Scheduling Applications on NUMA Multicore Platforms with an Asymmetric Interconnect Topology

Fabien Gaud
INRIA
fabien.gaud@inria.fr

Renaud Lachaize
Grenoble University
renaud.lachaize@inria.fr

Baptiste Lepers
CNRS
baptiste.lepers@inria.fr

Gilles Muller
INRIA
gilles.muller@inria.fr

Vivien Quéma
CNRS
vivien.quema@inria.fr

Abstract

This paper builds on the observation that some currently available NUMA multicore machines are asymmetric. Indeed, some processors have a higher connectivity degree to their peers than others. We show that this asymmetry impacts the access latencies to remote memory nodes and, more surprisingly, to local memory nodes as well. According to this characterization, processors are classified into two groups: slow processors, which have high memory latencies, and fast processors, which have lower latencies.

We show that existing schedulers, such as the Linux one, do not take this parameter into account and lead to suboptimal application placement. In order to mitigate this issue, we propose the MAPI metric (number of main memory accesses per retired instruction), which estimates the performance impact of the asymmetric topology on applications. Then, we evaluate the practical relevance of this indicator on the PARSEC 2 benchmark suite. Finally, we propose a new memory topology aware scheduler, OAAS, which leverages the MAPI metric and the physical page migration facilities of NUMA-aware OS kernels. On a 16-core NUMA machine, we show that OAAS performs within 3% of the best empirically-determined setting on various workloads taken from the PARSEC 2 benchmark suite.

1. Introduction

Multicore machines with Non-Uniform Memory Accesses (NUMA) are becoming commodity platforms. Efficiently exploiting their resources remains an open research problem. Most of the body of existing work focuses on increasing locality between computations and memory or I/O resources. This is achieved by allocating data items preferably in local memory nodes [1, 18], by moving computations close to I/O devices [22] or by moving already allocated memory pages close to the applications which use them most [11, 16].

In these works, researchers always assume that all processors have equal memory performance. Even recent works regarding task scheduling on machines with heterogeneous cores make this assumption [24, 25]. In this paper, we highlight that this assumption is not always valid. More precisely, we study a NUMA machine that exhibits an irregular connectivity between processors. Some processors are directly connected to all other processors and access memory nodes with a low latency. Other processors have a lower degree of connectivity and need more hops to access some memory nodes and access memory with a higher latency. This setup is not unique. For instance, Baumann et al. [4] describe a 32-core NUMA machine in which some processors require at most 3 hops to access remote memory while others require at most 2 hops. Using microbenchmarks, we show that such characteristics can induce a noticeable asymmetry in memory performance. While current operating systems are not aware of such performance characteristics, we show that the completion time of applications taken from the PARSEC 2 benchmark suite can vary by up to 15% depending on the processor they are scheduled on. We argue that OS schedulers should take this asymmetry into account in order to make efficient decisions.

In this paper, we make the following contributions. First, we propose a methodology to quantify the impact of the interconnect topology on local and remote memory accesses latencies. Not surprisingly, we show that processors that are not directly interconnected to all other processors have a higher latency to distant memory nodes than processors interconnected to all other processors (up to 32% on our 16-core test machine). More surprisingly, we show that these processors also have a higher latency when accessing their local memory (up to 36%). This comes from the fact that the cache coherency protocol requires a higher number of messages when these processors perform memory accesses. Second, we propose a metric, MAPI (number of main Mem-

ory Accesses Per retired Instruction), to predict the impact of processor interconnect asymmetry on the performance of applications. We empirically evaluate the relevance of this metric on applications taken from the PARSEC 2 benchmark suite. We show that this metric helps estimating the performance gap between running an application on a "well-interconnected" processor and on a "weakly-interconnected" one. Third, we show that the Linux scheduler is not aware of such performance gaps. We evaluate its performance on various workloads taken from the PARSEC 2 benchmark suite and conclude that the performance obtained with the Linux scheduler can be up to 39% worse than the best possible scheduling decision. Lastly, we show that the MAPI metric can help the scheduler making efficient decisions. We build an online scheduler that dynamically assigns applications to processors depending on their MAPI metric. We show that this scheduler performs within 3% of the best possible scheduling decision.

The rest of the paper is organized as follows. Section 2 characterizes the impact of the processor interconnect topology on memory latencies. Section 3 quantifies the impact of the interconnect asymmetry on the performance of the PARSEC 2 benchmark suite. Section 4 evaluates the performance of multiple scheduling policies on several PARSEC 2 workloads. Section 5 presents a new scheduler which leverages the interconnection topology to optimize scheduling decisions. Finally Section 6 presents the related work and Section 7 concludes the paper.

2. Hardware characterization

In this section, we first present the architecture of the hardware we are using. In our architecture, processors are not fully interconnected, i.e. some processors do not have direct connections to all other processors. More precisely, some processors require at most one hop to access memory, while others need up to two hops. We then show, using a microbenchmark, that this asymmetric processor interconnection highly impacts the latency of memory accesses. Processors that are not directly connected to all other processors have a significantly higher latency to both local and distant memory nodes.

