ca

Justin Funston
Simon Fraser University
jfunston@sfu.ca

Fabien Gaud
Simon Fraser University
fgaud@sfu.ca

Renaud Lachaize
UJF
renaud.lachaize@imag.fr

Baptiste Lepers
CNRS
baptiste.lepers@imag.fr

Vivien Quéma
Grenoble INP
vivien.quema@imag.fr

Mark Roth
Simon Fraser University
mroth@sfu.ca

Abstract

NUMA systems are characterized by Non-Uniform Memory Access times, where accessing data in a remote node takes longer than a local access. NUMA hardware has been built since the late 80's, and the operating systems designed for it were optimized for access locality. They co-located memory pages with the threads that accessed them, so as to avoid the cost of remote accesses. Contrary to older systems, modern NUMA hardware has much smaller remote wire delays, and so remote access costs *per se* are not the main concern for performance, as we discovered in this work. Instead, *congestion on memory controllers and interconnects*, caused by memory traffic from data-intensive applications, hurts performance a lot more. Because of that, memory placement algorithms must be redesigned to target traffic congestion. This requires an arsenal of techniques that go beyond optimizing locality. In this paper we describe *Carrefour*, an algorithm that addresses this goal. We implemented *Carrefour* in Linux and obtained performance improvements of up to 3.6× relative to the default kernel, as well as significant improvements compared to NUMA-aware patchsets available for Linux. *Carrefour* never hurts performance by more than 4% when memory placement cannot be improved. We present the design of *Carrefour*, the challenges of implementing it on modern hardware, and draw insights about hardware support that would help optimize system software on future NUMA systems.

Categories and Subject Descriptors D.4.1 [*OPERATING SYSTEMS*]: Process Management—scheduling

Keywords NUMA, operating systems, multicore, scheduling

1. Introduction

Modern servers are built from several *processor nodes*, each containing a multicore CPU and a local DRAM serviced by one or more memory controllers (see Figure 1). The nodes are connected into a single cache-coherent system by a high-speed *interconnect*. Physical address space is globally shared, so all cores can transparently access memory in all nodes. Accesses to a local node go through a local memory controller; accesses to remote nodes must traverse the interconnect and access a remote controller. Remote accesses typically take longer than local ones, giving these systems a property of *N*on-*U*niform *M*emory *A*ccess time (NUMA).

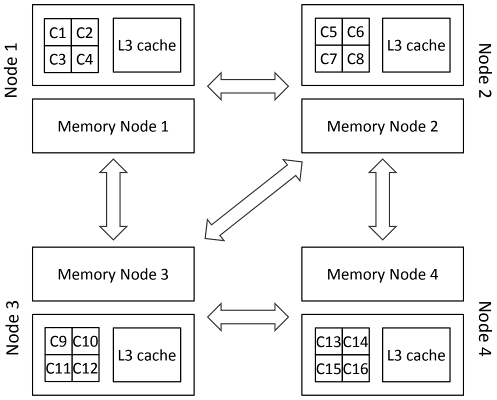


Figure 1. A modern NUMA system, with four nodes and four cores per node. At the time of the writing, NUMA systems are built with up to 8 nodes and 10 cores per node.

It is well understood that optimal performance on NUMA systems can be achieved only if we place threads and their memory in consideration of the system's physical layout. For instance, previous work on NUMA-aware memory placement focused on maximizing locality of accesses, that is, placing memory pages such that data accesses are satisfied from a local node whenever possi-

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03. . . $15.00

ble. That was done to avoid very high costs of remote memory accesses. Contrary to insights from previous work, we discover that on modern NUMA systems remote wire delays, that is, delays resulting from traversing a greater physical distance to reach a remote node, are *not* the most important source of performance overhead. On the other hand, *congestion on interconnect links and in memory controllers*, which results from high volume of data flowing across the system, can dramatically hurt performance. This motivates the design of new NUMA-aware memory placement policies.

To make these statements concrete, consider the following facts. On NUMA systems circa 1990s, the time to access data from a remote node took 4-10 times longer than from a local node [31]. On NUMA systems that are built today, remote wire delays add at most 30% to the cost of a memory access [7]. For most programs, this latency differential alone would not have a substantial impact on performance. However, fast modern CPUs are able to generate memory requests at very high rates. Massive data traffic creates congestion in memory controller queues and on interconnects. When this happens, memory access latencies can become as large as 1000 cycles, from a normal latency of only around 200. Such a dramatic increase in latencies can slow down data-intensive applications by more than a factor of three. Fortunately, high latencies can be avoided or substantially reduced if we carefully place memory pages on nodes so as to avoid traffic congestion.

In response to the changes in hardware bottlenecks, we approach the problem of thread and memory placement on NUMA systems from an entirely new perspective. We look at it as the problem of *traffic management*. Our algorithm, called *Carrefour*[1], places threads and memory so as to avoid traffic hotspots and prevent congestion in memory controllers and on interconnect links. This is akin to traffic management in the context of city planning: popular residential and business hubs must be placed so as to avoid congestion on the roads leading to these destinations.

The mechanisms used in our algorithm: e.g., migration and replication of memory pages, are well understood, but the algorithm itself is new. Our algorithm makes decisions based on global observations of traffic congestion. Previous algorithms optimized for locality, and relied on local information, e.g., access pattern of individual pages. We found that in order to effectively manage congestion on modern systems we need an arsenal of techniques that go beyond optimizing locality. While locality plays a role in managing congestion (when we reduce remote accesses, we reduce interconnect traffic), alone it is not sufficient to achieve the best performance. The challenge in designing *Carrefour* was to understand how to combine different mechanisms in an effective solution for modern hardware.

Implementing an effective NUMA-aware algorithm on modern systems presents several challenges. Modern systems do not have the same performance monitoring hardware that was present (or assumed) on earlier systems. Existing instruction sampling hardware cannot gather the profiling data needed for the algorithm with the desired accuracy and speed. We had to navigate around this problem in our design. Furthermore, the memory latencies that we are optimizing are lower than on older systems, so we can tolerate less overhead in the algorithm.

We implemented *Carrefour* in Linux and evaluated it with several data-centric applications: k-means clustering, face recognition, map/reduce, and others. *Carrefour* improves performance of these applications, with the largest gain of 3.6× speedup. When memory placement cannot be improved *Carrefour* never hurts performance by more than 4%. Existing NUMA-aware patches for the Linux kernel perform less reliably and in general fall short of improvements achieved with *Carrefour*.

2. Traffic congestion on modern NUMA systems

In this section, we demonstrate that the effects of traffic congestion are more substantial than those of wire delays, and motivate why memory placement algorithms must be redesigned. To that end, we report data from two sets of experiments. In the first set, our goal is to measure the effects of wire delays only. We run applications in two configurations: *local-memory* and *remote-memory*. Under *local-memory*, the thread and its data are co-located on the same NUMA node; under *remote-memory*, the thread runs on a different node than its data. To ensure that wire delay is the dominant performance factor, we had to avoid congestion on memory controllers and interconnects, so we run one application at a time and use only one thread in each application. We do not include applications with CPU utilization less than 30%, because memory performance is not their main bottleneck. The experiments are run on a system described in Section 4 (Machine A – illustrated in Figure 1). We use applications from the NAS, PARSEC and map/reduce Metis suites, also described in Section 4.

Figure 2(a) shows relative completion time under *remote-memory* vs. *local-memory* configuration. The performance degrades by at most 20% under *remote-memory*, which is consistent with at most 30% difference in local-vs-remote memory latencies measured in microbenchmarks [7].

In the second set of experiments, we want to observe traffic congestion, so we run each application with as many threads as there are cores. We demonstrate how performance varies under two memory placement policies on Linux, as they induce different degrees of traffic congestion. The first policy is *First-touch* (F) - the default policy where the memory pages are placed on the node where they are first accessed. The second policy is *Interleaving* (I) - where memory pages are spread evenly across all nodes. Although these are not the only possible and not necessarily the best policies, comparing them illustrates the salient effects of traffic congestion.

