Research Report

Scheduling Threads and Pages on NUMA Systems

Supervisor: Prof. Dr. Alexandro José Baldassin
Author: Gustavo Leite

Departamento de Estatística, Matemática Aplicada e Computação,
Instituto de Geociências e Ciências Exatas
Universidade Estadual Paulista - Unesp
Avenida 24A, 1515, Bela Vista, Rio Claro, SP

August 16, 2018

Contents

1 Introduction

The constant effort to build systems that performs better have lead microprocessor architects to create sophisticated designs. Conventional architectures are known as Uniform Memory Access (UMA) machines. This class of systems is usually comprised of a multicore processor attached to one or more memory banks via a bus. These computers are uniform in the sense that the latency to access memory is the same no matter what processor core execute it.

Now, consider a multi socket machine where each physical processor has its own memory banks. There is an interconnect network linking all these processors which allow for a unified address space and a coherent cache subsystem. This class of computers are referred to as Non-Uniform Memory Access (NUMA) and they stand in between symmetric multiprocessing machines and clusters in terms of scale. Writing programs to run on NUMA machines requires no modification since the hardware together with the operating system offer a single continuous virtual address space. Nonetheless, references from one physical processor to another's memory may take longer, thus presenting the non-uniformity.

Under these terms, it is straightforward to suspect that the placement of processes and its memory in the topology might affect the running time of programs. More remote accesses means more time waiting for loads and stores. Therefore, if the operating system used a clever thread and page placement policy, it would be possible to attain shorter execution times.

An alternative approach would consist in writing NUMA aware programs using existing libraries [33]. The programmer should know better than anyone else how threads access memory and could manually optimize the program. The downsides of using such method include increased development time and tight coupling with specific architectures. Writing a NUMA aware programs which encompasses every possible hardware organization is tiresome and gets outdated very easily.

Some works suggest using compiler techniques to optimize programs to run on NUMA environments [32]. The problem with this approach is that the compiler does not have dynamic information from runtime to select when to migrate pages and when not to, thus, very limited optimizations can be performed.

In summary, this research report: (a) provide a quick introduction to the architecture of NUMA systems; (b) surveys current techniques for automatic NUMA balancing techniques that are transparent to the programs; and (c) envisions future work on how to optimize current approaches. In Section 2 we will cover the fundamental concepts around NUMA. Section 3 presents current techniques highlighting the good and bad parts of each. Some preliminary results from comparing some of the techniques presented can be found in Section 4. Finally, concluding remarks and projections of relevant research in this area are provided in Section **??**.

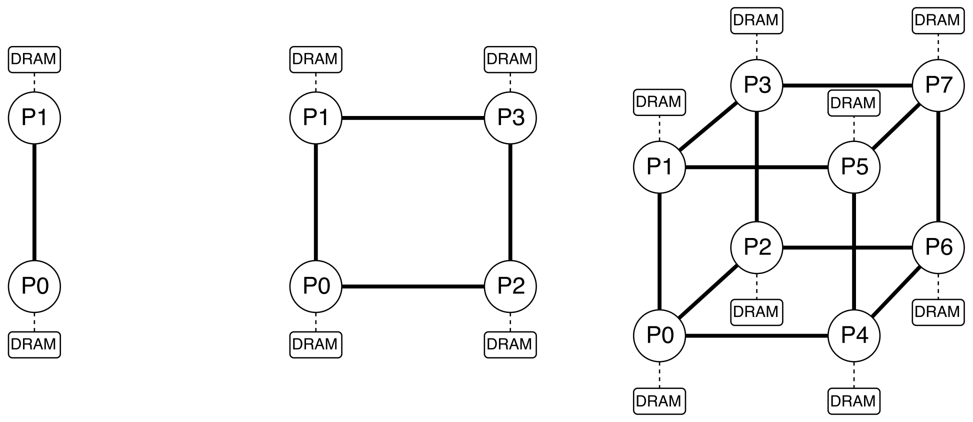
2 Foundations

NUMA machines are usually described as multi socket machines with memory physically distributed among processors. A processor coupled with its attached memory is called a **node**. The nodes are linked by a fast interconnect which forms a **topology**. Ignoring the effects of caching, references to local memory takes a uniform amount of time. Remote accesses, in turn, take longer because the request traverse the interconnect network to the destination. The actual latency can vary depending on multiple factors: interconnect traffic, memory controller bandwidth and distance in the network. The ratio of local versus remote latencies is often referred to as NUMA **factor**, in other words, the larger the NUMA factor, more costly are remote accesses. Modern

NUMA computers present lower NUMA factors when compared to older systems [20], but as we will show in subsequent sections, that are other intricacies in the problem of NUMA balancing that keeps this still a relevant research topic.

2.1 NUMA Hardware

The basic building block of NUMA machines are its nodes and the way the interconnect links all of them has a direct impact on system performance. Consider Figure 1 for instance. In this example we can see three different topologies varying on their sizes and organized in hypercube structures. This is the most common structure available since it minimize the **diameter** of the network for processors with n interconnect interfaces [18]. In a n-dimensional hypercube topology, the maximum distance between any two nodes is n and it is possible to create a mesh of up to 2^n nodes.



(a) $n = 1$ (b) $n = 2$ (c) $n = 3$

Figure 1: Hypercube topologies of various dimensions.

Ideally, we would like the interconnect to be a full mesh of connected nodes, still, limitations in the number of pins in the processor package prevents from adding arbitrarily many interfaces. Now, consider the topology in Figure 1c. If a thread executing on P0 reads a memory position mapped to P7's DRAM, the request will hop three times before arriving at the destination. Not surprisingly, the longer the distance the request travels, the longer it will take for the data to arrive back at the source.

A real example of a slightly modified hypercube topology is shown in Figure 2. It is built with four AMD® Opteron™ Bulldozer family 15h multichip modules. The machine is divided in 8 NUMA nodes, where each node has 8 x86_64 out-of-order virtual cores, summing up to 64 threads in the whole system plus 64GiB of DRAM.

