

Optimizing NUMA systems applications with Carrefour.

BY FABIEN GAUD, BAPTISTE LEPEERS, JUSTIN FUNSTON, MOHAMMAD DASHTI, ALEXANDRA FEDOROVA, VIVIEN QUÉMA, RENAUD LACHAIZE, AND MARK ROTH

Challenges of Memory Management on Modern NUMA Systems

MODERN SERVER-CLASS SYSTEMS are typically built as several multicore chips put together in a single system. Each chip has a local DRAM (dynamic random-access memory) module; together they are referred to as a *node*. Nodes are connected via a high-speed interconnect, and the system is fully coherent.

This means that, transparently to the programmer, a core can issue requests to its node's local memory as well as to the memories of other nodes. The key distinction is that remote requests will take longer, because they are subject to longer wire delays and may have to jump several hops as they traverse the interconnect. The latency of memory-access times is hence non-uniform, because it depends on where the request originates and where it is destined to go. Such systems are referred to as non-

uniform memory access (or NUMA).

Systems with NUMA characteristics were built as early as the 1980s, and along with the hardware operating system, support for NUMA has evolved. Modern NUMA systems are quite different from the old ones, so we must revisit our assumptions about them and rethink how to build NUMA-aware operating systems. This article evaluates performance characteristics of a representative modern NUMA system, describes NUMA-specific features

in Linux, and presents a memory-management algorithm that delivers substantially reduced memory-access times and better performance.

A Modern NUMA System

NUMA systems consist of several nodes, each containing a subset of the system's CPU cores and a portion of its RAM. If a

core accesses memory from within the same node, it is called a *local access*. Similarly, an access to a different node is called a *remote access*. Remote accesses have longer latencies than local ones, because they must traverse one or more *interconnect links*, communication pathways between nodes that also service cache-coherency traffic. Figure

1 is a diagram of a typical NUMA system with four nodes and four cores per node. At the time of this writing, NUMA systems are built with up to eight nodes and 10 cores per node.

Current x86 NUMA systems are cache coherent (called ccNUMA), which means programs can transparently access memory on local and remote nodes without changes to the code or special operating system support. This allows easy migration to NUMA systems, but it does not address important performance considerations. A naïve implementation—for example, a program that allocates all of its memory on a single node—can easily cause excessive remote accesses or the overloading of a memory controller.

New vs. old NUMA systems. Avoiding performance pitfalls on NUMA systems requires considering how the nodes are connected, where the program's memory is placed, and how it accesses that memory. Previous NUMA-aware operating systems focused on locality, attempting to minimize the number of remote accesses at all costs in order to avoid the performance penalty. Modern NUMA systems, however, have a strikingly different latency profile compared with the older ones. A remote access takes approximately 30% longer than a local one,^{2,7} while on older hardware, it could take up to seven times longer.³ The remote-access penalty is substantially reduced on modern NUMA systems.

On the other hand, current CPUs can generate an immense load on the memory subsystem, causing congestion on memory controllers and interconnect links (if requests are remote). If multiple cores are heavily accessing a single node, memory latencies can be as long as 1,200 cycles (!) due to congestion, while normal latencies are only around 300 cycles. Avoiding memory controller and interconnect congestion, therefore, becomes the key concern on modern NUMA systems. Here, we examine the effects of congestion on performance.

NUMA Performance—Locality And Congestion

Benchmarks can provide a complete picture of the performance characteristics of NUMA systems. The Numerical Aerodynamic Simulation (NAS),¹ Princeton Application Repository for

Figure 1. A modern NUMA system.

