Journal Title
XX(X):1–13
©The Author(s) 0000
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Modeling memory access behavior for data mapping

Matthias Diener, Eduardo HM Cruz, Philippe OA Navaux

Abstract

Many modern parallel architectures feature a Non-Uniform Memory Access (NUMA) behavior since they contain several memory controllers. In such architectures, deciding where to place memory pages has a high influence on the performance of parallel applications. This placement of pages to NUMA nodes is called data mapping. Two basic types of data mapping policies exist, focusing on improving the locality or the balance of memory accesses. In this article, we introduce a comprehensive set of metrics to characterize the memory access behavior of parallel applications with the aim of describing their suitability for different data mapping policies. We present and evaluate different policies that improve locality, balance, or both. Experiments were performed on three highly different NUMA architectures with two parallel benchmark suites. Results show that the improvements of the policies depend on the characteristics of the architectures and applications, and that our characterization has a high accuracy regarding the behavior. We also find that a mixed policy, which takes both locality and balance into account, can lead to the highest gains overall, while avoiding the performance losses that are caused by applying simple policies that focus only on one metric.

Keywords

NUMA, data mapping, locality, balance, memory access behavior

Introduction

The memory hierarchy present in parallel architectures with a Non-Uniform Memory Access (NUMA) behavior introduces different memory access latencies to the system, depending on the core that requested the transaction and the main memory bank that contains the requested data. In current NUMA architectures, each processor in the system contains one or more memory controllers (Awasthi et al. 2010). A core can access a memory bank using its local memory controller, which we call a *local* memory access, or access a memory bank through a memory controller located in another processor, which we call a *remote* memory access. In these NUMA architectures, two aspects are important to be taken into account. First, the *locality* of the memory accesses (Verghese et al. 1996; LaRowe Jr 1991; Ribeiro et al. 2010), as local memory accesses have a higher performance than remote memory accesses. Second, the *balance* of the memory controllers (Awasthi et al. 2010; Dashti et al. 2013; Blagodurov et al. 2010), to avoid that a memory controller processes many more accesses than others.

One of the most important techniques to improve locality and balance of memory accesses is the mapping of pages to NUMA nodes considering the memory access behavior (LaRowe Jr 1991), which is called *data mapping*. To perform this data mapping, it is necessary to develop methods to analyze parallel applications. To improve locality with data mapping, it is necessary to discover which threads access each memory page of the application, in order to map the pages to the NUMA nodes of the cores running the threads that access them (Diener et al. 2014). To improve balance with data mapping, we need to know how many memory accesses are performed to each page, such that we can map the pages to NUMA nodes in a way that all memory controllers handle a similar amount of transactions (Dashti et al. 2013). Current research performs mostly a qualitative analysis (Majo and Gross 2012) of memory access behavior for data mapping, without providing a quantitative description of the behavior.

In this article, we introduce formal metrics and a methodology to analyze the page usage of parallel applications to characterize their suitability for data mapping on NUMA

Sunrise Setting Ltd, UK

Corresponding author:
Alistair Smith, Sunrise Setting Ltd Torbay Innovation Centre, Vantage Point, Long Road, Paignton, Devon, TQ4 7EJ, UK.
Email: alistair.smith@sunrise-setting.co.uk

architectures. These metrics determine application suitability for different types of data mapping policies. We analyze several data mapping policies, verifying how the different policies and memory access behaviors influence the performance of parallel applications. We also introduce a new policy, *Mixed*, that focuses on improving both locality and balance of memory accesses. In an experimental evaluation on three different NUMA machines with the NAS-OMP (Jin et al. 1999) and PARSEC (Bienia et al. 2008) benchmark suites, the Mixed policy resulted in the highest improvements overall compared to the default first-touch policy of Linux. It also avoids the performance loss that happens for some applications when applying simple mapping policies that improve only one metric.