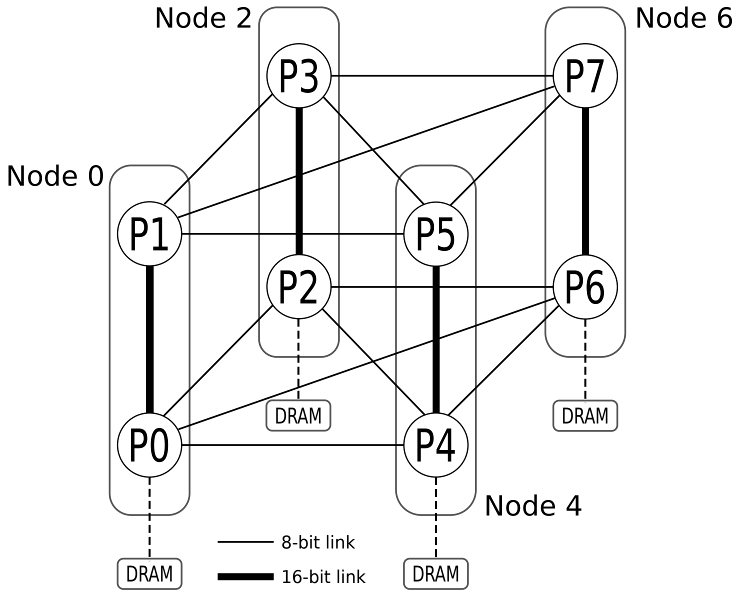


Figure 2: Topology of an AMD® machine. (Source: [17])

Observe that this hypercube topology was extended with four additional links in the upper and lower planes that reduce the network diameter from three to two at the expense of decreased bandwidth by using 8-bit links instead of 16-bits. On top of that, DRAM is connected only to the lower plane nodes, divided in 4 banks of 16 GiB each.

As mentioned earlier, the AMD® Opteron™ 6200-series processor is a multichip module that carries two Interlagos dies in the same chip. To the operating system, this is interpreted as two distinct nodes. In the topology shown in Figure 2, nodes linked by 16-bit interconnects reside in the same package, and thus are connected to a single socket. The block diagram of an Opteron™ package is presented in Figure 3.

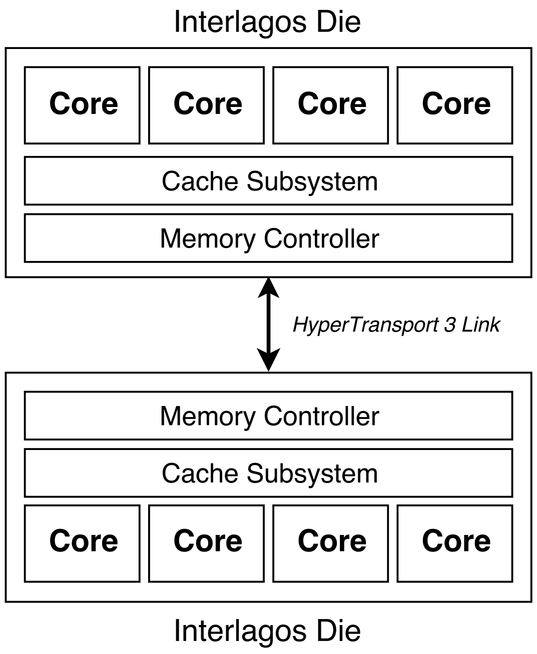


Figure 3: An AMD® Opteron™ multichip module block diagram that carries two Interlagos dies. There are a four of these in the system being described.

A single Interlagos die is packed with 4 “Bulldozer” cores that supports executing 2 threads using simultaneous multithreading [21]. The cache hierarchy for this processor is presented in Figure 4. There are 16KiB of L1 data cache per core; 64KiB of instruction cache shared between two cores; 2MiB of L2 cache shared between two cores; and 6MiB of L3 cache shared between all cores.

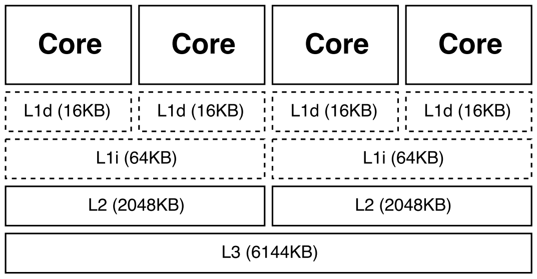


Figure 4: Cache hierarchy of the AMD® Interlagos die.

The Opteron™ processor keeps cache coherency between nodes by using a directory based [21] implementation. Computers with this characteristic are often referred to as ccNUMA (cache coherent NUMA), and in this work we limit ourselves to study coherent systems.

NUMA machines are not a novelty when it comes to hardware design. Some of the older computers with non-uniform characteristics date back to the 1980s, but the hardware as well as operating systems have changed [20]. Back then, programs were single threaded and operating systems would optimize locality, which was sufficient. Now the workloads are different, with multiple threads and larger memory footprints, which makes it hard to rely solely on locality. Moreover, in old NUMA systems remote accesses would take up to 7 times longer than local accesses [16] whereas current NUMA technology made remote accesses cost 30% compared to local accesses [5, 6].

2.1.1 The Good and the Bad

In terms of scaling, NUMA machines provide a good balance between cost and computing power. Although cheap, symmetric multiprocessing computers lack in processor cores. On the other side of the spectrum, clusters provide plenty computing capacity but at high cost. NUMA machines stand in between these two. In fact, modern servers creates clusters composed of NUMA machines.

Another great feature of this class of computers is that, even though remote access take longer, they are still faster than accessing a remote computer in a cluster through a message passing interface. Programmers can, therefore, use the simpler multithreaded programming paradigm. Finally, because memory is distributed, this means that all of the banks can be read at the same instant, resulting in larger memory throughput, benefiting memory intensive applications.

However, there are some details about how these machines are built that impact performance in a negative way and should be taken into account if we want to squeeze all the computing capacity the hardware has to offer.