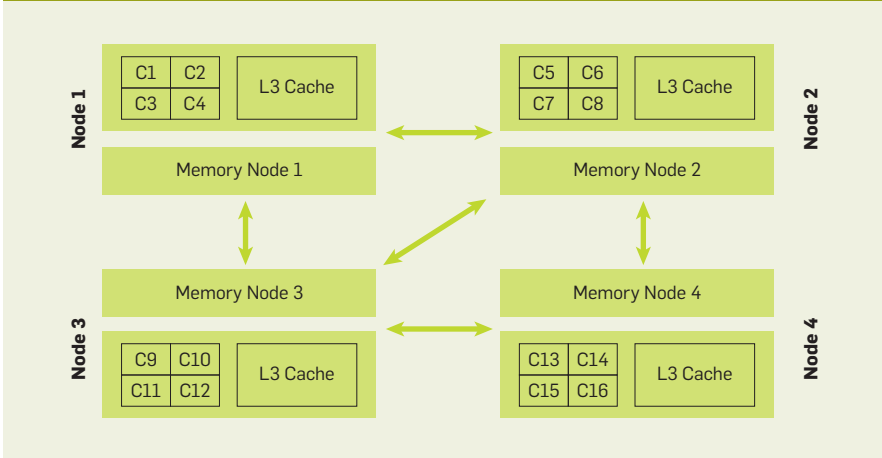
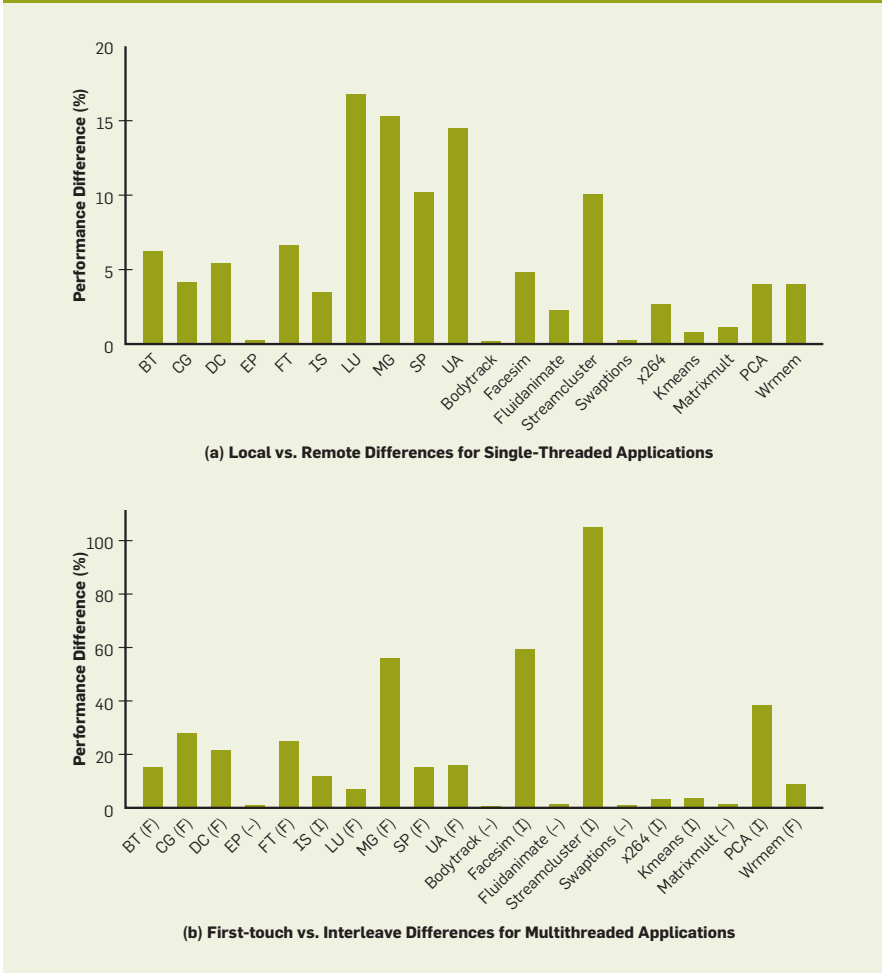


Figure 2. Performance differences.




Shared-Memory Computers (PARSEC)¹⁰ and Metis MapReduce⁹ suites were chosen here because they have a CPU utilization greater than 30%, allowing the focus to be on NUMA effects and not other factors such as disk I/O or blocking synchronization. The experiments were conducted on an AMD server with four quad-core CPUs, as we describe later.


The first experiment quantified the effect of only the remote-access penalty, without the presence of memory-subsystem congestion. To do this, the benchmarks were run with only a single thread, limiting the pressure on the memory controllers and interconnects, and then compared under two different memory configurations: local and remote. In the local memory configuration, applications were executed with their memory and thread on the same node. In the remote memory configuration, standard Linux tools were used to force the application thread to run on a different node from its memory. Therefore, in the remote-memory case all memory accesses were remote, and in the local-memory case all memory accesses were local.

Figure 2a shows performance differences between local and remote memory configurations for the single-threaded versions of applications used in this experiment. Performance never degraded by more than 20%, even when all memory requests were remote.

Although the remote-access penalty is worth minimizing when possible, that is not the whole story of NUMA performance effects. To demonstrate this, the benchmarks were run using multiple threads, with one thread per core and under two different common NUMA memory-allocation policies: *first-touch* and *interleave*. Linux's default policy is *first-touch*, where memory is allocated on the same node as the thread that first accesses a memory page. The *first-touch* policy is meant to maximize local accesses over remote accesses, but of course it cannot guarantee local accesses because threads on multiple nodes can share data. On the other hand, the *interleave* policy distributes memory allocations equally on all nodes regardless of which threads access it. *Interleaving* ensures that memory allocations are balanced but not necessarily that



Avoiding performance pitfalls on NUMA systems requires considering how the nodes are connected, where the program's memory is placed, and how it accesses that memory.



memory accesses will be balanced. Both policies work at the granularity of a page (typically 4KB).

Figure 2b shows the absolute performance difference between *first-touch* (F) and *interleave* (I) for multithreaded versions of applications. The applications are labeled (F) or (I) depending on which policy performed best. The figure compares the two policies by showing the performance difference between the best and worst policy for each benchmark. If there was no negligible difference, the application is labeled (-). The first observation to make is that no one policy is best for all applications. Several applications perform best with the *first-touch* policy, but many prefer *interleaving*. The second observation is that NUMA effects beyond the remote-access penalty can indeed severely affect performance. For the *Streamcluster* benchmark, using the *first-touch* policy nearly doubled the running time over the *interleave* policy.