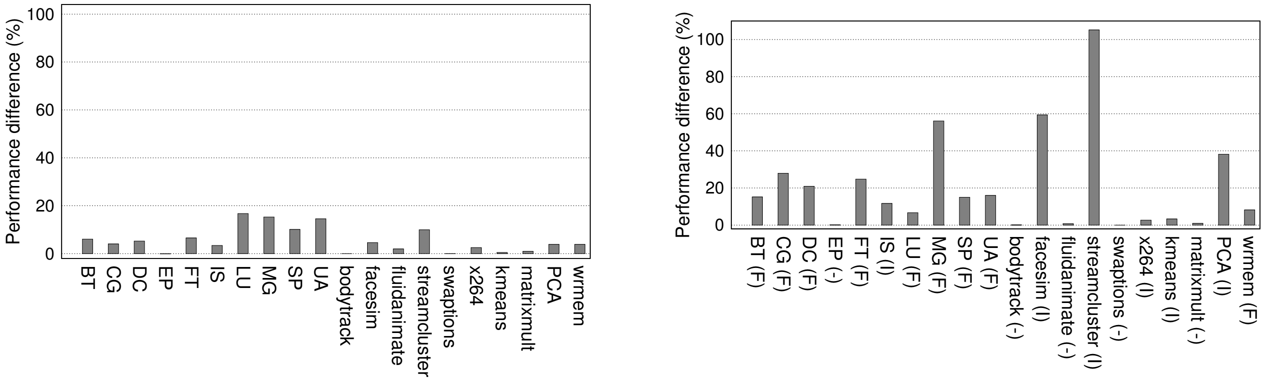
Figure 2(b) shows the absolute difference in completion time achieved under *first-touch* and *interleaving*. The policy that performed the best is indicated in parenthesis next to the application name; a ”-” is shown when the application performs equally well with either policy. We observe that the differences are often much larger than what we can expect from wire delays alone. For Streamcluster, a k-means clustering application from PARSEC, the performance varies by a factor of two depending on the memory placement policy!

To illustrate that these differences are due to traffic congestion, we show in Table 1 some supporting data for the two applications, *Streamcluster* and *PCA* (a map/reduce application)[2]. ***Local access ratio*** is the percent of all memory accesses sourced from a local node; ***Memory latency*** is the average number of cycles to satisfy a memory request from any node; ***Memory controller imbalance*** is the standard deviation of the load across all memory controllers, expressed as percent of the mean. Load is measured as the number of requests per time unit; ***Average interconnect (IC) usage*** shows the utilized interconnect bandwidth as percent of total, averaged across all links, the ***imbalance*** shows the standard deviation of utilization across the links as percent of mean utilization; ***L3MPKI*** is the number of last-level (L3) cache misses per thousand instructions; ***IPC*** is the number of instructions per cycle.

The data in Table 1 leads to several curious observations. First, we see that locality of memory accesses either does not change regardless of the memory management policy, or *decreases* under the better performing policy. For *Streamcluster*, most of the memory pages happen to be placed on a single node under *first-touch* (be-

[1] *Carrefour*– (French) intersection, crossroads.

[2] We are unable to present the same data for all applications due to space constraints, but the conclusions reached from their measurements are qualitatively similar.



(a) Perf. difference for single-thread versions of applications between local and remote memory configurations.

(b) Absolute perf. difference for multi-thread versions of applications between First-touch (F) and interleaving (I).

Figure 2. Performance difference of applications depending on the thread and memory configuration.

	Streamcluster		***PCA***	
	Best (I)	Worst (F)	Best (I)	Worst (F)
Local access ratio	25%	25%	25%	33%
Memory latency	476	**1197**	465	**660**
Mem-ctrl. imbalance	8%	**170%**	5%	**130%**
IC: imbalance, (avg)	22% (59%)	**85%** (33%)	20% (48%)	**68%** (31%)
L3MPKI	16.85	16.89	7.35	7.4
IPC	0.29	**0.15**	0.52	**0.36**

Table 1. Traffic congestion effects

cause a single thread initializes them at the beginning of the program). Under *interleaving* the pages are spread across all nodes, but since the threads access data from all four nodes, the overall access ratio is about the same in both configurations. For *PCA*, interleaving decreases the local access ratio and yet *increases* performance. So the first surprising conclusion is that ***better locality does not necessarily improve performance***!

And yet, the IPC substantially improves (2× for *Streamcluster* and 41% for *PCA*), while the L3 miss rate, as well as L1 and L2 miss rates, remain unchanged. The explanation emerges if we look at the memory latency. Under interleaving, the memory latency reduces by a factor of 2.48 for *Streamcluster* and 1.39 for *PCA*. This effect is entirely responsible for performance improvement under the better policy. The question is, *what is responsible for memory latency improvements*? It turns out that interleaving *dramatically reduces memory controller and interconnect congestion* by alleviating the load imbalance and mitigating traffic hotspots. Rows 5, 6 in Table 1 show significant reductions in imbalance under interleaving, and Figure 3 illustrates these effects visually for *Streamcluster*. So even without improving locality (we even *reduce* it for *PCA*), we are able to substantially improve performance. And yet, existing NUMA-aware algorithms disregarded traffic congestion, optimizing for locality only. Our work addresses this shortcoming.

Although the two selected applications performed significantly better under interleaving, this does not mean that interleaving is the only desired policy on modern NUMA hardware. In fact, as Figure 2(b) shows, many NAS applications fared a lot worse with interleaving. In the process of designing the algorithm we learned that a range of techniques – interleaving, page replication and co-location – must be judiciously applied to different parts of the address space depending on global traffic conditions and page access patterns. So the challenge in designing a good algorithm is understanding when to apply each technique, while navigating around the challenges of obtaining accurate performance data and limiting the overhead.

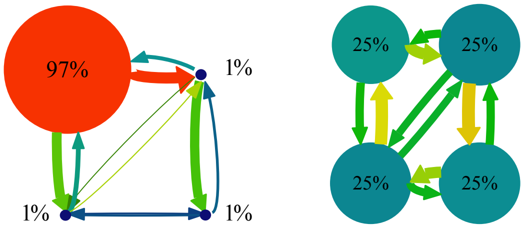


Figure 3. Traffic imbalance under first-touch (left) and interleaving (right) for *Streamcluster*. Nodes and links bearing the majority of the traffic are shown proportionately larger in size and in brighter colours. The percentage values show the fraction of memory requests destined for each node. The figure is drawn to scale.

3. Design and Implementation

We begin by describing the mechanisms composing the algorithm: *page co-location*, *interleaving*, *replication* and *thread clustering*. Then we explain how they fit together.

3.1 The mechanisms

Page co-location is when we re-locate the physical page to the same node as the thread that accesses it. Co-location works well for pages that are accessed by a single thread or by threads co-located on the same node.

Page interleaving is about evenly distributing physical pages across nodes. Interleaving is useful when we have imbalance on memory controllers and interconnect links, and when pages are accessed by many threads. Operating systems usually provide an interleaving allocation policy, but only give an option to enable or disable it globally for the entire application. We found that interleaving works best when judiciously applied to parts of the address space that will benefit from it.

Page replication is about placing a copy of a page on several memory nodes. Replication distributes the pressure across memory controllers, alleviating traffic hotspots. An added bonus is eliminat-

Global statistics	
MC-IMB	Memory controller imbalance (as defined in Section 2)
LAR	Local access ratio (as defined in Section 2)
MAPTU	Memory (DRAM) accesses per time unit (microsecond)
Per-application statistics	
MRR	Memory read ratio. Fraction of DRAM accesses that are reads
CPU%	Percent CPU utilization
Per-page statistics	
Number of accesses	The number of sampled data loads that fell in that page
Access type	Read-only or read-write

Table 2. Statistics collected for the algorithm.

ing remote accesses on replicated pages. When done right, replication can bring very large performance improvements. Unfortunately, replication also has costs. Since we keep multiple copies of the same page, we must synchronize their contents, which is like running a cache coherency protocol in software. The costs can be very significant if there is a lot of fine-grained read/write sharing. Another potential source of overhead is the synchronization of page tables. Since modern hardware walks page tables automatically, page tables themselves must be replicated across nodes and kept in sync. Finally, replication increases the memory footprint. We should avoid it for workloads with large memory footprints for fear of increasing the rate of hard page faults.

Thread clustering is about co-locating threads that share data on the same node, to the extent possible[3]. Examination of prior work on the subject revealed two thread clustering algorithms most relevant for our project [15, 30]. Tam's algorithm [30] always co-located on the same node threads that shared data. Kamali's algorithm [15] balanced between increased contention and the benefits of co-operative sharing. It only co-located data-sharing threads, provided that these threads would not aggressively compete for the node's shared resources. We confirmed the importance of balancing the two goals in our experiments, so our algorithm assumes the Kamali's variant of thread clustering. Since thread clustering is well documented and understood, we did not evaluate it in our implementation.