2.1 Hardware architecture

Our experiments are performed on a Dell PowerEdge R905 machine. The overall architecture of the system is summarized in Figure 1[1]. This machine has four AMD Opteron 8380 processors, each hosting 4 cores, leading to a total of sixteen cores. Each core is clocked at 2.5 GHz, has private L1 and L2 caches (of 64 and 512 KB, respectively) and shares a 6 MB L3 cache with the three other cores of the same processor. The machine has 32 GB of DDR2 RAM at 667 MHz, organized in four NUMA memory nodes (one node per socket, 8 GB per node). Using a microbenchmark, we measured a peak throughput of 61 Gb/s between a memory controller and a local memory bank. Processors communicate (for I/O, memory and cache coherence probes and responses) using HyperTransport (HT) 1.0 links, with a measured peak throughput of 24 Gb/s. All processors have three HyperTransport links. Processors 1 and 2 use these three links to connect to all other processors. Processors 0 and 3 use two of these three links to connect to processors 1 and 2, and the remaining link to connect to I/O controllers. Consequently, as illustrated on Figure 1, there is no direct connection between processors 0 and 3. Messages between these two processors are routed either by processor 1 or by processor 2 (through the processor's crossbar). Local and distant memory accesses are performed by the memory controllers (MCT). MCTs are also responsible for triggering the cache coherence protocol.

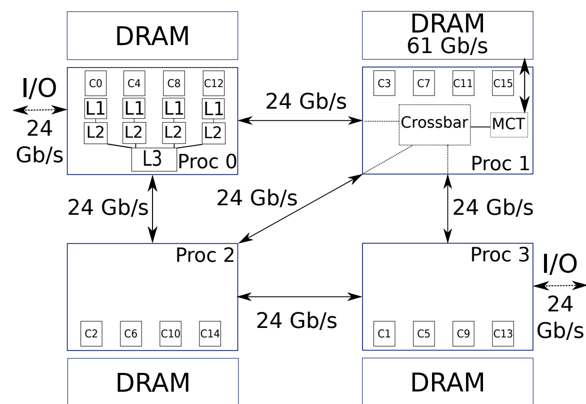

Figure 1. Interconnections between processors on the Dell PowerEdge R905 test machine.

A consequence of this HT link topology is that processors 1 and 2 need a single hop to access the memory of other processors, whereas processors 0 and 3 need between one and two hops, depending on which processor they access the memory of. This has two consequences. First, all processors do not have equal performance when accessing a distant memory node. Second, the cache coherency protocol has a different cost depending on which processor is performing memory accesses. Indeed, when a memory controller triggers the cache coherency protocol (CCP), it has to probe all the other processors to make sure that none of them has a modified version of the data in its cache[2]. As illustrated in Figure 2, when triggered by processors 1 and 2, the CCP probes all other processors in a single hop. In contrast, the CCP requires between one and two hops to probe other processors when triggered by processors 0 and 3.

[1] Note that, for the sake of readability, MCT, crossbar and caches are only represented on one processor, but are present in all processors.

[2] More precisely, the CCP is triggered in the two following cases: (i) when a read or write access is performed and there is no valid copy of the data item in the local processor caches, and also (ii) when a write access is performed on a shared cache line [2].

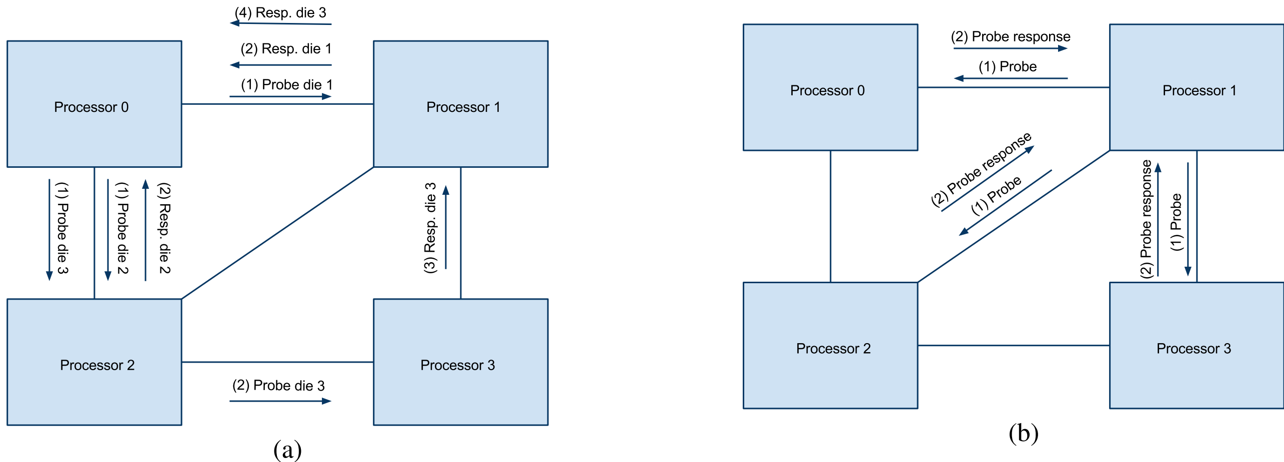

Figure 2. Cache coherency protocol messages when triggered by a read access from (a) processor 0 or (b) processor 1. For the sake of clarity, acknowledgments required by the HyperTransport protocol are not depicted in this figure. (Note that a very similar sequence of steps occurs in the case of a write access).

2.2 Impact of asymmetric connectivity on memory latencies

Using microbenchmarks [9], we evaluate the impact of the asymmetric connectivity on the latency of read memory accesses. We study two different cases: local and distant accesses. For local memory accesses, the benchmarked core reads memory words that are located in its local memory node and not already present in its cache. Regarding distant memory accesses, the benchmarked core reads memory words located in the farthest memory node: one hop away for processors 1 and 2 and two hops away for processors 0 and 3.

Requesting processor	Local memory read latency	Farthest memory read latency
0	256	377
1	193	287
2	194	290
3	263	376

Table 1. Memory read latency (in clock cycles) depending on the memory node hosting the data item and the requesting processor.