This article is an extension to our previous work (Diener et al. 2015). We make the following main additions in this version. We extend our model of the memory access behavior to evaluate the improvements over the default data mapping policy of most operating systems, the first-touch policy. We further introduce metrics that analyze the dynamic behavior of the applications. With these extensions, we can more accurately predict the performance improvements from different data mapping policies. Furthermore, we provide a comprehensive overview of common data mapping issues and how they can be resolved using a well-known parallel application kernel. This can help developers to better understand applications' memory access behaviors. Finally, we show a brief example of data mapping policy behavior to illustrate the differences between the policies.

Related work

In the past years, data mapping has received renewed interest due to the proliferation of NUMA architectures. Several researchers have analyzed different parallel architectures and applications, verifying how data mapping influences their performance. Majo et al. (Majo and Gross 2012; Majó 2014) are examples of such research. The authors study the memory access behavior of several applications, identifying which threads access each portion of the data structures of the applications. Although the authors find interesting conclusions, most of the profiled data from the applications are qualitative, and they do not present a systematic method to analyze the parallel applications. In our work, we propose formal metrics to accurately describe the behavior of the applications. Mandal et al. (Mandal et al. 2010) use the pChase benchmark to evaluate memory access latency and bandwidth of modern NUMA systems. The measured behavior is used to build a latency and bandwidth model of the Nehalem and Opteron architectures. This model does not discuss data mapping and is not evaluated with other applications however.

Regarding data mapping policies, most traditional strategies focus on improving the locality of memory accesses. In policies such as *first-touch*, pages are not migrated during execution, losing opportunities to improve performance. There are policies that profile applications by introducing additional page faults (Corbet 2012; Diener et al. 2014; LaRowe Jr 1991) and migrate pages to the NUMA nodes that recently accessed them. Recent proposals have been addressing the issue of the balance between memory controllers. The Carrefour mechanism (Dashti et al. 2013) balances the load in memory controllers by mapping pages to NUMA nodes, gathering page usage information using hardware counters available on AMD architectures. Awashti et al. (Awasthi et al. 2010) present a page migration mechanism that uses queuing delays and row-buffer hit rates from memory controllers. These profiling mechanisms consider only a small sample of the memory accesses, which decreases the accuracy of the mapping. Furthermore, they consider only the locality or balance separately, while our work considers both.

Qualitative page usage

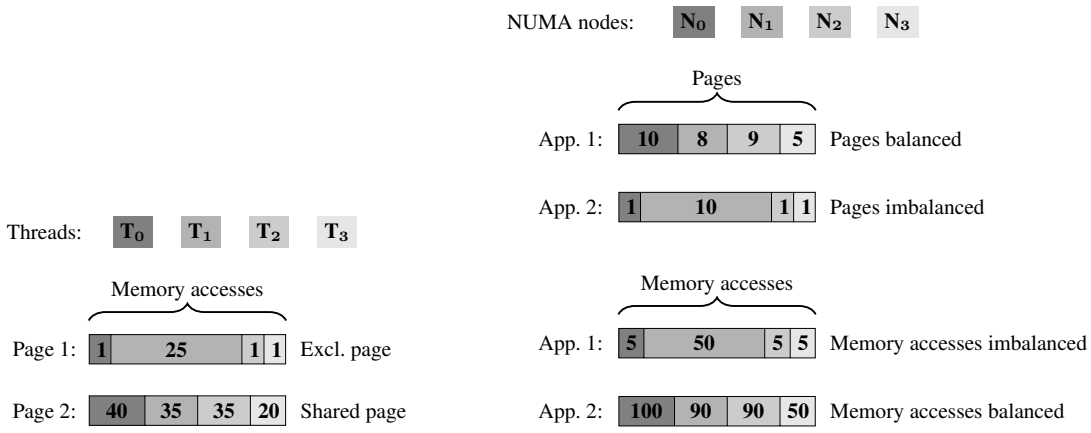
This section introduces and discusses the main concepts of the page usage of parallel applications qualitatively, focusing on how they interact with different data mapping policies.