3.2 The Algorithm

Our memory management algorithm has three components: *measurement*, *global decisions* and *page-local decisions*. The measurement component continuously gathers various metrics (Table 2) that later drive page placement decisions. Global and per-application metrics are collected using hardware counters with very low overhead. Per-page statistics are collected via instruction-based sampling (IBS) [12], which can introduce significant overheads at high sampling rates. Section 3.3 describes how we keep the overheads at bay. Global decisions are based on system-wide traffic congestion and workload properties which determine what mechanisms to use. Page-local decisions examine access patterns of individual pages to decide their fate.

3.2.1 Global Decisions

The global decision-making process is outlined in Figure 4.

Step 1: we decide whether to enable *Carrefour*. We only want to run *Carrefour* for applications that generate substantial memory traffic. Other applications would not be affected by memory placement policies, so there is no reason to subject them to sampling overhead. This decision is driven by the application's memory access rate (MAPTU – see Table 2). *Carrefour* is enabled for applications with the MAPTU above a certain threshold. The MAPTU threshold is to be determined experimentally and the right setting may vary from system to system. We found the threshold of 50 MAPTU worked well on all hardware we evaluated, and the performance was not very sensitive to its choice. To determine the right MAPTU threshold on a system very different from ours, we recommend running a benchmark suite under different NUMA policies, noting which applications are affected and using the lowest observed MAPTU from those experiments.

Once we decided whether there is sufficient memory traffic to justify running *Carrefour*, we need to decide which of the available mechanisms, replication, interleaving and co-location, should be enabled for each application given its memory access patterns. The goal here is to choose the most beneficial techniques and avoid any associated overhead. The next three steps take care of this decision.

Step 2: we decide whether it is worthwhile to use replication. Replication risks introducing significant overheads if it forces us to run out of RAM (and causes additional hard page faults) or requires frequent synchronization of pages across nodes (see more discussion in Section 3.3.2). To avoid the first peril, we conservatively enable replication only if there is sufficient free RAM to replicate the entire resident set. That is, the fraction of free RAM must be at least $1 - \frac{1}{NUM\_NODES}$. This is a conservative threshold, because not all pages will be replicated, and not all resident pages will be accessed frequently enough to generate significant page fault overhead if evicted. Evaluating the trade-off between replication benefit and potentially increased page-fault rate was outside the scope of the work. This requires workloads that both benefit from replication and have very large memory-resident sets, which we did not encounter in our experiments.

To avoid the overhead associated with the synchronization of page content across nodes, we do not replicate pages that are frequently written. An application must have the memory read ratio (MRR) of at least 95% in order for its memory pages to be considered for replication[4]. The setting of this parameter can have a very significant effect on performance. While we found that the performance was not sensitive when we varied the parameter in the range of 90-99%, it is always safe to err on the high side.

Step 3: we decide whether to use interleaving. Interleaving improves performance if we have large memory controller imbalance. We enable interleaving if memory controller imbalance is above 35%, but found that the performance was not highly sensitive to this parameter. Applications that benefit from interleaving usually begin with a very large imbalance.

Step 4: we decide whether or not to enable co-location. Co-location will be triggered only for pages that are accessed from a single node, and so it will not exacerbate the imbalance if memory-intensive threads are evenly spread across nodes, which is ensured by thread clustering. Therefore, we enable co-location if the local access rate is slightly less than ideal ($LAR < 80\%$). Performance is not highly sensitive to this parameter; we observed that if this parameter is completely eliminated from the algorithm, the largest performance impact is only a few percent.

Although we expect that optimal settings for the parameters used in the algorithm would vary from one system to another, we found that we did not need to adjust the settings when we moved between the two experimental systems used in our evaluation. Although our systems had the same number of nodes and both used

[3] We never want to sacrifice CPU load balance in favour of clustering.

[4] MRR is approximated as fraction of L1 refills from DRAM in modified state, because there is no a hardware counter that provides this quantity precisely per core, as opposed to per-node. Similarly, due to hardware counter limitations described in Section 3.3.2, it is very difficult to measure the MRR per page. That is why we use the MRR for the entire application.

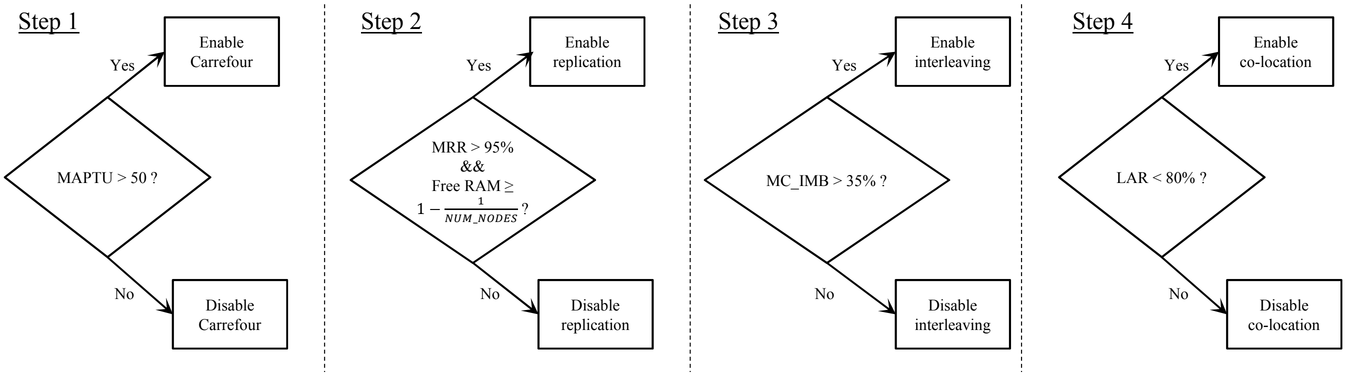


Figure 4. Global decisions in *Carrefour*.

AMD CPUs, they differed in the number of cores per node, the cache-coherency protocol (broadcast vs. directory-based), and one had a higher interconnect throughput than the other. Therefore, it is possible that the algorithm parameters settings are rather stable across all but drastically different systems.

3.2.2 Page-local Decisions

Carrefour makes page-local decisions depending on the mechanisms enabled: e.g., pages are only considered for replication if replication is enabled for that application. The following explanation assumes that all three mechanisms are enabled.

To decide the fate of each page, we need at least two memory-access samples for that page. If the page was accessed from only a single node we migrate it to that node. We do not migrate the thread, because we assume that thread clustering, performed before memory placement, already made good thread placement decisions. If the page is accessed from two or more nodes, it is a candidate for either interleaving or replication. If the accesses are read-only, the page is replicated. Otherwise it is marked for interleaving. To decide where to place a page marked for interleaving, we use two probabilities: $P_{migrate}$ and P_{node}. $P_{migrate}$ determines the likelihood of migrating the page away from the current node. $P_{migrate}$ is the MAPTU of the current node as the fraction of MAPTU on all nodes, so the higher the load on the current node relative to others, the higher the chance that we will migrate a page. P_{node} gives us the probability of migrating a page to a particular node, and it is the complement of $P_{migrate}$ for that node, so *Carrefour* will migrate the page to the least loaded node.

3.3 Implementation

We implemented *Carrefour* in the Linux kernel 3.6.0. *Carrefour* runs measures the selected performance indicators, and with periodicity of one second makes decisions regarding page placement and resets statistic counters. To a large extent, *Carrefour* relies on well-understood mechanisms in the Linux kernel, such as physical page migration. The non-trivial aspects of the implementation were understanding how to accomplish fast and accurate sampling of memory accesses and navigating around the overheads of replication. We describe how we overcame these challenges in the two sections that follow.

3.3.1 Fast and accurate memory access sampling

A crucial goal of the algorithm is to quickly and accurately detect memory pages that cause the most DRAM accesses, and accurately estimate the read/write ratio of those pages. To that end, we used Instruction-Based Sampling (IBS) – hardware-supported sampling of instructions available in AMD processors. Intel processors support similar functionality in the form of PEBS: Precise Event-Based Sampling. IBS can be configured to deliver instruction samples at a desired interval, e.g., after expiration of a certain number of cycles or micro-ops. Each sample contains detailed information about the sampled instruction, such as the address of the accessed data (if the instruction is a load or a store), whether or not it missed in the cache and how long it took to fetch the data. Unfortunately, every delivered sample generates an interrupt, so processing samples at a high rate becomes very costly. Other systems that relied on IBS performed off-line profiling [17, 27], so they could tolerate much higher overhead than what would be acceptable in our online algorithm.