We run each experiment 10 times and observed a very low deviation (below 1%). Averaged results are presented Table 1. We can make two observations. First, reads performed on the farthest memory node have a higher latency on processors 0 and 3 than on processors 1 and 2 (+32%). In addition, we observe that even local read latencies have a higher latency on processors 0 and 3 than on processors 1 and 2 (+36%). This is a result of the extra hop needed by the cache coherency protocol when triggered by processors 0 and 3. Finally, let us note that we observed similar phenomenons for write accesses.

In the remainder of this paper, processors 0 and 3 will be named *slow processors*, while processors 1 and 2 will be referred to as *fast processors*. We will also refer to the processor interconnect asymmetry as *memory asymmetry*.

3. Impact of memory asymmetry on application performance

In the previous section, we have shown that the memory asymmetry impacts the latency of both distant and local memory accesses. In this section, using a microbenchmark, we first show that, depending on the frequency of their memory accesses, applications are heterogeneously impacted by the memory asymmetry. We then present a metric that can be used to quantify the impact of memory asymmetry on the performance of applications. Finally, we show that this metric can accurately estimate the impact of memory asymmetry on the performance of real applications from the PARSEC 2 benchmark suite.

3.1 Quantifying the performance difference between fast and slow processors

In this section, we evaluate the impact of memory access frequency on the memory access throughput difference between fast and slow processors. For this purpose, we create a microbenchmark that initially performs CPU-intensive calculations (i.e. multiplication and modulo operations on registers). Then, we run this microbenchmark multiple times, increasingly interleaving memory read operations between calculations. These read operations are performed on a 64 MB array. Besides, this array does not fit into the processor caches. The next memory address to read is given by the result of the previous read operation. The array is initially filled with random memory addresses thus ensuring that the compiler does not optimize memory reads. These settings ensure that a large majority of read operations reach the main

memory. For each core, the accessed array is private and allocated in the local memory node. For each run, we measure the aggregate throughput (i.e. the number of memory read operations done per second) that can be achieved by all cores on a fast processor and compare it to the aggregate memory read throughput that can be achieved by all cores on a slow processor. Measurements are performed ten times with a very low observed variation (less than 1%).

Figure 3 presents the throughput improvement of fast processors over slow processors depending on the frequency of memory accesses. More precisely, we express this frequency as the *the number of memory accesses performed per retired instruction (MAPI)*[3]. These results show that there is a correlation between the MAPI indicator of an application and the performance gap between processors. When the number of memory accesses per instruction grows, the performance difference between processors increases as well. Initially, when the benchmark is doing only computationally intensive work, fast and slow processors have a similar memory read throughput. In contrast, as the number of memory accesses per instruction becomes higher, the performance difference between fast and slow processors grows and stabilizes near 31% for a MAPI of 0.35 and beyond.

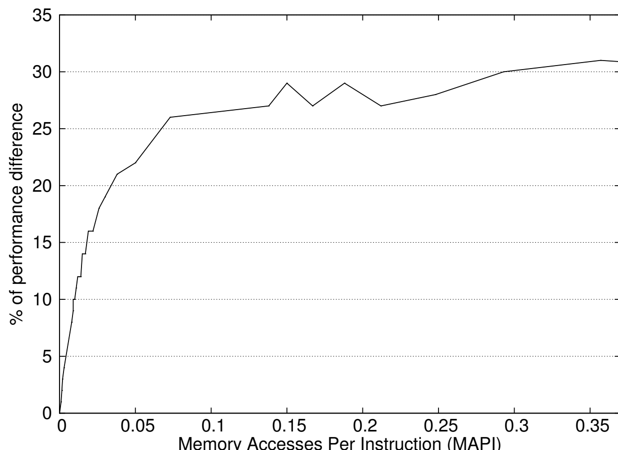

Figure 3. Throughput improvement of fast processors over slow processors depending on the number of memory accesses per instruction, in the case of the read microbenchmark.

As a summary, the MAPI metric is a good candidate to estimate the performance gap of an application between processors. Indeed, CPU-intensive applications, or applications that work mainly on an in-cache dataset, have a low MAPI and are less likely to be impacted by the topology of memory links than memory intensive applications, which have a higher MAPI.

[3] The number of memory accesses performed by an application as well as the number of instructions retired are obtained using performance counters available on recent processors. More precisely, we use the `CPU to DRAM` counter, which gives the number of memory accesses performed by a processor to a given memory node, and the `RETIRED INSTRUCTIONS` counter, which gives the number of instructions effectively issued [2].

3.2 Evaluating the impact of memory asymmetry on real applications

To evaluate the relevance of the previous results, we measure the MAPI indicator of the PARSEC 2 benchmark suite [5]. This suite is composed of representative parallel applications from diverse domains such as media processing, data mining and animation. We configured all datasets and inputs for applications to complete their work in 10 seconds when scheduled on a fast processor. All datasets fit in a memory node and are prefetched during a preliminary run in order to avoid disk accesses. We configure each application to use four threads (one thread per core on a given processor). These choices allow us to precisely isolate the impact of a given processor category (fast/slow) on application performance.

For each application, we evaluate its performance on a fast processor and on a slow processor. The performance metric that we consider is the application completion time. Such a metric has already been used in similar works (e.g. [14, 28]). We profile all the applications to get their average number of memory accesses per retired instruction (*MAPI*). Experiments are run ten times and we observe a negligible variation between runs (less than 1%).