We further investigated the NUMA performance characteristics of *Streamcluster* and *PCA* (another benchmark that has significant performance loss with the *first-touch* policy) by using hardware performance counters to gather the following key metrics:

- ▶ *Local access ratio*. The portion of RAM accesses that result in a local access.
- ▶ *Memory latency*. The number of cycles it takes to perform a RAM access, on average. Higher latencies mean the CPU must stall for longer on a last-level cache miss, which will negatively affect performance.
- ▶ *Memory-controller imbalance*. The standard deviation (as a percentage of the mean) of the load on the memory controllers, where the load is measured as the number of requests per time unit. Along with interconnect imbalance, it is a sign of congestion.
- ▶ *Average interconnect usage*. The average bandwidth utilization of the interconnect links. A low interconnect usage could imply that either the application is not very memory intensive, or there is imbalance because some links are left underutilized.
- ▶ *Average interconnect imbalance*. The standard deviation (as a percentage of the mean) of the bandwidth utilization of interconnect links.
- ▶ *L3MPKI*. The number of last-level cache misses per 1,000 instructions.

This is a relative indicator of how much pressure an application puts on the memory subsystem and of how sensitive an application is to memory latencies.


► *Instructions per cycle (IPC)*. For the same application and workload, a higher IPC means better performance.

The metrics for Streamcluster and PCA are reported in the accompanying table. Traffic congestion effects are highlighted by differences in key NUMA metrics for each benchmark under the first-touch (F) and interleave (I) policies. The performance difference between the two NUMA policies cannot be explained by a change in the last-level cache miss rate, which stays the same. Nor can it be explained by the local-access ratio, which stays the same for Streamcluster and in fact is worse for PCA in the case of the better performing interleave policy. The local-access ratio of PCA drops from 33% to 25% when interleaving memory, but performance improves significantly, so the conclusion is that better locality does not necessarily improve performance.


When using the first-touch policy both applications show signs of congestion with high last-level cache miss rates, memory-controller imbalance, and interconnect imbalance. The congestion results in high memory latencies. In the case of Streamcluster, the average memory latency with the first-touch policy is more than double the latency of the interleave policy. Interleaving balances the memory among the nodes, which reduces traffic hotspots and congestion, and therefore improves memory latency and overall performance. A visualization of the memory traffic and congestion of Streamcluster is shown in Figure 3; traffic imbalance under first-touch is shown on the top and interleaving on the bottom. Nodes and links bearing the majority of the traffic are shown proportionately larger in size and in brighter colors. The percentage values show the fraction of memory requests destined for each node.

NUMA Memory Placement Strategies

The results in Figure 2 and the table motivate a NUMA memory-management algorithm that places importance on congestion management, rather than focusing solely on reduc-



A NUMA memory-management algorithm should place importance on congestion management, rather than focusing solely on reducing remote accesses.



ing remote accesses. That is not to say reducing remote accesses is not important (they do, after all, add latency and contribute to interconnect congestion), but this should not be the only goal. Managing congestion effectively means being concerned with how the memory-access traffic is spread across the system. It is not enough simply to use interleaving. Many applications do not suffer from imbalance, so they would needlessly incur remote-access delays (for example, the benchmarks in Figure 2b that prefer the first-touch policy). The algorithm must be able to intelligently place memory based on the application's access patterns, such that congestion is reduced whenever possible but locality is not sacrificed when congestion is minimal. Since access patterns are not known *a priori*, the algorithm must also be able to determine the access patterns and move memory at runtime with low overhead.

Later, we present our NUMA algorithm, called Carrefour, which takes all of these considerations into account. First, though, we describe existing NUMA tools available on Linux. (This article gives an overview of the Carrefour algorithm. Please see Dashti et al.⁶ for a complete discussion, including implementation details and exhaustive experimental results.)

NUMA on Linux. Linux allows administrators to set the NUMA policy for applications via the `numactl` utility. The NUMA policies available are first-touch, interleave, and restricting allocations to specific nodes. As described earlier, first-touch is the policy of allocating memory on the same node as the CPU that first accesses the memory page, and the interleave policy distributes memory pages on nodes in a round-robin manner.

Linux exposes manual NUMA memory-management functions to programmers through the `libnuma` library and associated system calls. This allows a program to query the NUMA topology, set NUMA policies for specific address ranges, and migrate memory pages to different nodes at runtime. (See Lameter⁸ for detailed information on Linux's NUMA facilities.)

Linux also provides robust support for hardware performance counters, which are used for counting CPU events such as cycles elapsed, instructions re-

tired, branch mispredictions, or cache misses. These events can be used to calculate the important NUMA metrics listed previously. Perf is the standard Linux tool for using hardware performance counters to profile applications. It can gather data for several events with negligible performance overhead and minimal developer effort.