After experimenting with IBS on our systems, we found that for most applications the sampling interval of 130,000 cycles incurs a reasonable overhead of less than 5%. The desired sampling rate can be trivially derived for new systems: it amounts to experimenting with different sampling rates and settling for the one that generates acceptable runtime overhead.

Our initial decision was to filter out all the samples that did not generate a DRAM access. However, we found that the resulting number of samples was extremely low. Even very memory-intensive workloads access DRAM only a few times for every thousand instructions. That, combined with a low IBS sampling frequency, gave us the sampling rate of less than one hundred thousandth of a percent, and made it very difficult to generate a sufficient number of samples. Furthermore, filtering samples that did not access DRAM caused us to miss the accesses generated by the hardware prefetcher. These accesses are not part of any instruction so they will not be tagged by IBS. For prefetch-intensive applications, we obtain a very small number of samples and a very distorted read-write ratio.

To address this problem, we used two solutions. First, is the adaptive sampling rate. When the program begins to run, we sample it at a relatively high rate of 1/65,000 cycles. If after this measurement phase we take fewer than ten actions in the algorithm (an action is any change in page placement) we switch to a much lower rate of 1/260,000 cycles. Otherwise we continue sampling at the high rate.

The second solution was, when filtering IBS samples, to retain not just the data samples that accessed the DRAM, but those that hit in the first-level cache as well. First-level cache loads include accesses to prefetched data, so we avoid prefetcher-related inaccuracy. On the one hand, considering cache accesses can introduce ”noise” in the data, because we could be sampling pages that never access DRAM. On the other hand, *Carrefour* is only activated for memory-intensive applications, and for them there is a higher correlation between the accesses that hit in the cache and those that access DRAM.

With these two solutions combined, we were able to successfully identify the pages that are worth replicating, while this was nearly impossible prior to introducing these solutions. For example, for *Streamcluster* we used to be able to detect only a few percent of the pages that are worth replicating, but with these solutions in place, we were able to identify 100% of them[5].

However, even though performance became much better (we were able to speed up *Streamcluster* by 26% relative to the default kernel), we were still far from the "ideal" manual replication, which sped it up by more than 2.5×. To approach ideal performance, we had to mitigate the overheads of replication, which we describe next.

3.3.2 Replication

Replication has overhead from the following three sources. First, there is the initial set-up cost and slightly more expensive page faults. Modern hardware walks page tables automatically, so a separate copy of a page table must be created for each node. Page faults become slightly more costly, because a new page table entry must be installed on every node. To avoid these costs when we are not likely to benefit from replication, we avoid replication unless the applications has at least a few hundred pages marked for replication[6].

The second source of overhead comes from additional hard page faults if we exceed the physical RAM capacity by replicating pages. As explained earlier, we avoided this overhead by conservatively setting the free memory threshold when enabling replication.

The final and most significant source of overhead stems from the need to synchronize the contents of replicated pages when they are written. This involves a physical page copy and is very costly. Before explaining how we avoid this overhead we provide a brief overview of our implementation of replication.

In Linux, a process address space is represented by a `mm_struct`, which keeps track of valid address space segments and holds a pointer to the page table, which is stored as a hierarchical array. Since modern hardware walks page tables automatically, we cannot modify the structure of the page table to point to several physical locations (one for each node) for a given virtual page. Instead, we must maintain a separate copy of the page table for each node and synchronize the page tables when they are modified, even for virtual pages that are not replicated. Linux dictates that the page table entry (PTE) be locked when it is being modified. We do not make any changes to this locking protocol. The only difference is that we designate one copy of the page table as the *master copy*, and only lock the PTE in the master copy while installing the corresponding PTEs into all other replicas.

When a page is replicated, we create a physical copy on every memory node that runs threads from the corresponding application. We install a different virtual-to-physical translation in each node's page table. We write-protect the replicated page, so when any node writes that page we receive a page protection fault. To handle this fault, we read-protect the page on all nodes except the faulting one, and enable writing on the faulting node. If another node accesses that page, we must copy the new version of the page to that node, enable the page for reading and protect it from writing.

We refer to all the actions needed to keep the pages synchronized as *page collapses*. Collapses are extremely costly, and would occur if we replicate a page that is write-shared. Even with very infrequent writes (e.g., one in 1000 accesses), the collapse overhead could be prohibitively high. With limited capabilities of IBS, we are unable to detect read/write ratio on individual pages with sufficient accuracy. That is why we use the application-wide MRR and disable replication for the entire application if the MRR is low. Furthermore, we monitor collapse statistics of individual pages and disable replication for any page that generated more than five collapses.

With these optimizations, as well as those described in Section 3.3.1, we were able to avoid replication costs for the applications we tested and approached within 10% the performance of manual replication for *Streamcluster*.

4. Evaluation

In this section, we study the performance of *Carrefour* on a set of benchmarks. The main questions that we address are the following:

1. *How does Carrefour impact the performance of applications, including those that cannot benefit from its heuristics?*
2. *How does Carrefour compare against existing heuristics for modern NUMA hardware?*
3. *How well does Carrefour leverage the different memory placement mechanisms?*
4. *What is the overhead of Carrefour?*

To assess the performance of *Carrefour*, we compare it against three other configurations: *Linux* – a standard Linux kernel with the default first-touch memory allocation policy, *Manual interleaving* – a standard Linux kernel with the interleaving policy manually enabled for the application, and *AutoNUMA* – a recent Linux patchset [2] considered as the best thread and memory management algorithm available for Linux.

The rest of the section is organized as follows: we first describe our experimental testbed. Next, we study single-application scenarios, followed by workloads with multiple co-scheduled applications. We then detail the overhead of *Carrefour* and conclude with a discussion on additional hardware support that would improve *Carrefour*'s operation.

4.1 Testbed

We used two different machines for the experiments:
Machine A has four 2.3GHz AMD Opteron 8385 processors with 4 cores in each (16 cores in total) and 64GB of RAM. It features 4 nodes (i.e., 4 cores and 16GB of RAM per node) interconnected with HyperTransport 1.0 links.
Machine B has four 2.6GHz AMD Opteron 8435 processors with 6 cores in each (24 cores in total) and 64GB of RAM. It features 4 nodes (i.e., 6 cores and 16GB of RAM per node) interconnected with HyperTransport 3.0 links.

All experiments were performed on both machines. We present the main performance results for both machines. However, due to space constraints, we only show the detailed profiling measurements for machine A (the results obtained on machine B are qualitatively similar).

We used Linux kernel v3.6 for all experiments. For the AutoNUMA configuration, we used AutoNUMA v27 and disabled PMD scan because we it decreases performance on all applications we measured[7].

We used the following set of applications: the PARSEC benchmark suite v2.1 [26], the FaceRec facial recognition engine

[5] *Streamcluster* holds shared data in a single large array, so it is trivial to detect which data is worth replicating and implement a manual solution to use as the performance upper-bound.

[6] We use the threshold of at least 500 pages. Performance is not highly sensitive to this parameter.

[7] We also tested AutoNUMA v28. The performance results are very similar. However, we observed a significantly higher standard deviation (up to 18% on machine A and 23% on machine B) which makes profiling very difficult.

v5.0 [10], the Metis MapReduce benchmark suite [22] and the NAS parallel benchmark suite v3.3 [24]. PARSEC applications run with the native workload. From the available workloads in NAS we chose those with the running time of at least ten seconds. We excluded applications whose CPU utilization was below 33%, because they were not affected by memory management policies. We also excluded applications using shared memory across processes (as opposed to threads) or memory-mapped files, because our replication mechanism does not yet support this behaviour. For *FaceRec*, we used two kinds of workloads: a short-running one and a long running one (named *FaceRecLong* in the remainder of the paper). The reason why we present two different workloads is to show that *Carrefour* is able to successfully handle very short workloads (less than 4s on machine B when running with *Carrefour*). Each experiment was run ten times. Overall, we observed a standard deviation between 1% and 2% for the *Linux*, *Manual interleaving* and *Carrefour* configurations. *AutoNUMA* has a more significant standard deviation (up to 9% on machine A and 13% on machine B).

4.2 Single-application workloads

Performance comparison. Figures 5 and 6 show the performance improvement relative to default Linux obtained under all configurations for machine A and machine B. Performance improvement is computed as:

$$\frac{DefaultLinux_{time} - System_{time}}{System_{time}} * 100\%,$$

where $System$ can be either *Carrefour*, Manual interleaving or AutoNUMA.