The first aspect to consider are the interconnects. In the ideal case the interconnect would present no latency and infinite bandwidth. Of course, there are limitations. We discussed earlier how remote accesses are so much cheaper when compared to old systems, but they are not free of delay either. On top of the interconnect and memory latency, we should realize that bandwidth is finite and if the links are busy, requests can be delayed. The bandwidth is not only consumed by memory transfer across nodes but also to keep cache coherency. In terms of actual technology, AMD® uses the HyperTransport™3 technology [8] while Intel® uses QuickPath™Interconnect [39].

Recent research has demonstrated that interconnect links are asymmetric [26]. This means that depending on the direction the information flows in the interconnect, we could measure different speeds. Not until relatively recently, algorithm designers started to acknowledge the possibility of asymmetry and incorporate that into their models.

This is strong evidence that maximizing locality is not enough [30]. By centralizing memory, we could potentially be creating a hotspot, which takes us to the next deficiency: memory controllers. There is only many requests that can served simultaneously. If the number of requests goes over the limit, the controller will simply deny further remote accesses until older ones are finished.

Confronted with these barriers, newer balancing techniques started facing this as a traffic management problem [15, 17] instead of a purely placement one.

2.2 Hardware Sampling

In the context of NUMA balancing, using instruction sampling should provide valuable and precise contextual information to guide the placement of threads and pages throughout the system. A large volume of information can be obtained at low overhead.

When performing automatic balancing of NUMA workloads, algorithms generally track how active tasks in the operating system access memory. This tracking can be done by sampling the instructions being executed on the processor. Sampling instructions in software consists in interrupting the current task at periodic intervals and obtaining information about what the current instruction is doing, what it is accessing, etc [37]. Of course, this approach introduce a significant overhead. The other alternative is using hardware sampling.

Despite not being restricted to NUMA machines, modern CPUs implement specialized hardware which allow sampling random instructions passing through the pipeline with great accuracy and high resolution keeping the overhead minimal at the same time. This hardware is often referred to as Performance Management Unit (PMU) [35] and the data collected includes but is not restricted to: number of floating point operations; number of L1, L2 or L3 cache misses; linear and physical addresses being accessed; branch predictor outcomes; and more.

Instruction sampling is not part of the x86 instruction set architecture and thus each vendor developed its own extension. AMD® implements an extension called "Instruction Based Sampling" (IBS) [1, 19] whereas Intel® designed "Precise Event Based Sampling" (PEBS) [22]. On GNU/Linux operating systems, it is possible to query if one of the extensions is available by running the following command on a shell:

```
$ cat /proc/cpuinfo | egrep '(ibs|pebs)'
flags : ... ibs ...
```

If the extensions are in fact implemented by the processor, the command above should print one or more lines describing the processor flags which contains the key-words `ibs` or `pebs`.

IBS stores information in machine specific registers (MSRs) that can be read and written using the x86 instructions `rdmsr` and `wrmsr` respectively. This instruction takes the MSR number in the `ecx` register and returns (or reads) a 64 bit value in `edx:eax`. The specific register number are described in the "BIOS and Kernel Developer's Guide" [1]. PEBS, on the contrary, stores information on a mapped region of memory that can be read just by dereferencing a pointer.

2.3 Challenges of NUMA Balancing

In earlier sections we have already outlined that, despite some positive characteristics, modern NUMA machines still present some limitations that need to be overcome. Put in simple terms, it is needed to deal with four simultaneous goals [20]:

(a) Keep threads close to the memory pages they access;
(b) Evenly distribute the workload among nodes;
(c) Maintain memory demand below memory controllers bandwidth; and
(d) Dynamically reassign threads and pages to follow changes in the memory access pattern of the program.

A solution to this problem requires careful evaluation of the tradeoff among these four conflicting goals. First, a thread should be allocated to the node that contains the pages that the thread accesses so that it can benefit from fast local memory accesses. Second, given that typical parallel programs share data, it is often not possible to both evenly distribute the thread's workload amongst the nodes and keep all accesses local; hence the allocation algorithm has to determine which pages should stay local and which pages will be accessed remotely. Additionally, the algorithm must take into consideration the asymmetry characteristic of interconnect links in order to maximize traffic. Third, memory controllers have limited bandwidth and thus the algorithm must distribute pages in a way that does not exceed the memory throughput of each controller; if the memory demand for a specific node goes above its memory controller supported bandwidth, memory accesses are delayed thus reducing performance. Finally, programs have execution phases; memory accesses change during a program execution and thus require the migration or replication of pages and threads to avoid performance degradation.

Observe that increasing locality also increases memory controller imbalance and leaves the workload less distributed. On the other hand, assigning a similar amount of load to every node may compromise locality. The fourth goal also has its limitations because migrating memory between nodes consumes bandwidth, takes time and invalidate caches. The algorithm must pick carefully and migrate only as a last resort. When replicating pages, it is necessary to keep the contents of both pages synchronized. If a replicated page is written in one node, the page in the other node must be updated to avoid race conditions, which also has a cost. There are many tradeoffs involved and these goals must be taken into consideration simultaneously since it is not enough to reach just a subset of them.

Writing a NUMA-aware computer program is not an easy task. Although libraries [33], runtimes [7] and compiler support [32] have been proposed to assist developers, the allocation of threads and pages to nodes is complex and is typically left to the scheduler or operating system. Existing strategies attempt to increase data locality [27, 36, 38] and/or minimize congestion [4, 15, 28]. For example, Linux has a NUMA-aware scheduler that seeks to co-locate threads that share data [9, 12, 13]. However, most of these approaches do not simultaneously take into consideration the interplay between page accesses, thread and bandwidth allocation and the existing asymmetry in the links of modern NUMA computers [26]. Some users reported that a fine-grained tuning is still better than using automatic NUMA balancing and kernel developers agree that there is still work to be done [14].

2.3.1 Static Allocation Policies

Before diving into dynamic allocation and balancing policies for NUMA workloads, we will briefly describe the many static policies [25] available and why they are not sufficient [17]. The operating system offers appropriate system calls to set and get available policies, although the use of system calls directly is discouraged in favor of libraries [24]. Next, we will describe some static allocation policies.