Hardware instruction sampling is an advanced CPU feature similar to performance counters. With instruction sampling, a proportion of instructions are tagged by the hardware. Tagged instructions will record extra information about their execution. It is necessary for obtaining some NUMA-related statistics, including memory-access latency and the addresses of memory accesses. The feature is implemented as IBS (instruction-based sampling) on AMD CPUs and as PEBS (precision event-based sampling) on Intel CPUs. Unfortunately, Linux support for hardware-instruction sampling is limited and requires a custom kernel module for most uses.

AutoNUMA. AutoNUMA⁴ aims to provide Linux with a more proactive NUMA solution. A kernel task routinely iterates through the allocated memory of each process and tallies the number of memory pages on each node for that process. It also clears the present bit on the pages, which will force the CPU to stop and enter the page-fault handler when the page is next accessed.

In the page-fault handler it records which node and thread is trying to access the page before setting the present bit and allowing execution to continue. Pages that are accessed from remote nodes are put into a queue to be migrated to that node. After a page has already been migrated once, though, future migrations require two recorded accesses from a remote node, which is designed to prevent excessive migrations (known as page bouncing).

AutoNUMA's memory-placement algorithm, now known as Automatic NUMA Balancing, has been merged into the Linux kernel. It can be enabled through the sysctl interface by setting `kernel.numa_balancing` to 1.

AutoNUMA also uses thread placement to try to improve locality. The scheduler will consider migrating or swapping threads if it will cause more of the thread accesses to be local (based on the gathered page-fault statistics)

but not at the cost of introducing load imbalance among CPUs.

Carrefour is a memory-placement algorithm for NUMA systems that focuses on traffic management: placing memory so as to minimize congestion on interconnect links or memory controllers.

Carrefour uses global information and memory-usage statistics to inform three primary techniques for limiting congestion:

- *Memory collocation.* Moving memory to a different node so accesses will likely be local.

- *Replication.* Copying memory to several nodes so threads from each node can access it locally (useful for read-only and read-mostly data).

- *Interleaving.* Moving memory such that it is distributed evenly among all nodes.

All three of these techniques have been analyzed individually in prior studies, but Carrefour combines them into a novel algorithm that is effective for modern NUMA systems.

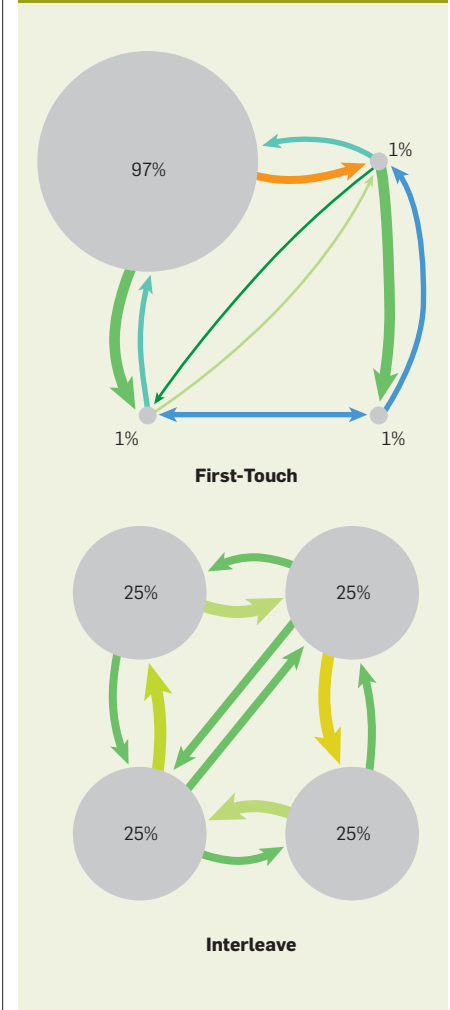
To combine these techniques and apply them judiciously, Carrefour collects per-page, per-process, and global statistics from hardware performance counters. Carrefour also uses hardware-instruction sampling to log which threads and nodes access which memory pages. Instruction sampling lets Carrefour gather many more samples at low overhead than AutoNUMA's page-fault handler technique because it has lower overhead per sample.

The first metric Carrefour uses is the number of RAM accesses per microsecond. If it is less than the threshold (50 in our experiments), then the rest of the algorithm is completely disabled until it becomes greater than the threshold. If the rate of RAM accesses is low, then

the application is unlikely to benefit from better memory placement, so this rule prevents the overhead of Carrefour when it is not needed. If the algorithm does remain enabled, then Carrefour iterates over the memory pages for which it has gathered statistics and applies the replication, collocation, and interleaving techniques.

We implemented memory replication in the Linux kernel with a patch

Figure 3. A composable select element.



Traffic congestion effects.