Introduction to page usage

Since data mapping focuses on the placement of memory pages, all discussion of page usage and its metrics uses the granularity of the page size of the architecture. Page usage can be improved by considering two metrics, the *locality* and *balance* of memory accesses. For a locality-based policy, pages should be mapped to NUMA nodes from which they are most accessed. For a balance-based policy, pages should be mapped in such a way that all memory controllers handle a similar amount of memory accesses. Intuitively, pages that are accessed mostly from a single NUMA node are candidates for a locality-based policy, where these pages are mapped to the node with the most accesses to each page. For pages that are accessed in an equal way from all the nodes, a locality-based policy can usually not improve the overall memory access performance. However, these shared pages can be mapped in such a way that the number of memory accesses handled by the memory controllers is equalized with a balance-based policy, in order not to overload some of the nodes.

Example of page usage

The two main concepts of page usage, locality and balance of memory accesses, are illustrated in this section. Both



(a) Example of memory access exclusivity for an application that accesses 2 pages and consists of 4 threads.

(b) Example of memory access balance for 2 applications and 4 NUMA nodes.

Figure 1. Exclusivity and balance of memory accesses.

describe the access pattern to memory pages from threads and NUMA nodes.

Exclusivity and locality *Exclusivity* describes how many of the memory accesses to a page are caused by a single thread or a NUMA node. Pages with a high exclusivity, that is, pages that are accessed mostly from a single thread, are more suitable for locality-based policies since such a policy can reduce the memory access distance effectively. Figure 1a illustrates two pages with different exclusivities. Page 1 is mostly accessed by thread 1, with 25 of the 28 accessed performed by this thread. Therefore, this page has a high exclusivity and we call it an *exclusive page*. Page 2, on the other hand, has a much more distributed access pattern, with a similar number of accesses by all threads. This page has a low exclusivity and we refer to it as a *shared page*. Page 1 is more suitable for a locality-based mapping policy, but it has a lower overall impact on the memory accesses since it receives far fewer accesses than page 2. Since the locality of page 2 varies only slightly with the actual data mapping, it can be used to balance the memory accesses between memory controllers with a balance-based policy.

Balance While exclusivity and locality can be discussed for a single page, the memory access balance needs to be evaluated by taking into account multiple pages. Two balance metrics can be analyzed: *page balance* and *memory access balance*. Figure 1b shows examples of both metrics for two different applications. Application 1 balances the number of pages fairly among the NUMA nodes, while application 2 allocates the majority of pages on node 1. This leads to a high imbalance for application 2. However, a high page balance does not necessarily imply a high memory access balance, because the number of accesses to each page might be different. An example of this situation is shown in the figure. Although application 1 has a high page balance, its memory accesses are imbalanced. Application 2 has the opposite behavior. Since a balance-based policy focuses on the memory accesses, simple mapping policies that are based on balancing pages (such as interleave (Kleen 2004)) might not result in an optimal balance.

```
#define N 128
int A[N][N], B[N][N], C[N][N];

int main(int argc, char *argv[])
{
  int i, j, k;

  #pragma omp parallel for
  for(i=0; i<N; i++)
    for(j=0; j<N; j++)
      for(k=0; k<N; k++)
        C[i][j] += A[i][k] * B[k][j];

  return 0;
}
```

Figure 2. Matrix multiplication code.

Common types of page usage

This section discusses common types of page usages and their impact on a first-touch mapping policy. We selected a parallel matrix multiplication, since it contains examples of all types that we discuss. The source code of the application is shown in Figure 2. The application has three main data structures, `A`, `B`, and `C`, representing square matrices of size $N \times N$. After initialization, the application calculates `C` $=$ `A` $\times$ `B` in parallel.