Local Local allocation policy places a new page in the same node as the thread that issued the allocation of memory. If executing a workload of many independent single threaded programs this policy can work really well since it maximizes locality (considering that the thread is not migrated away from its data). On the other hand, when executing multithreaded applications the threads might be spread out across the system, leading to a memory hotspot in the node that holds all the pages.

Interleave This scheme places pages in a round robin fashion in all banks of memory. The advantage of using such policy is that no single node gets overwhelmed by requests, bandwidth is more balanced in the interconnect links and every memory bank holds a similar amount of data. Of course, the number of remote access will increase but pages may be read/written at the same time, increasing throughput.

Binded This is the policy used when programmers write NUMA-aware programs. With the help of libraries and the operating system [24], the developer has a fine-grained control on how to take full advantage of the topology by selecting where pages are placed. Programs in this class tend to achieve the best performances, however, writing and maintaining these applications is more problematic. Also, heavy coupling with the machine it was designed to run hinders portability.

Preferred Preferred allocation policies tries to allocate pages on a predefined node first, if it does not succeed, the OS selects another node. This scheme is similar to binding pages but more flexible. This flexibility is a desired characteristic because when using binded pages, the scheduler and memory manager must do whatever it takes to serve what the user requested. This implies that the memory manager might introduce overhead of migrating pages to accommodate everything.

First Touch When using the first-touch policy, the actual page allocation is delayed to the first time it is accessed instead of allocating at the time of request. First-touch was created based on the heuristic that threads which allocate memory (masters) are not the ones actually using the data (workers). This is the default NUMA policy on GNU/Linux operating systems.

Next Touch Next-touch policy is a variation of first-touch: the page is allocated when accessed for the first time, but when the second access happen, the page is migrated to the node that referenced it. The assumption here is that there are three groups of threads: (i) the ones that allocate data and fire other threads; (ii) the ones that read input and write to memory (first access); and (iii) the threads that actually use the data (second access).

The reader should notice that no matter how elaborate and ingenuous static policies are, they are not sufficient to ensure that the goals described in the last section will be achieved. Dynamic policies on the other hand can, in theory, make cleverer decisions by using runtime information available. Furthermore, it has been shown that the memory access patterns of programs change with time [15, 17], so even if a static policy could perform close to ideal allocation at the start of the program, that does not imply that it will be sufficient for the rest of execution.

A strong argument on the opposite direction is that designing NUMA-aware programs is sufficient and that balancing techniques will never be as performant as manually tuning the application. That might be true, to some extent. However, we shall note that some NUMA applications cannot be optimized ahead of time. As a counter example, consider a virtual machine monitor (also called Hypervisor). The hypervisor developer does not know *apriori* how the programs running inside virtual machine will behave, and thus can make no assumptions on how memory will be referenced.

As will be discussed in the subsequent sections, dynamic allocation policies can also make adjustments to the current memory layout of the process by migrating or replicating pages throughout the system. This process if often referred to as balancing.

3 Existing Techniques

This section provide an overview of current technologies of NUMA balancing. Section 3.1 presents the Linux NUMA-aware scheduler. Next, on Section 3.2, a kernel module proposed by Dashti *et al.* [15] named Carrefour is briefly introduced. Finally, Section 3.3 exhibit PTB, an integrated allocation approach proposed by de Oliveira [17], which shall be the starting point for the research project we envision to carry out next.

3.1 Linux NUMA Balancing

Due to the existence of some competing solutions [9, 11, 12], it was not until version 3.8 (released in February 18th 2013) that NUMA support was added to the Linux kernel.

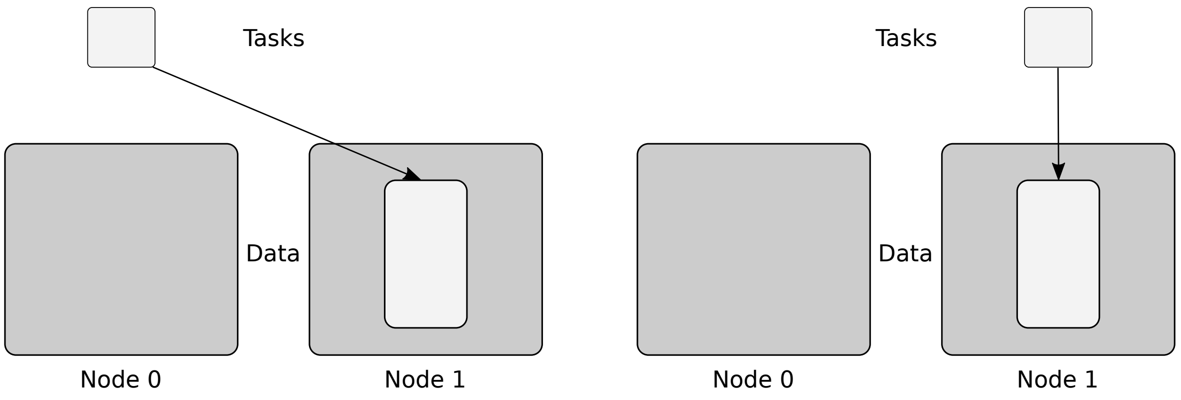
In a series of patches, Mel Gorman attempted to create a foundation to NUMA balancing which would combine the best of both proposed strategies [10]. The patch set introduces a naive policy called "*Migrate on Reference of pte_numa Node*" (MORON) which was not ideal but performed better than the vanilla kernel. Although the existence of this patch, Gorman claims that performance can still be improved by using a clever policy.

The Linux NUMA scheduler can perform three different operations in order to keep the workload balanced across the system and minimize the number of remote accesses.

Thread Balancing If a thread is performing a lot of remote acesses, it can be migrated to a suitable node where its pages reside. This policy is called CPU-follows-memory. It is often desirable to favor thread instead of page migration since a program's code is usually smaller than its data, and thus the operation can be carried faster. An illustration of thread migration in a 2-node system is shown in Figure 5.