	Streamcluster		PCA	
	Best (I)	Worst (F)	Best (I)	Worst (F)
Local-access ratio	25%	25%	25%	33%
Memory latency	476	1197	465	660
Memory-controller imbalance	8%	170%	5%	130%
Interconnect imbalance	22%	85%	20%	68%
Interconnect usage	59%	33%	48%	31%
L3MPKI	16.85	16.89	7.35	7.4
IPC	0.29	0.15	0.52	0.36

to the virtual memory layer, and our implementation is able to automatically maintain consistency when there is a write to a replicated page. To enable replication there must be enough

free memory available, and at least 95% of the application's memory accesses must be reads because the performance cost of a write to a replicated page is quite high. The rule for replicat-

ing particular pages is simple: pages are replicated if they are observed to have accesses from multiple nodes in read-only mode. Replication improves both locality and congestion because a replicated page can be accessed locally from more than one node.

Collocation is enabled if the local access ratio is less than 80%. Pages that have been accessed only by a single remote node are migrated to that node, thereby improving locality.

The primary purpose of interleaving is to alleviate congestion by distributing memory—and, therefore, memory accesses—among multiple nodes. The first step is to consider the memory-controller imbalance. If it is below 35%, then interleaving is deemed unprofitable and it is disabled globally. Otherwise, pages that have recorded read and write accesses from more than one node are migrated to a random node, where the probability of being migrated to a specific node is inversely proportional to the relative load on that node's memory controller.

The source code for Carrefour is available at <https://github.com/Carrefour>.

Evaluation

Testbed. All experiments were conducted on an AMD system with 64GB of RAM and four quad-core Opteron 8385 processors running at 2.3GHz. It is divided into four NUMA nodes with four cores and 16GB of RAM per node (the topology is shown in Figure 1) interconnected with HyperTransport 1.0 links.

The operating system was Linux kernel v3.6, and the AutoNUMA configuration used v27 of the patch.

A variety of multithreaded benchmarks were used for the evaluation: PARSEC benchmark suite v2.1,¹⁰ FaceRec v5.0,⁵ Metis MapReduce benchmark suite,⁹ and the NAS parallel benchmark suite v3.3.¹ The PARSEC benchmarks used the “native” workloads, and the NAS benchmarks used problem sizes that provided running times of at least 10 seconds. Applications that had CPU utilizations below 33% were excluded because they are not affected by memory-management policies. Each configuration and benchmark was run 10 times, which resulted in standard deviations of less than 2% for the Carrefour, default Linux, and interleaving configurations. AutoNUMA

Figure 4. PARSEC and Metis.

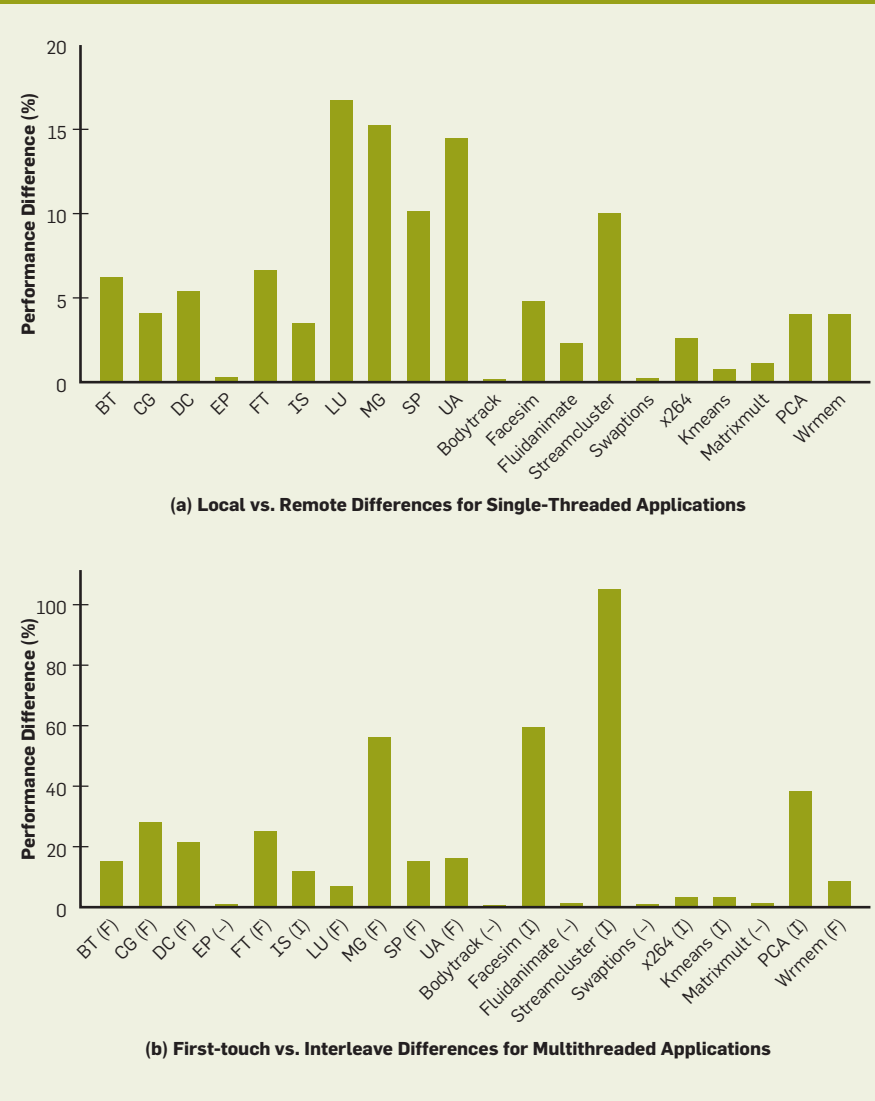


Figure 5. NAS parallel benchmarks.