Figure 4 presents the performance improvement of fast processors over slow ones for each application. In addition, it also displays the performance improvements obtained in the case of the microbenchmark described in section 3.1. Note that, for the sake of clarity, we use a smaller scale (for both axes) than in Figure 3. We can make several observations. First, out of the 13 applications of the PARSEC 2 benchmark suite, 4 are highly sensitive to memory asymmetry. `Streamcluster, canneal, ferret` and `facesim` perform between 5% and 14% faster when they are executed on a fast processors rather than on a slow processor. The other 9 applications are not significantly sensitive to the processor placement. Second, the higher the ratio of memory accesses per instruction, the higher the performance gap between a fast and a slow processor. Third, the observed values are close to the estimations made previously in Section 3.1 using the microbenchmark. Overall, these results confirm the relevance of the MAPI metric for realistic use cases.

4. Evaluating scheduling policies

In the previous section, we have presented a metric that can be used to classify applications and to estimate their sensitivity to memory asymmetry. However, current operating systems ignore the impact of memory asymmetry. When scheduling multiple applications at the same time, current schedulers are thus unlikely to make good placement decisions. In this section, we first consider the case of a single application. In a second part, we consider the more general problem of application co-scheduling: we look for the best

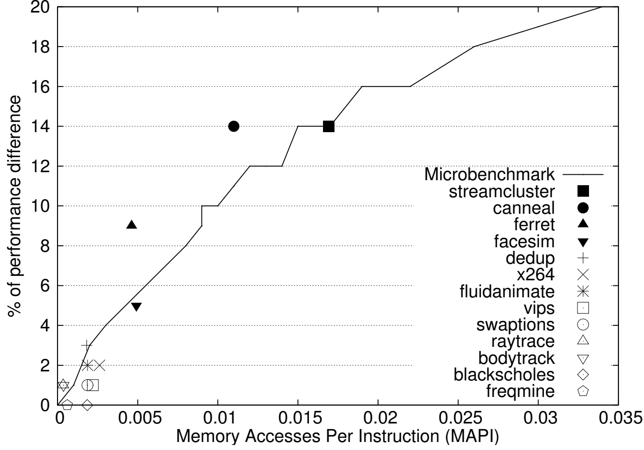

Figure 4. Performance difference between fast and slow processors on PARSEC 2 applications as compared to the previous results obtained with the microbenchmark.

placement of several workloads consisting of a mix of applications taken from the PARSEC 2 benchmark suite. In all cases, we compare the obtained performance to that of the default Linux scheduler.

We make two simplifying assumptions regarding scheduling. First, we do not take into account time multiplexing strategies and we only consider space multiplexing strategies. This choice helps focusing on the asymmetry issue. Moreover, one may argue that, in future manycore systems, space multiplexing will have an increasingly dominant weight over time multiplexing [10]. Second, we consider that an application has at most as many threads as the number of cores on a processor (i.e., a maximum of four threads in our setup). In this way, the concerns related to the affinities between threads [26] may be left aside by assigning one processor to a given application. We check that it is the best scheduling decision in Section 4.1. Consequently, on our 16-core (quad-processor) test machine, we consider co-scheduling groups of four applications with four threads in each.

4.1 Single-application scheduling policies

The purpose of this section is to evaluate whether it is better to assign a processor to a given application or to distribute its threads among processors. In this purpose, for each application, we evaluate three different strategies to assign threads to cores. The first strategy (S1) assigns all threads to a fast processor (i.e. one thread per core of the processor). This strategy may give the best results if the application is sensitive to the memory asymmetry. The second strategy (S2) assigns all threads to a slow processor. If the application is insensitive to the memory asymmetry, this strategy may give similar results to the first strategy. The third strategy (S3) assigns one thread to a core of each processor. This strategy may yield poor results if the application threads share data, because they cannot benefit from a shared L3 cache. Note that we observe that the Linux scheduler usually falls down to the third strategy.

Table 2 presents, for each application, the performance gap between the first strategy and the two others. We can make several observations. First, S1 almost always yields better performance than S2 and S3 and, in the worst case, is never inferior to them. Second, none of the 13 applications benefits from being distributed across processors. As previously explained, the reason is both the impact of the memory asymmetry and the lack of a shared cache between processors. Distinguishing the relative impact of these two reasons is difficult since an increase in cache misses implies an increase in main memory accesses and thus a higher impact of memory latencies. To evaluate the impact of the shared L3 on application performance, we measured the L3 miss rate of each application with S1 and S3. For example, in the case of `streamcluster`, the L3 miss rate grows from 12% with S1 (i.e. when the application is running on a single processor) to 47% with S3 (i.e. when it is running on all processors). We conclude that *streamcluster* highly benefits from the shared cache between cores. As a summary, under our assumptions, it is always more efficient to schedule an application on a single processor (either fast or slow).

Application	S1 over S2	S1 over S3
Canneal	+15%	+8%
Streamcluster	+14%	+39%
Ferret	+10%	+11%
Facesim	+5%	+5%
Dedup	+4%	+6%
x264	+3%	+5%
Fluidanimate	+2%	+0%
Swaptions	+1%	+4%
Blackscholes	+1%	+2%
Raytrace	+1%	+2%
Vips	+1%	+2%
Bodytrack	+0%	+5%
Freqmine	+0%	+0%

Table 2. Performance comparison (higher is better) of the three placement strategies on each of the PARSEC 2 applications: S1 (all threads on a fast processor), S2 (all threads on a slow processor) and S3 (one thread per processor).