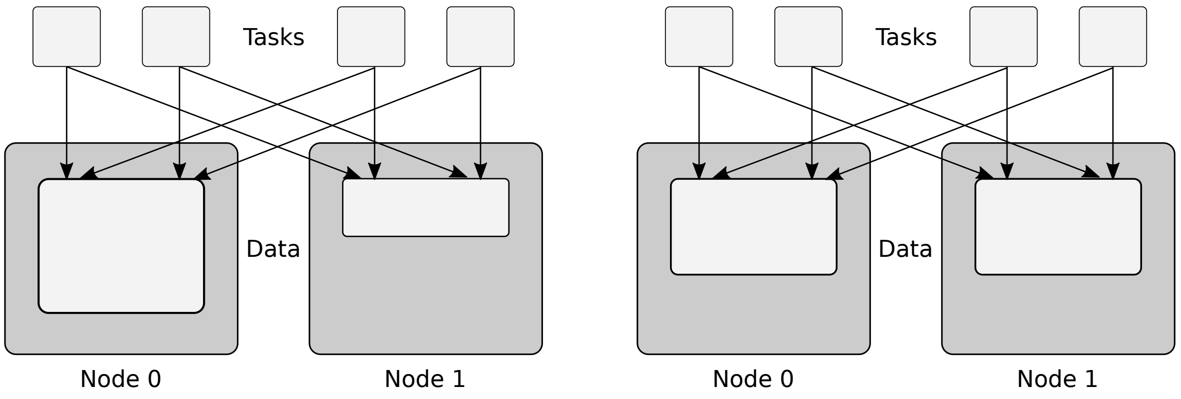
Page Balancing Migrating threads only can cause the undesirable situation where a single node is overloaded with threads while other nodes are idle. In this scenario, it makes more sense to spread out pages in the system. Also, maintaining pages in a single node can cause memory controller imbalance and bandwidth issues, degrading performance ever further. A visual representation of page migration is shown in Figure 6.

Task grouping Using the idea of NUMA groups, which is the combination of related threads and its pages, the scheduler can migrate these groups independently. As shown in Figure 7, notice that threads access mostly pages within its group denoted with different colors , and thus, placing each group in a separate node will reduce the number of remote acesses (exceptions could exist like shared libraries



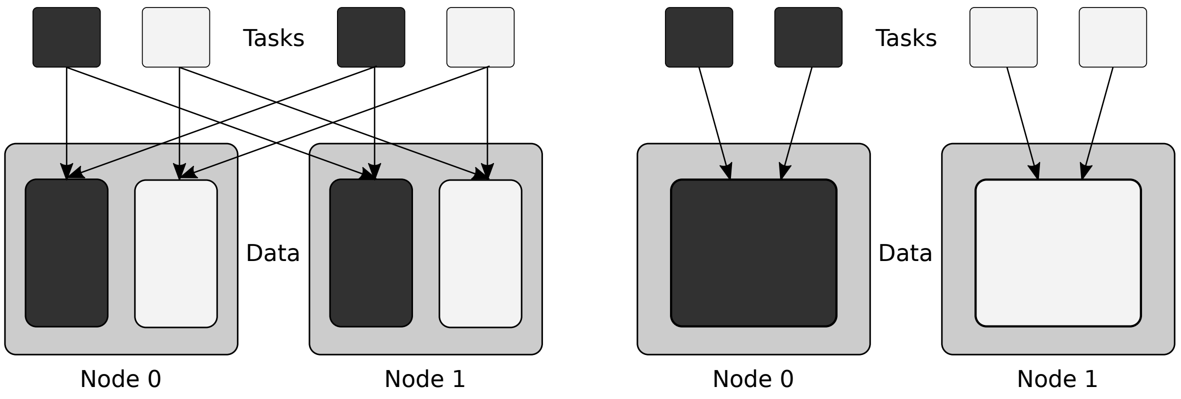
(a) Before (b) After

Figure 5: Thread migration (Source: [17])



(a) Before (b) After

Figure 6: Page migration (Source: [17])



(a) Before (b) After

Figure 7: Task grouping (Source: [17])

Table 1: Carrefour statistics

Statistic	**Description**	**Threshold**
MAPTU	Memory accesses per time unit (1us)	50
MRR	Memory read ratio	95%
MC_IMB	Memory controller imbalance	35%
LAR	Local access ratio	80%

and data which are not illustrated in the figure).

Besides not presenting good speedups, the NUMA balancing mechanism can even slowdown some applications. Furthermore, it does not take into consideration aspects like heterogeneous topologies, interconnect link and memory controller congestion [15, 34], which are all important to get the most performance out of the system.

3.2 Carrefour

Carrefour is an algorithm which envisions a holistic approach to memory placement [15]. The authors point out that, in contemporary NUMA machines, remote accesses latency are not the main concern anymore. Rather, congestion in interconnect links and memory controllers are the real barriers for applications with high demand for performance. In this sense, the algorithm works by making decisions using the current traffic statistics in the system and per-application specific data as input and ponder over enabling some built-in mechanisms like migration and replication of memory pages. The variables used in this decision-making process and their thresholds[1] are highlighted in Table 1.

The algorithm first goes through a measurement phase to collect system statistics (as detailed in Table 1) using hardware counters and instruction sampling. Second, global decisions are made on a per process basis in order to decide which mechanisms

[1]The thresholds were determined experimentally by the authors of Carrefour.

to use. Based on the information gathered it then decides what to do with each page. The global decision phase is carried out in four steps, summarized in Figure 8. The following list provide details about each step.