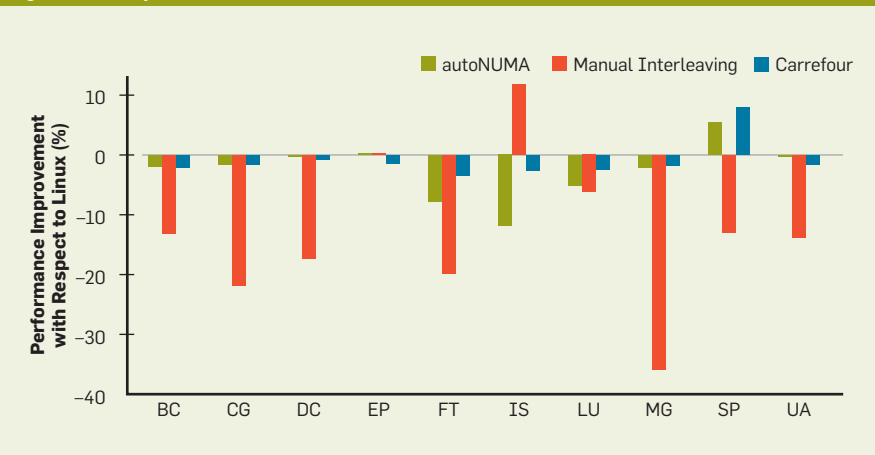
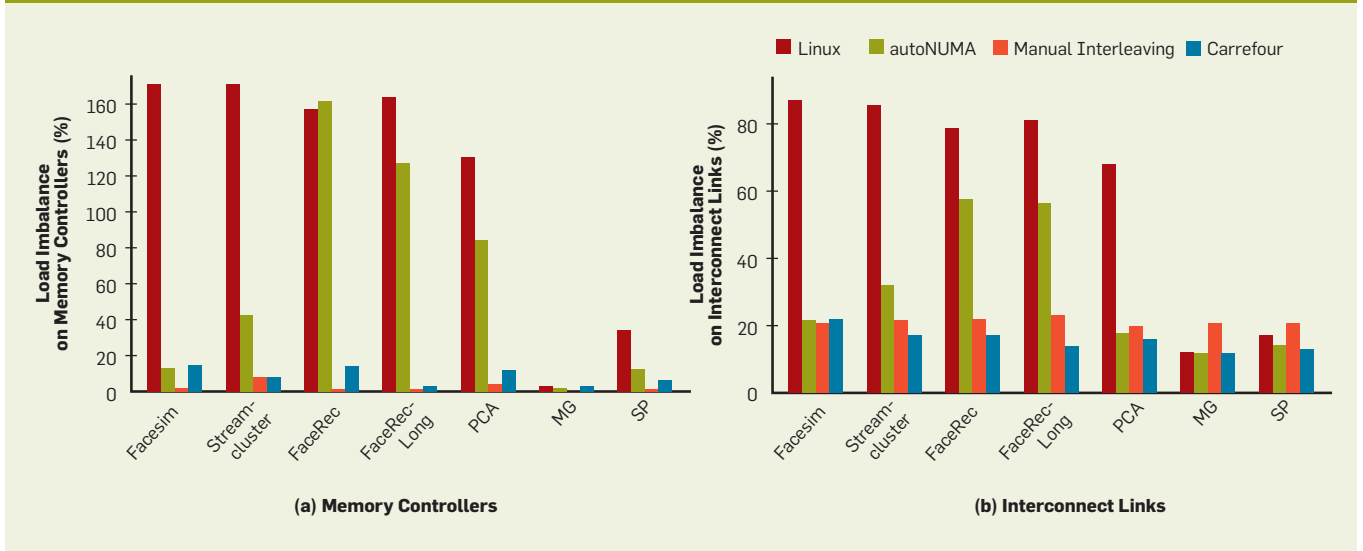


Figure 6. Load imbalance for selected benchmarks.



gave standard deviations of up to 9%.

Performance. We evaluated Carrefour's performance against Linux's default policy (first-touch), manually interleaving memory using Linux's interleave policy, and the AutoNUMA patch. Figures 4 and 5 show the performance improvement relative to default Linux.

There are three general classes of applications. First are those that have the same performance no matter which NUMA technique is used (for example, Bodytrack and Swaptions). These applications are not memory intensive and tend to have a low last-level cache miss rate. They also do not suffer much overhead from Carrefour or AutoNUMA, because most of the overhead is proportional to the memory intensiveness of the application.