4.2 Application co-scheduling policies

We now consider the case of multiple applications jointly running on the same machine. Because of the very high number of co-scheduling possibilities of PARSEC 2 applications, we choose to only benchmark five representative groups of four applications. We name these groups PG-X, where X represents the number of asymmetry sensitive applications in the group. The applications chosen in each PG-X group are presented in Table 3. Bold font is used to emphasize asymmetry sensitive applications. For example,

Group	Applications
PG-0	(1) bodytrack, (2) blackscholes, (3) swaptions (4)vips
PG-1	(1) bodytrack, (2) blackscholes, (3) swaptions **(4) canneal**
PG-2	(1) bodytrack, (2) blackscholes **(3) streamcluster**, **(4) canneal**
PG-3	(1) bodytrack **(2) ferret, (3) streamcluster, (4) canneal**
PG-4	**(1) facesim, (2) ferret (3) streamcluster, (4) canneal**

Table 3. Groups of applications used for evaluating scheduling policies. Each application in the group is numbered.

Name	App1 mapping	App2 mapping	App3 mapping	App4 mapping
M0	fast	fast	slow	slow
M1	fast	slow	fast	slow
M2	fast	slow	slow	fast
M3	slow	fast	fast	slow
M4	slow	fast	slow	fast
M5	slow	slow	fast	fast

Table 4. Possible co-scheduling strategies when running four applications on a quad-socket architecture with two fast and two slow processors.

PG-2 is composed of two asymmetry sensitive applications (`streamcluster` and `canneal`) and two applications that are not sensitive to memory asymmetry (`bodytrack` and `blackscholes`). This group of applications is the most relevant case for informed scheduling decisions since it is possible to assign sensitive applications to fast processors and insensitive applications to slow processors.

Given our previously described assumptions, there are six co-scheduling possibilities within a group, which are presented in Table 4. We chose the completion time of an application as the main performance metric. This metric has already been used by others [14, 28] and is relevant in the context of the PARSEC 2 benchmark suite since all applications work on a predefined input (none awaits inputs from the user or the network). We evaluate the performance of these co-scheduling possibilities (also referred to as *mappings*) as well as the performance achieved by the Linux scheduler. Lastly, we measure the performance achieved by a random policy. This policy consists in randomly choosing the processor assignment of an application. Consequently, each application has the same probability to be set on a fast or on a slow processor. The random policy is used to measure the average performance of an application when no processor is preferred. Experiments are run ten times. All static scheduling policies presented in Table 4 have a very low standard deviation (less than 1%). In contrast, and not surprisingly, the random policy exhibits a higher standard deviation (6%). The Linux scheduler also yields a high standard deviation (10%), mostly due to the aggressiveness of its load balancing mechanism. This variance has also been reported by others in a similar context [28]. For each scheduling policy and each application, we compare the obtained performance to the performance of this application when run on a fast processor[4].

Results of groups PG-1 to PG-4 are presented in Figure 5. For each of these groups of applications, we present the results obtained with the best, the worst and a random scheduling policy as well as the results measured with the Linux scheduler. Experiments named OAAS and OAAS-noPM are discussed in Section 5. The PG-0 group is not presented in this figure because the measurements simply confirm that it is insensitive to the chosen scheduling policy. This was an expected result since this group is only composed of applications that are not sensitive to memory performance.

We can make several observations. First, the impact of asymmetry on application performance in the case of co-scheduling is, overall, close to the impact observed for standalone execution (Section 3.2). Second, the best scheduling policy always consists in assigning the most asymmetry sensitive applications to fast processors. Such a scheduling policy can significantly improve the performance of asymmetry sensitive applications. For example, on the PG-1 applications group, three policies (M2, M4 and M5) achieve the highest performance because they place `canneal`, the only memory sensitive application, on a fast processor. On the PG-2 application group, only one scheduling policy is able to choose the best assignment (M5). Other policies lead to a performance degradation on either `canneal` (by up to -12%) or `streamcluster` (by up to -10%) or both of them. On PG-3 and PG-4, there is no ideal scheduling policy that does not degrade the performance of any application. In this case, the best solution is to limit the performance degration of `streamcluster` and `canneal`. Indeed, these applications are far more impacted by memory asymmetry than `facesim` and `ferret`. Third, the Linux scheduler always performs poorly with the `streamcluster, canneal, ferret` and `facesim` applications. As explained in Section 4.1, the Linux scheduler assigns the threads of an application to different processors. Thus, besides ignoring the interconnection topology, Linux also disregards the potential impact of a shared cache between threads. As discussed in Section 4.1, precisely quantifying the impact of each of these characteristics on performance is difficult. Lastly, the random algorithm also performs poorly for sensitive applications.

[4] This strategy is the best one for each application, as shown in Table 2.