We can make two main observations. First, *Carrefour* almost systematically outperforms default Linux, AutoNUMA, and Manual interleaving, sometimes quite substantially. For instance, when running *Streamcluster* or *FaceRecLong*, we observe that *Carrefour* is up to 58% faster than Manual interleaving, up to 165% faster than AutoNUMA, and up to 263% faster than default Linux on machine B. Second, we observe that, unlike other techniques, *Carrefour* never performs significantly worse than default Linux: the maximum performance degradation over Linux is below 4%. In contrast, AutoNUMA and Manual interleaving cause performance degradations of up to 25% and 38% respectively. Besides, we notice that Manual Interleaving has a very irregular impact on performance. While it does fairly well for PARSEC, NAS applications suffer significant performance degradation (up to 38%) when run with manual interleaving.

IS is an exception among the NAS benchmarks: Manual interleaving improves its performance, while *Carrefour* does no better than default Linux. That is because IS suffers from very short imbalance bursts that we are not able to correct due to limited sampling accuracy achievable with low overhead using existing hardware counters. Section 4.6 discusses hardware counter support that would help us address this problem.

In order to understand the reasons for the performance impact of the different policies, we study in detail a set of applications whose performance is improved the most by *Carrefour*. To that end, we present several metrics: the load imbalance on memory controllers (Figure 7(a)), the load imbalance on interconnect links (Figure 7(b)), the average memory latency (Figure 8(a)) and the local access ratio (Figure 8(b)).

We draw the following observations. First, *Carrefour* much better balances the load on both memory controllers and interconnect links than Linux and AutoNUMA. Not surprisingly, Manual interleaving is also very good at balancing the load. Nevertheless, we observe in Figure 8(a) that *Carrefour* induces lower average memory latencies than Manual interleaving, which explains its better performance. To understand why *Carrefour* reduces memory latencies we refer to Figure 8(b), which shows that *Carrefour* not only balances the load on memory controllers and interconnect links, but also often induces a much higher ratio of local memory accesses than other techniques. This is a consequence of *Carrefour*'s judiciously applying the right techniques (interleaving, replication or co-location) in places where they are beneficial. Interleaving mostly balances the load; replication and co-location in addition to balancing the load improve the local access ratio. Better locality improves latencies in two ways: it avoids remote wire delays and, most importantly, decreases congestion on the interconnect links.

We also study MG as a representative example of the NAS applications, for which *Carrefour* does not bring significant improvements over default Linux (but still performs better than AutoNUMA and Manual interleaving in most cases). MG has a low imbalance and a very good local access ratio to begin with. That is why Manual interleaving has a very bad impact on such workloads, significantly decreasing the local access ratio and as a result stressing the interconnect.

Looking inside *Carrefour*. To better understand the behavior of *Carrefour*, we show in Table 3 the number of replicated pages, the number of interleaved pages, and the number of co-located pages for the chosen benchmarks. These numbers provide a better insight into how *Carrefour* manages the memory. We observe that *all* three memory placement mechanisms are in use, and that most applications rely on two or three techniques. As discussed previously, MG does not suffer from traffic congestion, so *Carrefour* does not enable any technique for this application.

A legitimate question we can ask is whether *Carrefour* always selects the best technique. In Table 4, we report the performance improvement over Linux obtained when running a full-fledged *Carrefour* and when running a reduced version of *Carrefour* enabling only one technique at a time. We observe that *Carrefour* systematically selects the best technique. It is also interesting to remark how different techniques work together and how the numbers provided here echo those in Table 3. We notice that, for all the studied applications except MG and SP, the combination of several techniques employed by *Carrefour* outperforms any single technique, even when a given technique has a dominant impact (e.g., for Streamcluster). The slight performance degradation for MG corresponds to the monitoring overhead of *Carrefour*.

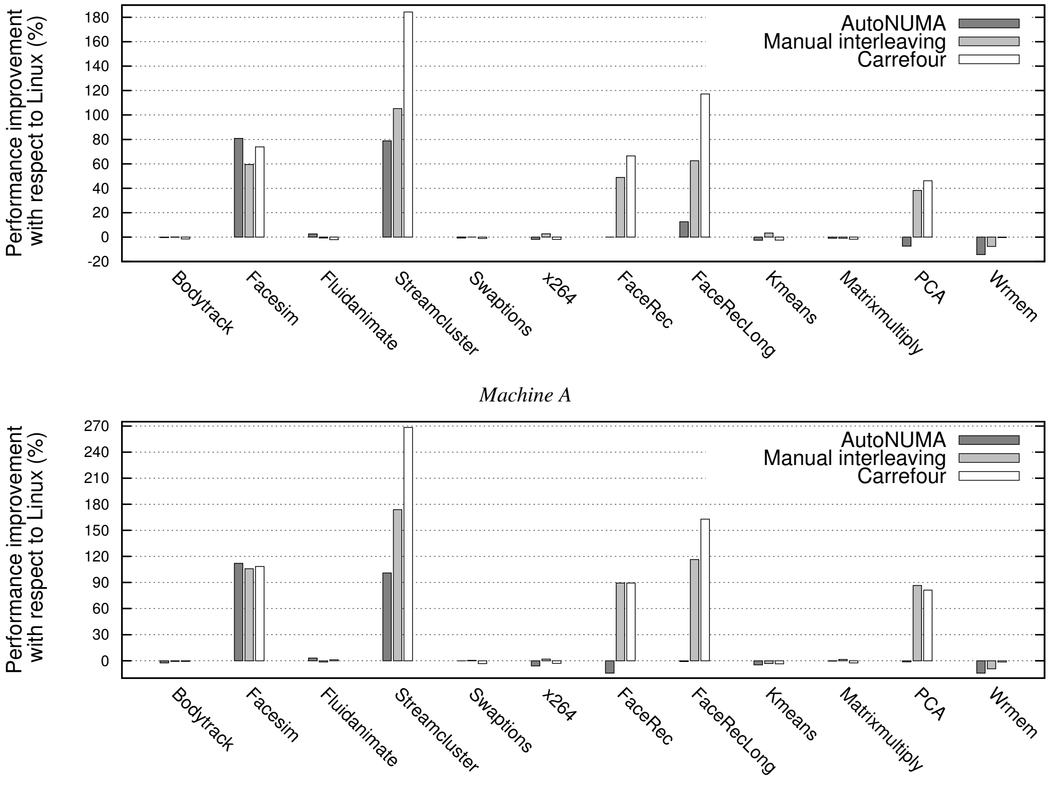
4.3 Multi-application workloads

In this section, we study how *Carrefour* behaves in the context of workloads with multiple applications that are co-scheduled on the same machine. The goal is to assess that *Carrefour* is able to work on complex access patterns and to make the distinction between the diverse requirements of different applications. We consider several scenarios based on some of the applications previously studied in Section 4.2:

(i) *MG + Streamcluster*, (ii) *PCA + Streamcluster*, (iii) *FaceRecLong + Streamcluster*. We chose these scenarios because they exhibit interesting patterns, which require combining several memory placement techniques in order to achieve good performance.

Each application is run with half as many threads as the number of cores (i.e., 8 threads on machine A, 12 on machine B). With two applications, the workload occupies all the available cores. The threads of each application are clustered on the same node, so each application uses all the cores on two of the four nodes on a machine.