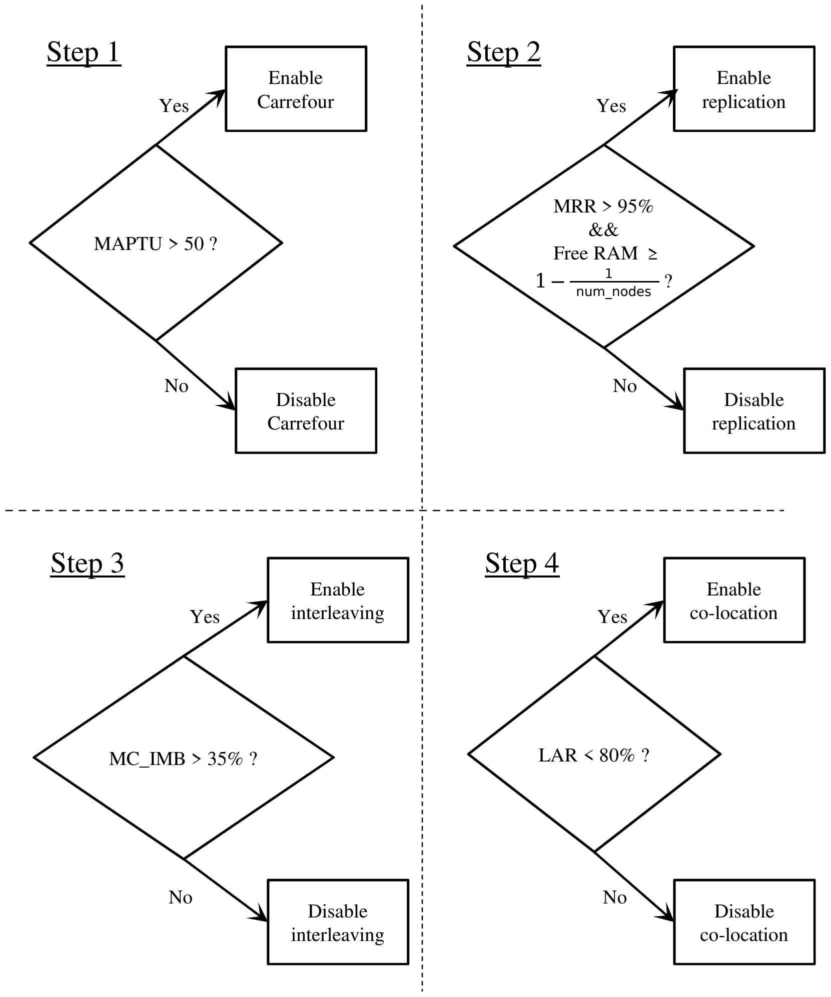


Figure 8: Global decisions on Carrefour (Adapted from: [15])

Step 1. The first decision is whether or not to activate Carrefour. If the number of DRAM accesses is low enough (≤ 50) within a time unit, the algorithm is not activated for that process. This prevents from introducing the monitoring overhead in applications that are not memory intensive and, thus, would not benefit so much by the traffic management.

Step 2. The second decision uses the proportion of memory reads versus writes and the amount of free ram in the system to choose between enabling or disabling page replication. Note that, by replicating pages, the amount of consumed RAM in the system goes up and synchronization overhead is added when the page is written. If $MRR > 95\%$ and $FreeRAM \geq 1 - \frac{1}{num\_nodes}$, this mechanism is enabled.

Step 3. If the *memory controller imbalance* is higher than 35%, the algorithm enables page interleaving. When pages are interleaved, the traffic is more evenly spread out across the system's interconnect links and memory controllers at the cost of reducing the number of local accesses. As pointed out earlier, however, maximizing locality is not top priority.

Step 4. The last decision is related to page co-location, in other words, moving pages closer to the threads that reference them. This mechanism is only enabled when the *local access ratio* is less than 80%. Otherwise the majority of accesses are already local and there is no reason to co-locate pages.

Based on the mechanisms enabled in this global phase (assuming Carrefour was in fact activated in step 1), each page will be either: (i) marked for interleaving; (ii) replicated; (iii) migrated to another node; or (iv) left as it is.

Carrefour's implementation consists of: a modified Linux kernel that enables page replication since the vanilla kernel does not provide such functionality; a kernel module where the decision procedure is implemented and the balancing is performed; and a driver in which the system administrator can interact with Carrefour and adjust its parameters during runtime. All of these items are free and open source software distributed under GNU General Public License 2.0.

3.3 PTB

PTB is an approximation algorithm that allocates **P**ages, **T**hreads and **B**andwidth to each node of a NUMA architecture [17]. It seeks to minimize memory access latency while balancing memory bandwidth usage and taking architectural asymmetry into consideration. A key insight in the design of PTB is an empirical procedure that measures actual machine parameters such as memory-controller and interconnect bandwidths so as to build an architecture performance model. PTB uses this model, and an online execution profiling, to dynamically assign threads and pages to computing nodes. This algorithm works in three steps that are executed periodically: (a) online profiling of memory access performed by threads to pages; (b) calculation of thread similarity indexes, followed by the corresponding thread migrations; and (c) memory pages redistribution based on bandwidth estimation. A description of these steps is shown below.

Online Profiling PTB's profiling approach uses hardware event sampling to track accesses by threads to pages and records this data in a hash table. The profiling phase runs for a determined period of time before continuing to the next step. The choice of using hardware event tracking instead of software approach is due to the high overhead introduced when modifying the kernel to add the possibility of building the hash table.

Thread Balancing The next step consists of balancing threads. From the hash table obtained in the last step, PTB calculates a normalized similarity index between every pair of threads running in the system. Using these indexes, a complete graph is built with nodes representing threads and edge weights representing similarities. The graph is then partitioned recursively using Kernighan-Lin's algorithm [23]. The process stops when there is at least one partition for every node. These partitions are then scheduled independently.

Page distribution Finally, pages are distributed using the algorithm devised by the original authors of PTB, which takes into consideration asymmetry in the interconnection links, estimated availability of bandwidth and memory-controller imbalance and, of course, where the accessing threads of a page are. The process then repeats from the profiling phase.

PTB was also implemented as a Linux kernel module which can be loaded and unloaded at runtime. It does not, however, require any changes to the underlying operating system at all. This characteristic is desirable since compiling and booting a modified kernel increases susceptibility to error besides slowing down development and deployment. This module was built on top of Carrefour described in the last section. PTB is also free and open source software licensed under GNU General Public License Version 2.

4 Research Plan and Preliminary Results

In this section we first present the preliminary results obtained by comparing PTB to other techniques. Second, we enumerate the next steps in this research project alongside a schedule.