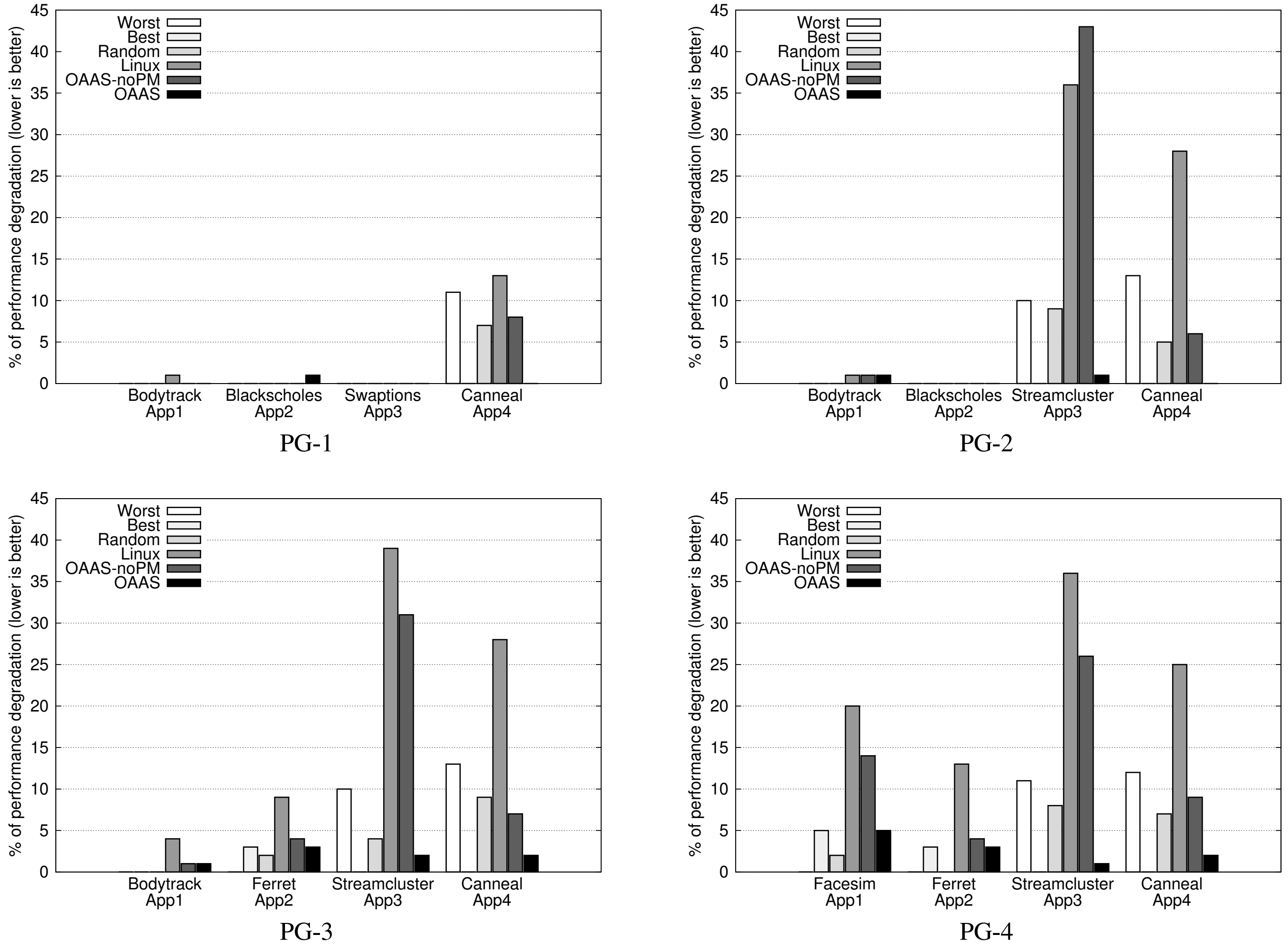

Figure 5. Performance comparison of different scheduling strategies on various groups of applications (lower is better). For each application, its performance is compared to the performance of this application when run on a fast processor.

5. The OAAS scheduler

As discussed in the previous section, the best scheduling strategy is to assign asymmetry sensitive applications to fast processors. In this section, we build an Online Asymmetry Aware Scheduler (OAAS) which leverages the MAPI metric to make scheduling decisions. We show that this scheduler is able to obtain performance within 3% of the best co-scheduling policy determined in the previous section.

5.1 Principles

The OAAS scheduler tries to schedule applications that are the most impacted by memory asymmetry on fast processors while keeping other applications on slow processors. Application sensitivity to memory asymmetry is given by the MAPI metric. We have shown in Sections 3.2 and 4 that this metric allows classifying applications. Thus, the scheduler periodically obtains the MAPI indicator for each application and sorts applications based on this metric. Applications with the highest MAPI indicator are assigned to fast processors while the other applications are assigned to slow processors.

The scheduler tries to make sure that all migrations costs will be amortized over time. Indeed, migrating an application implies that the application cannot benefit anymore from previously cached data and all memory pages allocated from a local node will be seen as distant pages after the migration. However, thanks to the results obtained in Section 3.1, the scheduler is able to estimate the performance gain of a migration. Lets P_A be the performance gap of an application A between a fast and a slow processor. If the scheduler decides to swap the location of two applications, A running on a fast processor and B running on a slow processor, then the expected performance difference on the whole system is :

$$P_{diff} = P_B - P_A$$

Using this performance gain indicator, the scheduler is able to determine whether a process migration will be amortized or not. Interestingly, this optimization also prevents two ap-

plications with approximately the same memory behavior from swapping processors too often. Indeed, in such a case, both applications will roughly have the same MAPI. As a consequence, P_{diff} will be close to zero and no migration will be triggered.

The scheduling approach described thus far moves a computation from one processor to another but not the processed data. This behavior may not be a significant problem if the application manipulates short-lived data items (i.e. if it frequently allocates and frees memory regions), because most data will quickly be allocated on the new memory node of the application. In contrast, if the application heavily manipulates long-lived data, moving this application may produce too many remote memory accesses that will decrease the performance of the application (because even on slow processors, local accesses are always faster than distant memory accesses). To circumvent this issue, the OAAS scheduler migrates applications with their memory pages. Note that the Linux scheduler does not perform memory migration, presumably because it moves processes frequently and cannot ensure that memory migration costs can be amortized.