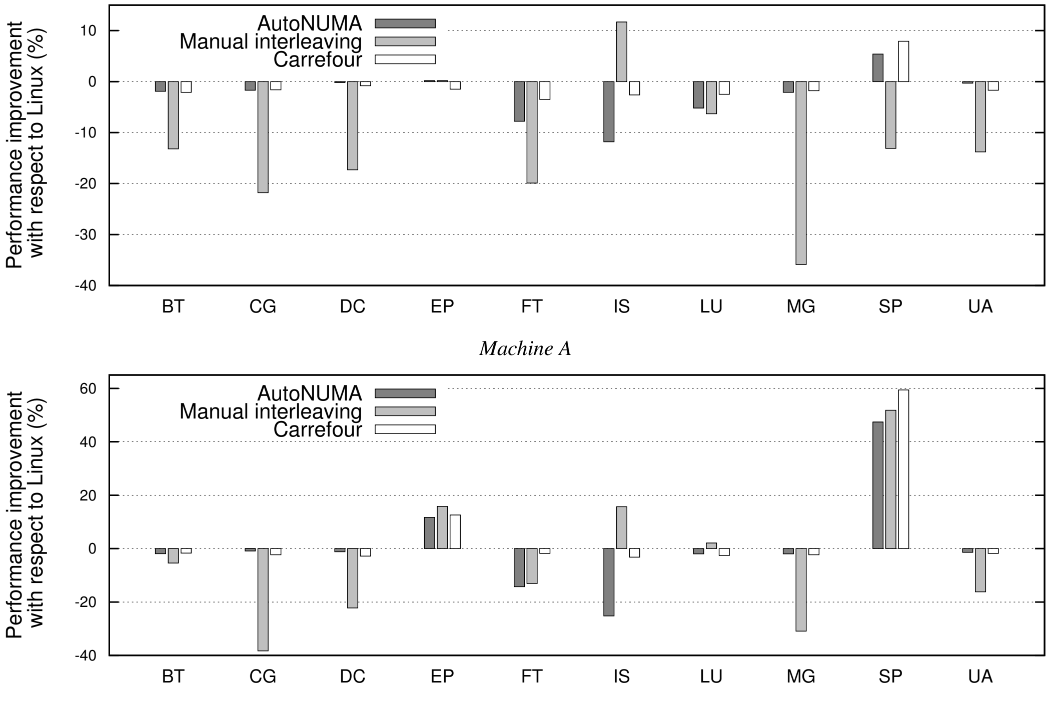
Figure 9 shows, for each workload, the performance improvement with respect to Linux for AutoNUMA, Manual interleaving and *Carrefour* on machine A and machine B. We observe that *Carrefour* always outperforms AutoNUMA and Manual interleaving, by up to 62% and 36% respectively. Besides, *Carrefour* also out-



Machine A

Machine B

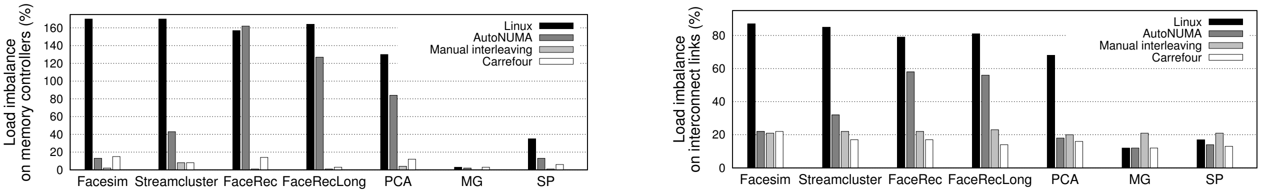
Figure 5. PARSEC/Metis: AutoNUMA, Manual interleaving and *Carrefour* vs.Default Linux.



Machine A

Machine B

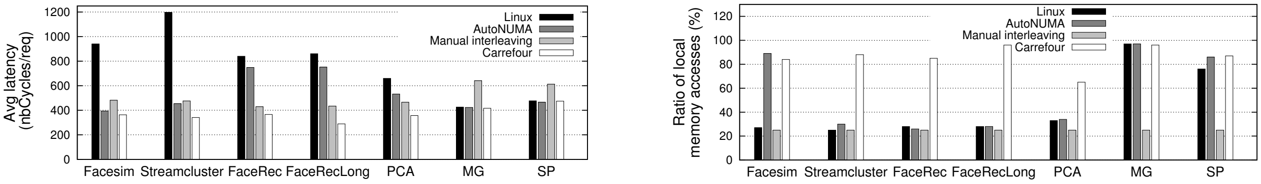
Figure 6. NAS: AutoNUMA, Manual interleaving and *Carrefour* vs. Default Linux.



(a) memory controllers

(b) interconnect links

Figure 7. Load imbalance for selected single-application benchmarks (machine A).



(a) Average memory latency

(b) Local memory access ratio

Figure 8. DRAM latency and locality for selected single-application benchmarks (machine A).

	Nb replicated pages	Nb interleaved pages	Nb migrated pages
Facesim	0	431	10.1k
Streamcluster	25.4k	14.5	858
FaceRec	4k	3	1.3k
FaceRecLong	4.1k	5	1.4k
PCA	31k	33	41.3k
MG	0	0	1
SP	0	305	1.7k

Table 3. Number of memory pages that are replicated, interleaved and co-located on single-application workloads (machine A).

	Carrefour	Replication	Interleaving	Co-location
Facesim	74%	-4%	0%	65%
Streamcluster	184%	176%	94%	51%
FaceRec	66%	61%	32%	1%
FaceRecLong	117%	113%	51%	1%
PCA	46%	45%	29%	24%
MG	-2%	-2%	-2%	-2%
SP	8%	-1%	-7%	8%

Table 4. Performance improvement over Linux when running *Carrefour* and the three different techniques individually on single-application workloads (machine A).

performs default Linux, while Manual interleaving hurts MG with a -25% slowdown. Overall, the results obtained with Manual interleaving are closer to the ones of *Carrefour* compared to the other setups.

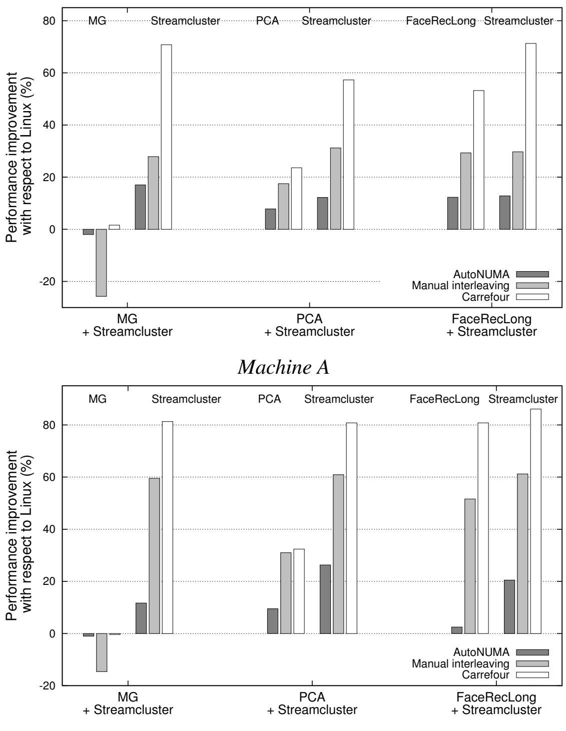
The reason why Manual interleaving performs relatively well in these scenarios is because, with each application using two domains, there is a lot less cross-domain traffic than in the single-application case. Hence there are fewer "problems" that need to be fixed and there is a smaller discrepancy between the performance of different memory management algorithms.

To explain the results, we show the same detailed metrics as in Section 4.2: load imbalance on memory controllers, load imbalance on interconnect links, average DRAM latency and ratio of local DRAM accesses in Figures 10, 11, 12 and 13 respectively. The metrics are aggregated for the two applications of each workload. As was the case with the single-application workloads, we see that *Carrefour* systematically improves the latency of the studied workloads for two reasons: a more balanced load on memory controllers and interconnect links as well as an improved locality for DRAM accesses. Note that, for each workload, the depicted latencies are averaged over the two applications. This explains the small latency variations between Linux, AutoNUMA and Manual Interleaving in the case of MG + Streamcluster.

Finally, we show in Table 5 the performance improvement for each application when the effects of only one of the techniques are enabled. We observe that the multi-applications workloads perform as well or better with an arsenal of techniques used in *Carrefour* rather than with any single technique alone (especially for PCA + Streamcluster).

4.4 Overhead

Carrefour incurs CPU and memory overhead. The first source of CPU overhead is the periodic IBS profiling. To measure CPU overhead, we compared performance of *Carrefour* with Linux on those applications where *Carrefour* does not yield any performance benefits. We observed the overhead between 0.2% and 3.2%. Adaptive sampling rate in *Carrefour* is crucial to keeping this overhead low. A second and potentially significant source of CPU overhead is replication, if we perform a lot of collapses. A single collapse costs a few hundred microseconds when it occurs in isolation. Parallel



Machine A

Machine B

Figure 9. Multi-application workloads: AutoNUMA, Manual interleaving and *Carrefour* vs. Default Linux.

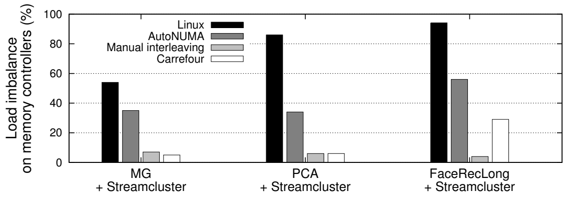


Figure 10. Multi-application workloads: load imbalance on memory controllers (machine A).