The second class of applications is memory intensive, but the default first-touch policy works well for them. BT, CG, DC, FT, MG, and UA fall into this category. For these applications, manual interleaving hurts performance because it eliminates the locality benefit of first-touch without reducing congestion. On the other hand, Carrefour does not cause poor memory placement but only has a small overhead.

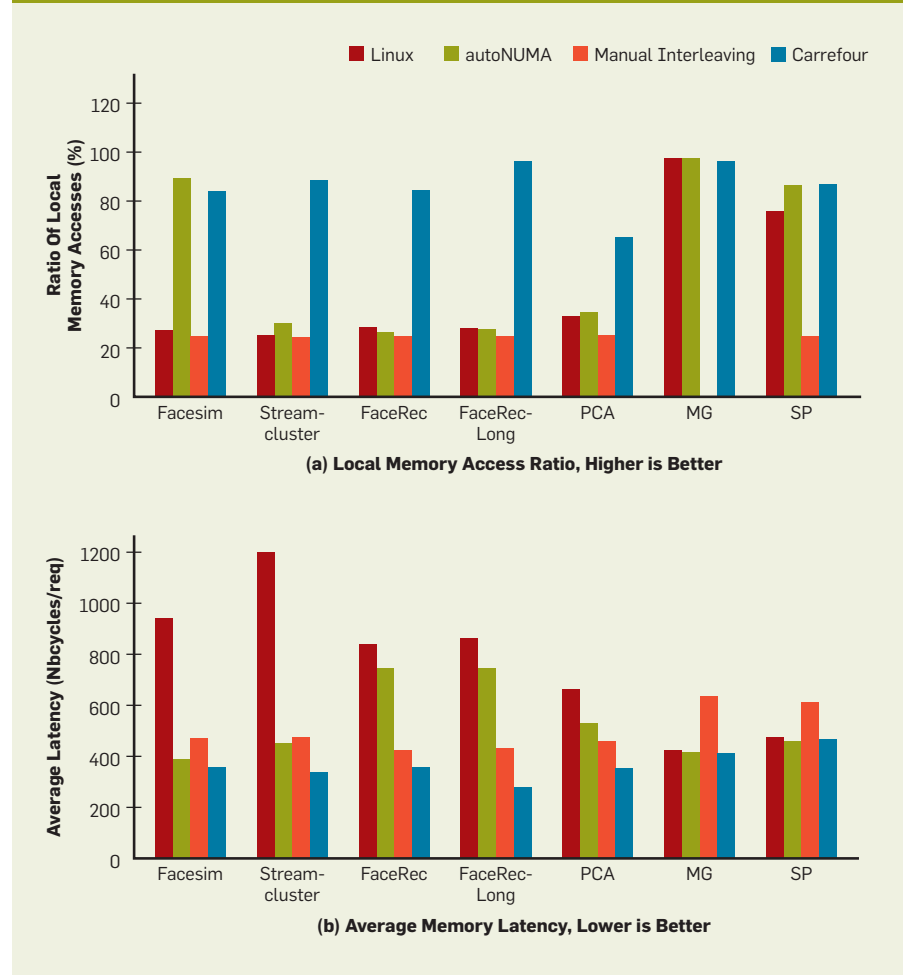
The remaining benchmarks suffer from poor memory placement under default Linux. AutoNUMA is able to improve the performance of some these applications, but for others (for example, FaceRec and PCA) it has only a small impact. Carrefour, on the other hand, significantly improves the performance of these applications, and

in two cases Carrefour greatly outperforms the second-best technique. It improves the performance of FaceRec Long by 120% over default Linux, where manual interleaving improves performance by only 60%. Similarly,

Carrefour improves the performance of Streamcluster by 180%, and manual interleaving improves it by only 100%.

One exception is IS, which is improved by manual interleaving but not by Carrefour. Carrefour's sampling

Figure 7. DRAM latency and locality for selected benchmarks.



and migration rate cannot keep up with the burst of traffic produced by IS, so the memory is not balanced in time to improve performance.

We further profiled select applications in order to see how Carrefour affects the key imbalance, locality, and latency metrics. Figures 6 and 7 present the results. Figure 6 shows the load imbalance for selected benchmarks. Lower is better.

In Figure 6a Carrefour consistently minimizes the imbalance on memory controllers, as does manual interleaving. AutoNUMA is sometimes able to reduce the imbalance but usually not to the same degree, and in the case of FaceRec it makes the imbalance worse than default Linux. The imbalance on interconnect links as depicted in Figure 6b shows similar trends.

Although manual interleaving is able to reduce imbalance, it always does so at the cost of locality. This is made evident in Figure 7a. Carrefour, on the other hand, always has the highest or nearly the highest local memory-access ratio. For applications that have good locality under default Linux (MG and SP), AutoNUMA retains the good locality but is able to improve the locality of Facesim.

The effects of imbalance and the local access ratio are reflected in the memory-access latency, shown in Figure 7b. As expected, Carrefour produces the lowest (or is tied for the lowest) average memory latencies for each profiled application. There is also a strong correlation between the average memory latency and a benchmark's performance. For example, Streamcluster and FaceRec have large reductions in memory latency with Carrefour, and they show large performance improvements in Figure 4.