5.2 Implementation

The OAAS scheduler is implemented in user-level on top of the Linux 2.6.35 kernel. OAAS makes use of the `Linux performance counter interface` to get the required information about the number of executed instructions as well as the number of memory accesses performed. Every 300 milliseconds, OAAS checks the MAPI of all applications' threads. Then, applications are sorted using these values. The scheduler only migrates applications if the expected benefit exceeds a predefined threshold (5%). This value has been determined experimentally, as it provides the best results on various workloads taken from the PARSEC 2 benchmark suite. The expected P_{diff} is evaluated using the MAPI indicators of the two applications. The scheduler interpolates the performance gaps measured on the microbenchmark presented in Section 3.1 and illustrated in Figure 3 to create a function of MAPI to performance gap. In the current implementation, this function is automatically generated by running the microbenchmark at system startup. If the scheduler decides to move an application to another processor, the effective relocation operation is performed with the `sched_setaffinity()` system call. All threads owned by the application are moved to the newly affected processor. The migration of memory pages is done using the `numa_migrate_pages()` system call. As mentioned earlier, the scheduler prevents applications from moving between processors of the same category (fast or slow) because this kind of migration only introduces extra overhead.

5.3 Evaluation

We evaluate the OAAS scheduler with and without page migration on the five groups of applications presented in Table 3. The goal of this evaluation is twofold. First, it allows comparing the OAAS scheduler to a static processor assignment, to a random processor assignment and to the default Linux scheduler. Second, it allows evaluating the impact of page migration on the performance of applications.

Each experiment is run ten times. For these experiments, applications are initially placed randomly on processors, then the OAAS scheduler periodically applies mapping decisions, as explained in Section 5.1. Results of the OAAS scheduler are presented in Figure 5. OAAS-noPM corresponds to the OAAS scheduler without page migration.

Based on these results, we can make several observations. First, the OAAS scheduler achieves very good performance. On the most favorable groups of application (PG-0, PG-1 and PG-2), OAAS has nearly the same performance as the best static assignment. This result shows that OAAS is able to efficiently classify applications based on their MAPI indicator. It also shows that OAAS has a very low overhead. On the PG-3 and PG-4 groups of applications, the performance results of OAAS are slightly bellow those of the best static assignment by up to -3% and -5% respectively. This is explained by the cost of process migrations. In PG-3 and PG-4, when the scheduler decides to swap two applications, it moves huge memory datasets (e.g. swapping `ferret` and `canneal` represents a migration of 133 kpages of 4 kB each). In contrast, memory datasets are much smaller on PG-0, PG-1 and PG-2 groups (e.g. swapping `blackscholes` and `canneal` represents a migration of 78 kpages). As a consequence, OAAS performs better on these workloads.

Second, we can see on Figure 5 that the page migration mechanism is crucial for the efficiency of OAAS. Applications always allocate data in their currently local memory node. When an application is moved from one processor to another without page migration, memory accesses performed to these data become distant accesses, have a higher access latency and degrade performance. An application is affected by the lack of page migration if (i) it spends a substantial amount of time performing memory accesses and (ii) it manipulates allocated data during long periods of time. We experimentally confirmed that it is the behavior of `streamcluster` and `canneal` which highly benefit from page migration. We used the `CPU to DRAM` performance counters to quantify the evolution of the ratio of local memory accesses before and after each migration. When `streamcluster` is scheduled on the processor it was spawned on, 99% of its memory accesses are local. After moving `streamcluster` from one processor to another, only 1% of its memory accesses are local. Indeed, `streamcluster` mostly accesses data allocated in a first initialization phase. `Canneal` has a similar memory behavior. `Ferret` and `facesim` are less impacted by page migration because (i) they are less memory intensive and (ii) they manipulate more short lived objects (after a migration, `facesim` has only a 35% ratio of remote memory accesses).

Lastly, an important property of OAAS is that it exhibits a low performance variation (less than 1%) compared to the Linux and the random scheduler. Getting the lowest possible variance is a good property for a scheduling mechanism. Indeed, scheduling mechanisms with a high variance can either have good or bad performance. To minimize the variance of OAAS, the page migration mechanism is very important. Without page migration, OAAS has a very high variance (by up to 30%). This variance is caused by the initial random processor assignment of applications. Applications initially allocate memory pages on their local memory node. When the scheduler decides of a new application placement, applications can be either moved on a new processor or left on their current processor. Consequently, initially allocated data can be located on either remote or distant memory, leading to very different memory costs. The page migration mechanism allows mitigating the effects of initial random placement by always moving data close to the applications.

6. Related work

Several research projects have focused on improving the performance of applications on multicore systems. Some papers have tackled the problem via scheduling. Our paper is closely related to the work by Fedorova et al. [24], which proposes a scheduler for asymmetric multicore processors. They define two types of processors, "slow" and "fast", based on their clock speed and hardware complexity (e.g. level of microarchitectural optimization). Their scheduler makes decisions using a *utility function* which represents the expected performance gain of switching from a "slow" core to a "fast" core. An application, that is memory intensive, will be scheduled on "slow" cores because it would not likely benefit from the extra processing power of a "fast" core. This decision is the opposite of the one made by our scheduler. Indeed, the architectural assumptions made by Fedorova et al. do not match with ours. In current NUMA-machines, there is no difference in clock speed nor hardware complexity between processors; rather the asymmetry results from the interconnect topology. The scheduling decision, on such machines, should then be to schedule memory intensive applications on fast cores, not on slow cores. However, in future multicore machines, asymmetry will probably arise from both the interconnect topology and complexity differences between cores. In such systems, a scheduler leveraging both the MAPI metric and the *utility function* defined by Fedorova et al. will likely provide the best performance.