We generated a memory access trace of the application when executing it with 4 threads, recording every memory

			Number of memory accesses per thread			
Address	Structure	First access	T_0	T_1	T_2	T_3
1537	B	T_0	17,222	9,347	8,173	9,054
1553	C	T_0	95,869	1,412	1,146	1,382
1557	C	T_0	12,544	56,598	0	0
1558	C	T_0	256	81,028	0	0
1564	C	T_0	256	0	70,835	0
1566	C	T_0	256	0	0	57,033
1572	A	T_0	131,328	0	0	0
1573	A	T_0	6,400	28,445	0	0
1577	A	T_0	256	2,594	38,912	0
1581	A	T_0	256	0	2,195	38,343
—	Stack T_0	T_0	5,381,218	1,230,331	1,017,819	1,177,382
—	Stack T_1	T_0	2	1,688,110	0	0
—	Stack T_2	T_0	2	0	1,395,867	0
—	Stack T_3	T_0	2	0	0	1,614,711
Total	—	—	6,517,798	3,527,898	2,917,345	3,427,447

Table 1. Page usage profile of the matrix multiplication, running with 4 threads (T_0–T_3).

access of every thread for each page that the application accesses. Table 1 shows the memory access profile for a subset of pages that the application uses. For each page, we show the address of the page, the structure it is associated to (one of the matrices, `A`, `B`, or `C`, or a thread's stack), which thread performed the first access to the page, and the number of memory accesses from each thread to the page. This page usage profile exposes several issues for data mapping, which will be discussed in this section. Both locality-based and balance-based policies are affected. For the discussion, consider that the application is executing on a system with 4 NUMA nodes, and that each thread is running on a different node.

Initialization from a single thread The first access column shows that all pages were first accessed from thread 0. This means that with the default first-touch mapping policy, all pages will be allocated on the NUMA node where thread 0 was executing when the application was starting up, before performing the actual work. This contradicts the ideas behind the introduction of the first-touch mapping policy, which was developed as an improvement of memory access locality compared to a round-robin mapping of pages to nodes (Marchetti et al. 1995).

This behavior is an opportunity to improve both the locality and balance of the memory accesses. For example, page 1558 is mostly accessed by thread 1, but it is allocated on the NUMA node of thread 0. Migrating this page to the NUMA node of thread 1 increases the locality of accesses to the page. Furthermore, the NUMA node of thread 0 will handle all main memory accesses of the application, creating an imbalance on its memory controller. By distributing the pages more fairly among the NUMA nodes, a balance-based policy can improve the overall load on the memory controllers.

Stacks of multithreaded applications A related problem is shown in the allocation of the stacks of each thread. In Linux, multithreaded applications share the same virtual address space, but also contain data that is private to each thread. To create a multithreaded application on Linux, the runtime system uses the `clone()` or `fork()` system calls, which are executed in the context of the calling thread. This calling thread is usually the master thread of the application, that is, thread 0. The private data of the newly created thread needs to be zeroed (such as the stack, to prevent information leakage) or initialized (such as the Thread-Local Storage, TLS). For this reason, the created thread's private data is first accessed by the master thread, leading to an unfavorable page mapping with a first-touch policy.

The resulting data mapping is inefficient both in terms of locality and balance. It overloads the NUMA node that runs the master thread with the stack accesses of all threads, leading to an imbalance. As shown in the table, the vast majority of memory accesses to each stack are performed by the thread that owns the stack, leading to a bad locality of these accesses. These issues are further aggravated by the fact that the stack receives a lot of memory accesses compared to the pages of the other structures. An improved data mapping policy can help to improve the balance and locality of the stack usage.

Exclusive pages As mentioned before, the memory pages used for the stacks have a high exclusivity since they are accessed mostly from a single thread. Two of the matrices of the matrix multiplication are also exclusive, the `A` and `C`

structures. For example, page 1553 is accessed mostly by thread 0, while page 1558 is accessed mostly by thread 1. This pattern is repeated for the whole structure. Some of the pages that are on the border between two areas of the matrix have an overlap of accesses by two different threads, such as page 1557, which is accessed by threads 0 and 1. Despite this overlap, the page is still relatively exclusive, since more than 80% of accesses are performed by thread 1. For exclusive pages, the most appropriate data mapping policy is based on increasing locality. By placing pages on NUMA nodes that perform the most accesses to them, the overall memory access latency can be reduced.