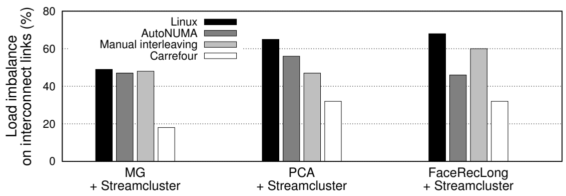


Figure 11. Multi-application workloads: load imbalance on interconnect links (machine A).

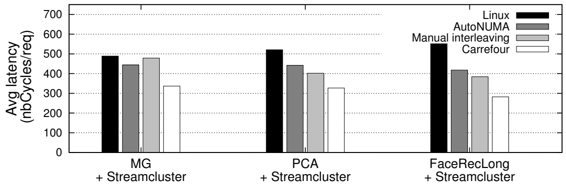


Figure 12. Multi-application workloads: average memory latency (machine A).

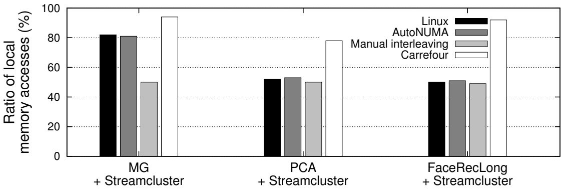


Figure 13. Multi-application workloads: local memory access ratio (machine A).

	Carrefour	Replication	Interleaving	Co-location
MG + Streamcluster	2% / 71%	2% / 73%	-5% / 17%	-1% / 6%
PCA + Streamcluster	24% / 57%	18% / 57%	8% / 5%	14% / 1%
FaceRecLong + Streamcluster	53% / 71%	53% / 71%	12% / 9%	12% / 4%

Table 5. Multi-application workloads: performance improvement over Linux when running *Carrefour* and the three different techniques individually (machine A).

collapses can take a few milliseconds because of lock contention. That is why it is crucial to avoid collapses and other synchronization events by disabling replication for write-intensive workloads, as is done in *Carrefour*.

The first source of memory overhead is the allocation of data structures to keep track of profiling data. This overhead is negligible: e.g., 5MB on Machine A with 64GB of RAM. *Carrefour*'s data structures are pre-allocated on startup to avoid memory allocation during the runtime. We limit the number of profiled pages to 30,000 to avoid the cost of managing dynamically sized structures. The second source of memory overhead is memory replication. When enabled, replication introduces a memory footprint overhead of 400MB (353%), 60MB (210%), 60MB (126%) and 614MB (5%) for Streamcluster, FaceRec, FacerecLong and PCA respectively.

4.5 Impact on energy consumption

It was observed that remote memory accesses require significantly more energy than local ones [11]. Since *Carrefour* may both decrease and increase the number of remote memory accesses, we were interested in evaluating its impact on energy consumption[8]. We show the results for selected applications from single-application workloads on Machine A. We report the increase in energy consumption as well as the increase in completion time of all configurations compared to default Linux in Figure 14. Completion time increase is computed here as:

$$\frac{System_{time} - DefaultLinux_{time}}{DefaultLinux_{time}} * 100\%.$$

We observe that there is a strong relationship between the completion time and the energy consumption: if the completion time is decreased, the energy consumption is also decreased proportionally. As a result, *Carrefour* saves up to 58% of energy. When no traffic management is needed, *Carrefour* on its own has a low impact on energy consumption (e.g., 2% on MG).

More generally, we found that the increase of remote memory accesses has little or no impact on the global energy consumption of the machine. For example, Manual interleaving drops the local access ratio of MG from 97% to 25% and thus proportion-

[8] We used IPMI which gives access to the current global power consumption on our servers.

ally increases the number of remote accesses. However, the energy consumption increase is slightly lower that the completion time increase, which indicates that the extra energy overhead of remote memory accesses have no strong impact on overall energy consumption.

4.6 Discussion: hardware support

We have shown that a traffic management system like *Carrefour* can bring significant performance benefits. However, the challenge in building *Carrefour* was the need to navigate around the limitations of the performance monitoring units of our hardware as well as the costs of replicating pages. In this section, we draw some insights on the features that could be integrated into future machines in order to further mitigate the overhead and improve accuracy, efficiency and performance of traffic management algorithms.

First, *Carrefour* would benefit from hardware profiling mechanisms that sample memory accesses with high precision and low overhead. For instance, it would be useful to have a profiling mechanism that accumulates and aggregates page access statistics in an internal buffer before triggering an interrupt. In this regard, the AMD Lightweight Profiling [1] facility seems a promising evolution of profiling hardware[9], but we believe the hardware should go even further, and not only accumulate the samples but be configured to aggregate them according to user needs, to reduce the number of interrupts even further.

Second, *Carrefour* would benefit from dedicated hardware support for memory replication. We believe that there should be interfaces allowing the operating system to indicate to the processor which pages to replicate. The processor would then be in charge of replicating the pages on the nodes accessing it and maintaining consistency between the various replicas (in the same way as it maintains consistency for cache lines). Given that maintaining consistency between frequently written pages is costly, we believe that such processors should also be able to trigger an interrupt when a page is written too frequently. The OS would then decide to keep the page replicated or to revert the replication decision.

This hardware support can be made a lot more scalable than cache coherency protocols, because it is not automatic, but controlled by the OS, which, armed with better hardware profiling, will only invoke it for pages that perform very little write sharing. So the actual synchronization protocol would be triggered infrequently.

5. Related Work

In this section, we explain how *Carrefour* relates to different works on multicore systems. First, we review systems aimed at maximizing data locality. Second, we contrast *Carrefour* with previous contention-aware systems. Third, we consider application-level techniques to mitigate contention on data-sharing applications. Finally, we discuss traffic characterization observations for modern NUMA systems.

Locality-driven optimizations NUMA-aware thread and memory management policies were proposed for earlier research systems [6, 9, 18, 31] as well as in commercial OS. Their main difference from our work is that their goal was to optimize *locality*. However, on modern systems, the main performance problems are due to traffic congestion. Our algorithm is the first one that meets the goal of mitigating traffic congestion. Among the above-mentioned works, the one most related to Carrefour is the system by Verghese et al. [31] for early cache-coherent NUMA machines, which leverages page replication and migration mechanisms. Their system relies on assumptions about hardware support that do not hold on currently available machines (e.g., precise per-page access statistics). Thus, *Carrefour*'s logic is more involved, as it is more difficult to amortize the costs of the monitoring and memory page placement mechanisms. The authors noticed that locality-driven optimizations could, as a side effect, reduce the overall contention in the system. However, their system does not systematically address contention issues. For instance, shared written pages are not taken into account, whereas *Carrefour* uses memory interleaving techniques when there is contention on such pages. Moreover, the load on memory controllers is ignored when making page replication/migration decisions.

Similarly to earlier NUMA-aware policies, Solaris and Linux focus primarily on co-location of threads and data, but to the disadvantage of data-sharing workloads, replication is not supported. Linux provides the option to interleave parts of the address space across all memory nodes, but the decision when to invoke the interleaving is left to the programmer or the administrator. Solaris supports the notion of a home load group, such that the thread's memory is always allocated in its home group and the thread is preferentially scheduled in its home group. This, again, favours locality, but does not necessarily address traffic congestion.

The recent AutoNUMA patches for Linux also implement locality-driven optimizations, along two main heuristics. First, threads migrate toward nodes holding the majority of the pages accessed by these threads. Memory residence is determined by page fault statistics. Second, pages are periodically unmapped from a process address space and, upon the next page fault, migrated to the requesting node. As shown in the evaluation section, this approach yields irregular results. We attribute this limitation to the following sources of overhead: local thread/page migration decisions that do not take data sharing patterns nor access frequencies into account (thus leading to page bouncing or useless migrations) nor MC/interconnect load (thus leading to memory load imbalance/congestion), continuous overhead due to the scanning/unmapping of page-table entries and the corresponding soft page faults. In contrast, *Carrefour* makes global data placement decisions based on precise traffic patterns and adjusts the monitoring overhead based on the observed contention level.

Locality-driven optimizations for data-sharing applications were addressed in a study that dynamically identified data-sharing thread groups and co-located them on the same node [30]. However, that solution was for a system with a centralized (UMA) memory architecture. Thus, it only studied the benefits of thread placement for improved cache locality and did not address the placement of memory pages on multiple memory nodes.