4.1 Experimental results

The machine used for the following experiment is the same as described in Section 2.1, and it will be replicated here for convenience. The system is comprised by four AMD® Opteron™ Bulldozer 6272, where each physical processor packs two Interlagos dies composed of eight virtual cores using simultaneous multithreading. The system also includes 64GiB of DRAM distributed into four 16GiB banks among the processors.

Benchmark suites Parsec 3.0 [3], NPB 3.3.1 [2] and Metis MapReduce [29] were compiled using the GNU Compiler tool chain 4.4.7 and run under CentOS operating system with Linux kernel version 4.9.1. Every application on the suites were used in this experiment besides: `dedup`, `vips` and `x256` either because the execution time is too short, or the application exited with error codes. The input parameters were tuned in a way that maximizes the execution time.

In this experiment, each application was executed five times, and the average execution time was calculated. Finally, the speedup was obtained relative to the default Linux scheduler, used as our baseline. The results were condensed in the graph of Figure 9 and are ordered in ascending order relative to PTB.

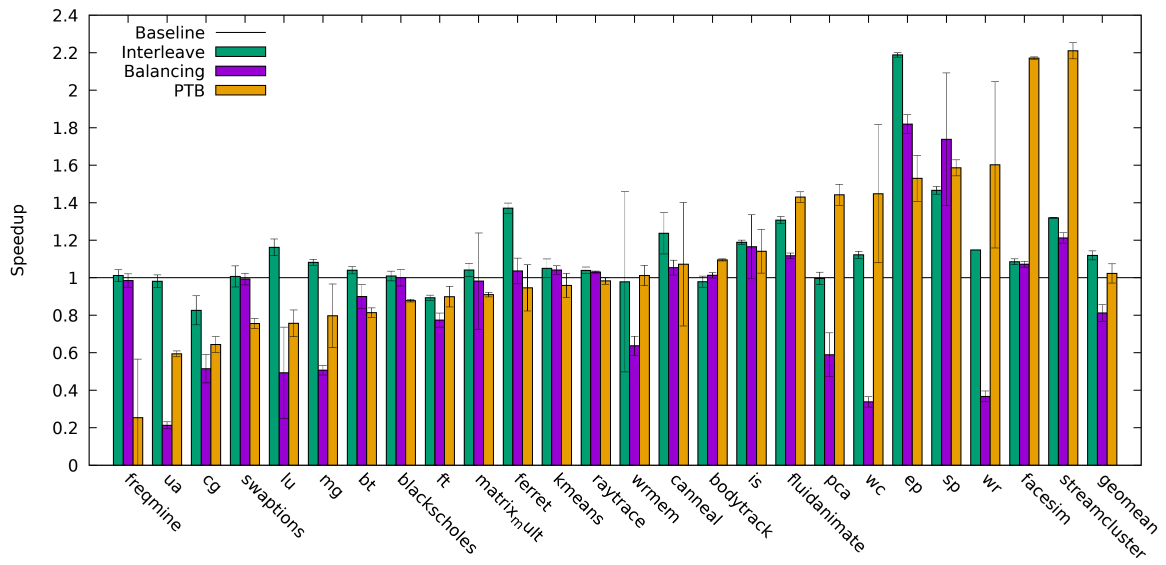


Figure 9: Comparison of Interleaving and NUMA Balancing policies to PTB. Ordered with respect to PTB speedup in ascending order. The last group is the geometric mean.

The first policy tested was interleaving, which distributes the pages of a process trough the topology in a round-robin fashion. We used the program `numactl` to enable interleaving when launching the applications. Second, the NUMA balancing feature embedded within the Linux kernel. This feature is turned off by default, to enable it we ran `sudo sysctl -w kernel.numa_balancing=1`. Third and last, PTB was compared to the previous techniques. Unfortunately, we were not able to add Carrefour to our experiments because it is not supported in this version of the Linux kernel.

By analysing the graph we can observe that the interleaving policy rarely causes slowdowns, and when it does, the execution time is not reduced by more than 20%, as is the case with `cg` application. Interleaving have shown positive results, more notably with the benchmarks `matrix_mult` (1.4x) and `ep` (2.2x). NUMA balancing, in turn, presented two outstanding speedups of 1.7x and 1.8x in applications `ep` and `sc` respectively, however, on several benchmarks, NUMA balancing presented severe slowdowns, below 0.6x (`ua`, `cg`, `lu`, `mg`, `pca`, `wc`, `wr`). PTB was able to achieve speedups beyond 2x in applications like `facesim` and `streamcluster`, however, some applications experienced massive slowdowns below 0.8x (`freqmine`, `ua`, `cg`, `swaptions`, `lu`, `mg`).

In summary, the interleaving policy performed best in this experiment as its geometric mean of 1.11x confirms. The worst result was obtained when using NUMA Balancing policy, highlighted by its geometric mean of 0.8x. With PTB, we could observe a geometric mean very close to the baseline (1.02x). This result is statistically insignificant, but at least PTB did not impact performance in a negative way.

We believe that PTB performance can be improved by directing more research efforts into the problem of NUMA balancing. While the interleaving policy is naive, it is more sophisticated compared to the default scheduler (local allocation policy). This can explain its success in this experiment. Nonetheless, interleaving is simply a static policy, and we believe that exists a dynamic approach which could better utilize

NUMA hardware.

4.2 Perspectives

In sight of all the obstacles presented in this report, it sometimes can create an impression that NUMA machines are deficient and should not be used. Quite the opposite, this class of computers is full of advantages, from the large number of computing elements to the high memory throughput. The point we tried to convey with this research is that they can be even better when used to their full potential. We still believe that the topic of scheduling threads and pages on NUMA systems is not only an interesting research problem but also very important, and its results can benefit both academy and industry alike.

All things considered, we have surveyed the problems related to Non-Uniform Memory Access machines in this research report. We have also given a panorama on how the current state-of-the-art techniques try to minimize the effects of wire delay and memory controller bandwidth by looking at it as a traffic management problem.