This example also illustrates another important aspect. Performing a locality-based mapping policy can also increase the balance of memory accesses in some cases. Considering the `C` matrix as an example, which consists of 16 pages in total, 5 of which are shown in Table 1. In a system with 4 NUMA nodes, a locality-based policy would map 4 consecutive pages of `C` to the NUMA node of each thread, increasing the locality of the accesses. In addition, this mapping also maximizes the balance of the mapping compared to the first-touch policy, which would map all 16 pages to the NUMA node of thread 0. The `A` matrix has very similar behavior. This shows that for these two structures, a locality-based policy also increases the balance.

Shared pages The `B` matrix has a different structure of memory accesses. All 16 pages of this structure have an access pattern similar to the page 1537 shown in Table 1. These pages are accessed by all threads in a similar manner, with only small differences between the minimum and maximum numbers of accesses. This indicates that these pages are shared between threads, and a locality-based policy can improve the overall memory access locality only slightly. However, these pages are candidates for a balance-based policy and can be used to distribute the load on the memory controllers more equally. For example, `B` could be distributed equally among all the NUMA nodes, or its pages could be migrated from an overloaded node to reduce its load.

Overall balance Finally, it is also important to consider the overall memory access balance of the whole application. The number of memory access of each thread is shown in the *Total* row of Table 1. The numbers show that thread 0 is performing the majority of memory accesses of all threads (about 40%). By placing the pages that the threads access in a more balanced way, it is possible to improve the overall load on the memory controllers and achieve a fairer balance.

Summary We conclude that the default first-touch policy of modern operating systems presents inefficiencies for many parallel applications, in terms of the locality and balance of memory accesses, and optimizes for neither metric. This indicates that large improvements can be achieved by improved data mapping policies. Section will formalize the intuitive ideas of exclusivity, locality and balance that were introduced here.

Quantitative page usage

In this section, we introduce metrics to describe the memory access behavior of parallel applications on NUMA systems. We present metrics for the suitability of applications for locality-based and balance-based policies. We also discuss dynamic application behavior through the changes to the exclusivity during execution. Finally, we introduce metrics for the minimum memory usage of an application, below which no gains from data mapping can be expected, and a metric to describe the locality of a given data mapping.

Memory access exclusivity

The potential for locality-based data mapping of a page is proportional to the amount of memory accesses from a single NUMA node. That is, if a page is mostly accessed from the same node, it has more potential for locality-based mapping than a page that is accessed from several nodes. We call the highest number of memory accesses to a page from a single node compared to the number of accesses from all nodes the *page exclusivity* E_{Page}. The higher the exclusivity of a page, the higher its potential for locality-based mapping. E_{Page} is calculated with Equation 1, where $MemAcc[p][n]$ is the number of memory accesses to page p from node n, and N is the number of NUMA nodes. The $\max$ function returns the highest number of memory accesses to a page from a node.

$$E_{Page}[p] = \frac{\max(MemAcc[p])}{\sum_{n=1}^{N} MemAcc[p][n]} \tag{1}$$

The exclusivity is minimal when a page has exactly the same number of accesses from all nodes. In this case, the exclusivity is given by $1/N$. The exclusivity achieves its maximum value of 1 when all accesses to the corresponding page originate from the same NUMA node.