Zhou and Demsky [32] investigated how to distribute memory pages to caches on a many-core Tilera processor, in order to implement an efficient garbage collector. The authors tried various policies but found that maximizing locality was the best approach for their system. This is in contrast to the systems *Carrefour* is targeting, where reducing congestion is more important than just improving locality. Also, the authors used a very different method for monitoring page access patterns that relies on software-serviced TLB misses, which is not possible on x86.

Contention management techniques Several recent studies addressed contention issues in the memory hierarchy. Some of these works were designed for UMA systems [16, 21, 33] and are inefficient on NUMA systems because they fail to address or even accentuate issues such as remote access latencies and contention on memory controllers and on the interconnect links [5]. Other works have been specifically designed for NUMA systems but only partially address contention issues. The N-Mass thread placement algorithm [19] attempts to achieve good DRAM locality while avoid-

[9] Unfortunately, Lightweight Profiling is only available on recent AMD processors (AMD Bulldozer) and we were not able to evaluate it in this work. We plan to study it in the future.

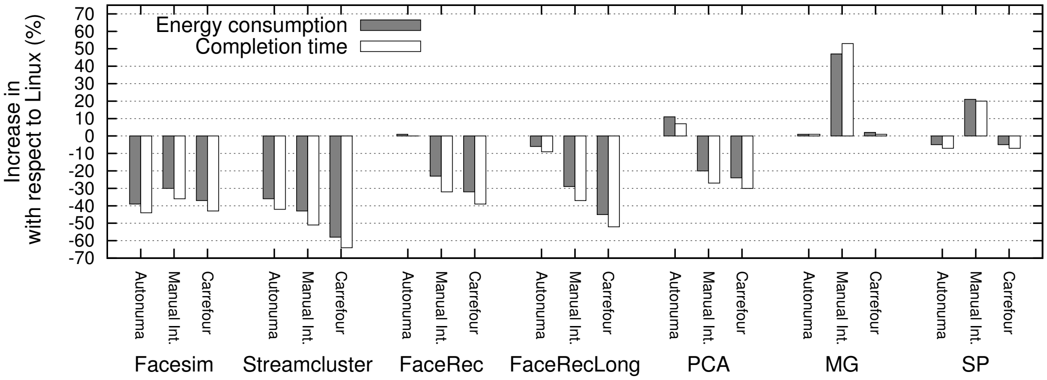


Figure 14. Increase in completion time and energy consumption for selected single-application benchmarks with respect to Linux (machine A). Lower is better.

ing cache contention. However, it does not address contention issues at the level of memory controllers and interconnect links. Two studies [3, 20] have shown the importance of taking memory controller congestion into account for data placement decisions, but they did not provide a complete solution to address multi-level resource contention. The most comprehensive work to date on NUMA-aware contention management is the DINO scheduler [5], which spreads memory intensive threads across memory domains and accordingly migrates the corresponding memory pages. However, DINO does not address workloads with data sharing between threads or processes, which require identifying per-page memory access patterns and making the appropriate data placement decisions.

No-sharing application design principle When introducing a new resource-management policy in the OS, it is worth asking whether a similar or better effect could be achieved by restructuring the application. In our context, it is important to consider the so-called *no-sharing* principle of application design. The key idea behind no-sharing is that the data must be partitioned or replicated between memory nodes, and a thread needing to access data in a different domain than its own either migrates to the target domain or asks the thread running in that domain to perform the work on its behalf, instead of fetching the data over the memory channels [4, 8, 14, 23, 25, 28, 29]. While the no-sharing architecture was primarily motivated by the need to avoid locking, it could similarly help reduce the amount of traffic sent across the interconnect, and thus alleviate the traffic congestion problem.

Unfortunately, no-sharing architectures are not a universal remedy. First of all, they trade-off data accesses for messages or thread migration; the trade-off is only worth making if the size of the data used in a single operation is much larger than the size of the message or the state of the migrated thread [8]. Second, adopting a no-sharing architecture often requires very significant changes to the application (and to the OS, if the application is OS-intensive). A good illustration of the potential challenges can be gleaned from two studies that converted a database system to the no-sharing design. The first study took the path of least resistance and simply replicated and/or partitioned the database among domains, adding a message-routing layer on top [28]. While this worked well for small read-mostly workloads, for large workloads replication had very significant memory overhead (unacceptable because of increased paging), and partitioning required *a priori* knowledge of query-to-data mapping, which is not a reasonable assumption in a general case. A solution that overcame these limitations, DORA [25], required a very significant restructuring of the database system, which could easily amount to millions of dollars in development costs for a commercial database.

Our goal was to address scenarios where adopting a no-sharing architecture is not feasible either for technical reasons or for practical considerations. Providing an OS-level, rather than an application-level, solution allows us to address many applications at once. Understanding the limitations of the OS-level solution and determining what optimizations can be done only at the level of an application is an open research question. Although we did provide some comparison of *Carrefour* with application-level techniques, a deeper analysis of this problem would be worthwhile.

Traffic characterization on modern systems A recent study characterized the performance of emerging ”scale-out” workloads on modern hardware [13]. The authors observed that there is little inter-thread data sharing, and the utilization of the off-chip memory bandwidth is low. As a result, they argue that memory bandwidth on existing processors is unnecessarily high. Our findings do not agree with this observation. Although it is also true that the workloads we consider perform very little fine-grained data sharing, they still stress the cross-chip interconnect, because they access a large working set, which is spread across the entire NUMA memory space. The authors of the scale-out study reported a very low bandwidth utilization (<10%), even for database workloads. In contrast, our measurements show a utilization as low as 45% in some cases. These differences could be because the authors of [13] used a system with only two chips and 12 cores in their experiments. On larger systems, more threads are making requests to remote memories and so there is greater pressure on bandwidth. Further, we found that low bandwidth utilization is not necessarily a healthy symptom. In our experiments, performance dropped steeply even as bandwidth utilization went from 10% to 30%. The interconnect became the bottleneck even at a fraction of the total available bandwidth! The reason is that memory requests are not spread evenly in time; they are bursty. Burstiness causes requests to clash on the link even if the overall bandwidth is not exceeded. In summary, we conclude that contrary to the suggestion made in [13], it is too early to reduce the bandwidth of cross-chip interconnects on large multicore systems, especially if they are used for running large data-centric workloads.

6. Conclusion

We presented *Carrefour*, a new memory management algorithm for NUMA systems that manages traffic on memory controllers and interconnects. Earlier NUMA-aware memory management policies aimed to mitigate the cost of remote wire delays, which is no longer the main bottleneck on modern systems. *Carrefour*'s design was motivated by the evolution of modern NUMA hardware, where traffic congestion plays a much larger role in performance than wire delays.

System design principles embodied in *Carrefour* are important not only for today's systems, but also for future hardware. The amount of memory bandwidth per core is projected to decrease in the future [8], so managing traffic congestion will be as crucial as ever.

7. Code availability

The source code for *Carrefour* will be made available at https://github.com/Carrefour.

8. Acknowledgments

We thank Oracle Labs and the British Columbia Innovation Council for funding this work.

References

[1] AMD64 Technology Lightweight Profiling Specification, Aug. 2010. `http://support.amd.com/us/Processor_TechDocs/43724.pdf`.

[2] AutoNUMA: the other approach to NUMA scheduling. LWN.net, Mar. 2012. `http://lwn.net/Articles/488709/`.

[3] 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.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

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

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

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *OSDI*, 2008.

[8] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. A software approach to unifying multicore caches. Technical Report MIT-CSAIL-TR-2011-032, 2011.

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

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

[11] B. Dally. Power, programmability, and granularity: The challenges of exascale computing. `http://techtalks.tv/talks/54110`.

[12] P. Drongowski and B. Center. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. 2007.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *ASPLOS*, 2012.

[14] B. Gamsa, O. Krieger, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *OSDI*, 1999.

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

[16] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):pp. 54–66, 2008.

[17] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX ATC*, 2012.

[18] R. P. LaRowe, Jr., C. S. Ellis, and M. A. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE TPDS*, 3:686–701, 1991.

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

[20] Z. Majo and T. R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *SYSTOR*, 2011.

[21] A. Merkel, J. Stoess, and F. Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *EuroSys*, 2010.

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

[23] Z. Metreveli, N. Zeldovich, and F. Kaashoek. Cphash: a cache-partitioned hash table. In *PPoPP*, 2012.

[24] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.

[25] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939, September 2010. ISSN 2150-8097.

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

[27] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys*, 2010.

[28] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *EuroSys*, 2011.

[29] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with OS clustering. In *EuroSys*, 2011.

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

[31] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS*, 1996.

[32] J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. In *ISMM*, 2012.

[33] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.