Here we present the perspectives for future research. A list of short-sighted tasks is shown below.

1. Adapt PTB to work with the most recent Linux kernel.
2. Add support to Intel® PEBS.
3. Test new algorithms, including but not limited to, machine learning, as inspired by related research [31].
4. Thorough experimental analysis of PTB, by comparing it to Carrefour and also testing the algorithm in different machines using well-known benchmark suites [2, 3, 29].
5. Elaboration of the final essay.

Table 2: Research schedule

Task	Sep.	Oct.	Nov.	Dec.	Jan.	Feb.	Mar.
1	●	●					
2		●	●				
3			●	●	●		
4				●	●	●	
5						●	●

The following table presents a research schedule for the next few months.

The first task is already in progress, the others are planned to be executed in the near future.

References

[1] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKGD) For AMD Family 15h Processors*, January 2014.

[2] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[3] Bienia, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] Blagodurov, S., Zhuravlev, S., Fedorova, A., and Kamali, A. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 557–558.

[5] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 43–57.

[6] Brecht, T. On the importance of parallel application placement in numa multiprocessors. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4* (Berkeley, CA, USA, 1993), Sedms'93, USENIX Association, pp. 1–1.

[7] Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.-A., and Namyst, R. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming 38*, 5 (Oct 2010), 418–439.

[8] Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., and Hughes, B. Cache hierarchy and memory subsystem of the AMD opteron processor. *IEEE Micro 30*, 2 (March 2010), 16–29.

[9] Corbet, J. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/, March 2012.

[10] Corbet, J. NUMA in a hurry. https://lwn.net/Articles/524977/, November 2012.

[11] Corbet, J. A potential NUMA scheduling solution. https://lwn.net/Articles/522093/, October 2012.

[12] Corbet, J. Toward better NUMA scheduling. https://lwn.net/Articles/486858/, March 2012.

[13] Corbet, J. NUMA scheduling progress. https://lwn.net/Articles/568870/, October 2013.

[14] Corbet, J. NUMA placement problems. https://lwn.net/Articles/591995/, March 2014.

[15] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 381–394.

[16] David, T., Guerraoui, R., and Trigonakis, V. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 33–48.

[17] de Oliveira, M. I. PTB: An integrated page, thread and bandwidth allocation approach for NUMA architectures. Master's thesis, Universidade Estadual de Campinas, August 2016.

[18] Drepper, U. What every programmer should know about memory. Tech. rep., Red Hat, Inc., November 2007.

[19] Drongowski, P. J. Instruction-based sampling: a new performance analysis technique for AMD family 10h processors. Tech. rep., Advanced Micro Devices, Inc., 2007.

[20] Gaud, F., Lepers, B., Funston, J., Dashti, M., Fedorova, A., Quéma, V., Lachaize, R., and Roth, M. Challenges of memory management on modern NUMA systems. *Commun. ACM 58*, 12 (Nov. 2015), 59–66.

[21] Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2012.

[22] Intel Corporation. *Intel (R) 64 and IA-32 Architectures Software Developer's Manual. Volume 3: System Programming Guide*, December 2017.

[23] Kernighan, B. W., and Lin, S. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal 49*, 2 (1970), 291–307.

[24] Kleen, A. A NUMA API for Linux. Tech. rep., SUSE Labs, August 2004.

[25] Lameter, C. NUMA (non-uniform memory access): An overview. *Queue 11*, 7 (July 2013), 40:40–40:51.

[26] Lepers, B., Quéma, V., and Fedorova, A. Thread and memory placement on NUMA systems: Asymmetry matters. In *USENIX Annual Technical Conference (USENIX ATC 15)* (July 2015), pp. 277–289.

[27] Li, H., Tandri, S., Stumm, M., and Sevcik, K. C. Locality and loop scheduling on NUMA multiprocessors. In *Parallel Processing, 1993. ICPP 1993. International Conference on* (Aug 1993), vol. 2, pp. 140–147.

[28] Majo, Z., and Gross, T. R. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage* (New York, NY, USA, 2011), SYSTOR '11, ACM, pp. 12:1–12:10.

[29] Mao, Y., Morris, R., and Kaashoek, M. F. Optimizing mapreduce for multicore architectures. In *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep* (2010), Citeseer.

[30] McCurdy, C., and Vetter, J. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)* (March 2010), pp. 87–96.

[31] Nemirovsky, D., Arkose, T., Markovic, N., Nemirovsky, M., Unsal, O., and Cristal, A. A machine learning approach for performance prediction and scheduling on heterogeneous cpus. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (Oct. 2017), vol. 00, pp. 121–128.

[32] Piccoli, G., Santos, H. N., Rodrigues, R. E., Pousa, C., Borin, E., and Quintão Pereira, F. M. Compiler support for selective page migration in NUMA architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 369–380.

[33] Pousa Ribeiro, C., Castro, M., Méhaut, J.-F., and Carissimi, A. Improving memory affinity of geophysics applications on NUMA platforms using minas. In *High Performance Computing for Computational Science – VECPAR 2010* (Berlin, Heidelberg, 2011), J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., Springer Berlin Heidelberg, pp. 279–292.

[34] RIEL, R. V. Automatic NUMA balancing. https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf, April 2014.

[35] SELVA, M., MOREL, L., AND MARQUET, K. numap: A Portable Library For Low-Level Memory Profiling. In *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* (Samos, Greece, July 2016).

[36] SEVCIK, K. C., AND ZHOU, S. Performance benefits and limitations of large NUMA multiprocessors. *Performance Evaluation 20*, 1 (1994), 185 – 205. Performance '93.

[37] WEAVER, V. M. Advanced Hardware Profiling and Sampling (PEBS, IBS, etc.): Creating a New PAPI Sampling Interface. Tech. rep., University of Maine, August 2016.

[38] WILSON, K. M., AND AGLIETTI, B. B. Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2001), SC '01, ACM, pp. 33–33.

[39] ZIAKAS, D., BAUM, A., MADDOX, R. A., AND SAFRANEK, R. J. Intel (tm); quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects* (Aug 2010), pp. 1–6.