Overall, we can conclude Carrefour systematically fixes NUMA memory-placement issues in nearly all situations, is able to greatly outperform other techniques in some circumstances, and does not significantly hurt performance for any application.

Conclusion

NUMA architecture is for scaling the processor count of today's server-class systems. In the near future, expect systems to have even more NUMA nodes and more complicated NUMA topologies. The experiments described here show the performance effects of NUMA

are significant and the problem is non-trivial, which motivates careful study and a comprehensive solution.


Contrary to previous NUMA studies, our experiments found congestion causes the most serious NUMA problems. Congestion happens when the rate of requests to memory controllers or the rate of traffic over interconnects is too high, which causes excessive delays for memory accesses. It can be alleviated by balancing the traffic among multiple memory controllers and interconnect links. The other factor of NUMA performance is locality, which is what previous NUMA algorithms have focused on. Good locality means that most of the memory accesses will be to the local node and therefore do not pay the latency cost of traversing interconnect links.

As shown earlier, the two NUMA concerns of congestion and locality are difficult to reconcile, and for any particular application we cannot know the best memory placement beforehand. Carrefour uses hardware performance counters and hardware sampling to determine an application's memory-access patterns online with low overhead. It then uses that knowledge to apply three page-level techniques: memory replication, memory interleaving, and memory collocation. Each technique serves a specific purpose: collocation improves locality, interleaving reduces imbalance, and replication does both in situations when reads vastly outnumber writes. The novelty of Carrefour is in combining these strategies and applying each only when appropriate.

The result is Carrefour is able to improve performance compared with default Linux for many applications. For two benchmarks, Streamcluster and FaceRecLong, performance is more than doubled by using Carrefour. Unlike manual interleaving and AutoNUMA, Carrefour never significantly degrades performance by making improper page placements.

As NUMA systems grow and the number of cores issuing memory requests increases, NUMA effects will continue being a concern. Carrefour demonstrates a collection of techniques that effectively reduce these concerns. Developers can use the methods and insights gained from Carrefour, along with the tools described earlier, to optimize their applications for NUMA systems.

Acknowledgments

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

Related articles on queue.acm.org

NUMA (Non-Uniform Memory Access): An Overview

Christoph Lameter

<http://queue.acm.org/detail.cfm?id=2513149>

Scalability Techniques for Practical Synchronization Primitives

Davidlohr Bueso

<http://queue.acm.org/detail.cfm?id=2698990>

Photoshop Scalability: Keeping It Simple

Clem Cole and Russell Williams

<http://queue.acm.org/detail.cfm?id=1858330>

References

1. Bailey, D. NAS Parallel Benchmarks. RNR Technical Report (1994); <http://www.nas.nasa.gov/publications/npb.html>.
2. Boyd-Wickizer, S. et al. Corey: An operating system for many cores. In 8th Usenix Symposium on Operating Systems and Design (2008), 43–57.
3. Brecht, T. On the importance of parallel application placement in NUMA multiprocessors. In *Proceedings of the Usenix Symposium on Experiences with Distributed and Multiprocessor Systems* (1993) 4, 1.
4. Corbet, J. AutoNUMA: The other approach to NUMA scheduling. LWN.net (2012); <http://lwn.net/Articles/488709/>.
5. CSU Face Identification Evaluation System. Evaluation of Face Recognition Algorithms. Colorado State University, 2010; <http://www.cs.colostate.edu/evalfacerec/index10.php>.
6. Dashti, M. et al. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), 381–394.
7. David, T., Guerraoui, R. and Trigonakis, V. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), 33–48.
8. Lameter, C. An overview of non-uniform memory access. *Commun. ACM* 56, 9 (2013), 59–65.
9. Metis MapReduce Library; <http://pdos.csail.mit.edu/metis/>.
10. PARSEC Benchmark Suite; <http://parsec.cs.princeton.edu/>.

Fabien Gaud is a senior software engineer at Coho Data, focusing on performance and scalability.

Baptiste Lepers is a postdoc at EPFL. His research topics include performance profiling, optimizations for NUMA systems, and multicore programming.

Justin Funston and **Mohammad Dashti** are Ph.D. students at the University of British Columbia. Funston's research interests include memory management, thread scheduling, and multicore systems. Dashti research focuses on operating systems, GPGPU, and heterogeneous CPU/GPU systems.

Alexandra Fedorova is an associate professor in the ECE department at the University of British Columbia. Her research focuses on performance, usability, and energy-efficiency of computer systems.

Vivien Quéma is a professor at Grenoble INP (ENSIMAG), France. His research is about understanding, designing, and building (distributed) systems.

Renaud Lachaize is an assistant professor at the University of Grenoble, France. His research interests are in the area of operating systems and distributed systems, with currently a particular focus on multicore systems.

Mark Roth currently works at Google as an engineer.

Copyright held by authors. Publication rights licensed to ACM. \$15.00