Besides the exclusivity of a page, the number of memory accesses to it has to be considered as well. The higher the number of accesses to a page, the higher its potential for data mapping. We take the number of memory accesses into account by measuring the exclusivity for the whole application. To calculate this *application exclusivity* E_{App}, we scale the exclusivity of each page with the number of memory accesses to it, and divide this value by the total number of memory accesses. This operation is shown in Equation 2, where P is the total number of pages. Like the page exclusivity, the minimum and maximum of the application exclusivity is $1/N$ and 1, respectively, when all

pages are fully shared or fully exclusive.

$$E_{App} = \frac{\sum_{p=1}^{P}(E_{Page}[p] \times \sum_{n=1}^{N} MemAcc[p][n])}{\sum_{p=1}^{P}\sum_{n=1}^{N} MemAcc[p][n]} \quad (2)$$

Memory access balance

Our second metric, *memory balance*, is used to analyze the suitability of a parallel application for balance-based data mapping. Memory balance is important in applications where the memory accesses are performed in such a way that some memory controllers are overloaded, while others are idle. To measure the *page balance* B_{Pages}, we introduce a new vector, $NodePages$, which consists of N elements. Each element $NodePages[n]$ contains the number of pages mapped to node n. Equation 3 calculates the page balance for all the pages that an application uses. Its structure is similar to the percent imbalance used in load balancing (Pearce et al. 2012).

$$B_{Pages} = \left(\frac{\max(NodePages)}{\sum_{n=1}^{N} NodePages[n]/N} - 1 \right) \times 100\% \quad (3)$$

The balance is maximum (with a value of 0%), when all nodes store the same number of pages. Higher values indicate a higher page imbalance. The highest imbalance is reached (with a value of $(N-1) \times 100\%$) when one node stores all the pages. Since not all pages have the same number of memory accesses, maximizing load balance considering Equation 3 may not improve the balance of memory controllers. To achieve an improved balance, we need to consider the number of memory accesses to each page. We store the number of memory accesses to each NUMA node in the $NodeAcc$ vector, as shown in Equation 4.

$$NodeAcc[n] = \sum_{p=1}^{P} MemAcc[p][n] \quad (4)$$

We can then calculate the *memory access balance* B_{Acc} of the application with Equation 5.

$$B_{Acc} = \left(\frac{\max(NodeAcc)}{\sum_{n=1}^{N} NodeAcc[n]/N} - 1 \right) \times 100\% \quad (5)$$

If the memory accesses are equally distributed among the nodes, B_{Acc} has a value of 0%. If all memory accesses are satisfied from a single node, the imbalance is maximized, and B_{Acc} has a value of $(N-1) \times 100\%$.

Dynamic behavior

To analyze the dynamic page usage, we divide the execution of each parallel application into time slices of a fixed size. For each time slice, we calculate for each page p if it needs to be migrated to the NUMA node with the most accesses. This behavior is expressed in Equation 6.

$$Migrate[p] = \arg\max(MemAcc_{cur}[p]) \neq \arg\max(MemAcc_{cur-1}[p] \quad (6)$$

For the analysis, we use time slices of 10 ms. This value is used to characterize the dynamic access behavior to pages and does not necessarily indicate that a page needs to be migrated. As another indicator of dynamic behavior, we also calculate the application exclusivity for each time slice. By determining the number of migrations with Equation 6, we can then calculate the *Dynamicity* of the page usage $Page_{Dyn}$ by normalizing it to the total execution time of the application, as shown in Equation 7. We expect applications with a higher dynamicity that require more migrations during execution to be less suitable for data mapping due to the overhead of the page migrations.

$$Page_{Dyn} = \frac{\#migrations}{execution\ time} \quad (7)$$

Locality of a page mapping

To compare different mapping policies to each other in terms of their locality, we first use Equation 8 to define if a single page is located on the node with the most accesses to it*. In the equation, $CurNode[p]$ is the NUMA node where page p is currently located, and the $\arg\max$ function returns the NUMA node with the highest number of accesses to p. Here, we assign 1 to the locality value if the page is located on the NUMA node with the highest number of accesses to the page, and 0 otherwise.

$$Loc_{Page}[p] = \begin{cases} 1, & \text{if } \arg\max(MemAcc[p]) = CurNode[p] \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