Besides, Li et al. [16] also distinguish processors on their frequency. They try to schedule applications on the fastest cores first, while taking into account migration overheads in a NUMA system. However, their work mainly focuses on avoiding costly thread migrations and assumes that all applications can equally benefit from being scheduled on "fast" cores. We show that the memory topology impacts different applications in different ways and use this information to make scheduling decisions. The two above-mentioned papers both artificially introduced asymmetry in their system by reducing the frequency of certain cores. To the best of our knowledge, we are the first to show that some commercially available NUMA systems, advertised as SMP (Symmetric Multi-Processors), are asymmetric and that this asymmetry impacts performance.

Other papers have proposed scheduling strategies for mitigating contention for shared resources (e.g. [6, 7, 13, 15]) or for leveraging cache locality (e.g. [26]) on multicore systems. In this paper, we chose to ignore these issues because they are not directly related to memory asymmetry. For the same reason, we do not consider time multiplexing issues, which have already been abundantly discussed in previous works (e.g. [8, 18, 20, 27]). However, in a real-world scheduler, such issues should be taken into account to make efficient load balancing strategies.

Finally, some papers try to detect and prevent multicore-related issues via profiling. Pesterev et al. [23] use *Instruction Based Sampling* to detect poor cache behaviors in the Linux kernel, Memphis [19] uses similar hardware counters to detect performance problems on NUMA platforms. These works allow developers to find hotspots in their applications but neither of them correlates performance with application placement on processors. Marathe et al. [17] use hardware counters to determine the best page placement of a given application on a NUMA system. This work only focuses on memory locality without considering asymmetry between NUMA nodes of a system.

7. Concluding remarks

In this paper, we have illustrated the importance of taking into account the interconnect topology of NUMA machines to make efficient scheduling decisions. We have described a metric which allows predicting the impact of the interconnect topology on application performance and have evaluated its relevance on the PARSEC 2 benchmark suite. Our scheduler, which leverages this metric, is able to dynamically make efficient placement decisions on various workloads taken from PARSEC 2. Below, we discuss additional points concerning related and future work.

Integration in a real-life scheduler Currently, OAAS disregards time multiplexing issues and is implemented as a user-level prototype. However, we believe that the presented optimizations could easily be integrated in a full-fledged scheduler. For example, the Linux scheduler, when choosing tasks to migrate, already takes into account the cache warmness of applications [18]. The scheduler could also take into account the MAPI metric before moving tasks; sensitive applications would then execute with a higher probability on fast processors, resulting in higher performance.

Applicability of our results We believe that the memory asymmetry described in this paper is not a specificity of the

studied machine. For example, Baumann et al.[4] use a 32 core, 8 socket, NUMA machine where some processors require at most 2 hops to access memory while others require up to 3 hops. Moreaud et al. [21] use a similar 8 socket machine with only 16 cores. One may argue that recent processor optimizations, such as AMD HT Assist [3], reduce the number of cache probes and limit the impact of the interconnect topology on local memory accesses. We intend to study this aspect in our future work. However, in any case, we do not believe that these optimizations will limit the applicability of our results on future machines. Indeed, new processors designs, such as the Intel SCC [12], even introduce memory asymmetry between cores of the same processor: some cores have no direct access to memory. This kind of complex and highly asymmetric interconnect topology may become common in future processors. Taking into account the degree of connectivity of cores, for schedulers running on such architectures, will likely be of prime importance to achieve good performance.

References

[1] AMD. NUMA aware heap memory manager `http://developer.amd.com/Assets/NUMA_aware_heap_memory_manager_article_final.pdf`.

[2] AMD. BIOS and Kernel Developer's Guide For AMD Family 10h Processors (rev 3.48), 2010.

[3] AMD DeveloperCentral. HT Assist - What Is It ? `http://blogs.amd.com/developer/2009/07/21/ht-assist-what-is-it/`.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schuepbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM symposium on Operating systems principles (SOSP)*, October 2009.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[6] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[7] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[8] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[9] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, December 2008.

[10] David Wentzlaff and Charles Gruenwald III and Nathan Beckmann and Kevin Modzelewski and Adam Belay and Lamia Youseff and Jason Miller and and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. Technical Report MIT-CSAIL-TR-2010-003, MIT, 2010.

[11] Brice Goglin and Nathalie Furmento. Memory Migration on Next-Touch. In *Linux Symposium*, Montreal Canada, 2009.

[12] Intel. Single-chip Cloud Computer `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.

[13] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[14] Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. Aash: an asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 85–96, New York, NY, USA, 2010. ACM.

[15] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28:54–66, May 2008.

[16] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 53:1–53:11, New York, NY, USA, 2007. ACM.

[17] Jaydeep Marathe and Frank Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 90–99, New York, NY, USA, 2006. ACM.

[18] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, 2008.

[19] Collin McCurdy and Jeffrey S. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 87–96, March 2010.

[20] Molnar Ingo. Modular Scheduler Core and Completely Fair Scheduler [CFS] `http://lwn.net/Articles/230501/`.

[21] Stéphanie Moreaud and Brice Goglin. Impact of numa effects on high-speed networking with multi-opteron machines. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 24–29, Anaheim, CA, USA, 2007. ACTA Press.

[22] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[23] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the European conference on Computer systems (EuroSys)*, 2010.

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

[25] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[26] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2007.

[27] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 159–166, New York, NY, USA, 1989. ACM.

[28] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.