We scale the locality of each page with the number of accesses to it to calculate the locality of a page mapping for the entire application, expressed with Loc_{App}. Equation 9 shows this operation, where P is the number of pages. The minimum for Loc_{App} is 0%, when no page is mapped to the node with the highest accesses to it, and 100% when all pages are mapped to the node with the most accesses.

$$Loc_{App} = \frac{\sum_{p=1}^{P}\left(Loc_{Page}[p] \times \sum_{n=1}^{N} MemAcc[p][n]\right)}{\sum_{p=1}^{P}\sum_{n=1}^{N} MemAcc[p][n]} \quad (9)$$

*The balance of mappings can be compared directly with B_{Acc}.

Data mapping policies

To evaluate the impact of data mapping on the performance of the parallel applications, we compare several mapping policies to the default first-touch policy. These policies will be presented in this section.

The policies

The following mapping policies were evaluated: *Random*, *RoundRobin*, *Interleave*, *Locality*, *Remote*, *Balanced*, and *Mixed*. The Interleave policy is available in many operating systems, such as via the `numactl` tool in Linux (Kleen 2004). Locality is a policy similar to previous mechanisms that focus on locality improvements in NUMA systems (Marathe et al. 2010; Diener et al. 2014). Remote is the opposite of Locality. Balanced and Mixed are two new mapping approaches. Balanced distributes pages in such a way that all memory controllers resolve the same number of memory accesses. The Mixed policy presents a trade-off between locality and balance. In the following description of the mapping policies, $node[p]$ represents the NUMA node of a page p, and N represents the total number of nodes. Consider that $MemAcc[p][n]$ contains the number of memory access to page p from node n.

In the **Random** mapping, each page gets assigned randomly to a NUMA node. We use this mapping to validate the importance of data mapping and because it is the mapping that is most independent from the memory access behavior of the application. The following equation calculates this policy: $node[p] = \text{random}() \bmod N$, where the $\text{random}()$ function returns a random integer.

In the **RoundRobin** mapping, pages get allocated to NUMA nodes in the order that they are first accessed. The node for a page p gets assigned with the following equations: $node[p] = count \bmod N; \quad count = count + 1$. This mapping ensures that the same number of pages gets assigned to each node, independent of the memory access behavior and the addresses that the pages are allocated at.

In the traditional **Interleave** policy, pages get assigned to NUMA nodes according to their address. Usually, the lowest bits of the page address are used to determine the node. The following equation describes this behavior: $node[p] = \text{addr}(p) \bmod N$, where $\text{addr}(p)$ returns the page address of page p. This policy ensures that memory accesses to consecutive addresses are distributed among the nodes. Similar to the RoundRobin policy, it also maps equal numbers of pages to each NUMA node in case pages have contiguous addresses.

In the **Locality** policy, each page gets assigned to the NUMA node with the most memory accesses to the page. This policy ensures that the highest possible number of memory accesses are resolved by the local NUMA node. However, it does not take the balance between nodes into account and can lead to overloaded nodes.

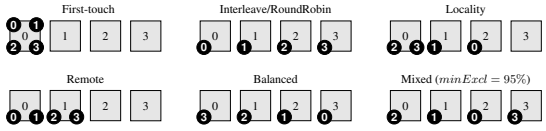


Figure 3. Output of the of data mapping policy example. Gray squares represent NUMA nodes, black circles repr. pages.

The following equation describes this behavior: $node[p] = \arg\max(MemAcc[p])$.

In the **Remote** policy, each page gets placed on the NUMA node that has the fewest accesses to that page. This mapping represents a worst-case for data mapping and is used to further evaluate its importance on parallel applications. The following equation describes this behavior: $node[p] = \arg\min(MemAcc[p])$.

In the **Balanced** mapping, we maximize the balance between the nodes, while still taking into account the locality of each page. In the algorithm, we first sort the list of pages by the number of accesses. Then, we map each page to the node with the most accesses to the page that is not overloaded. The full algorithm is presented in the original paper (Diener et al. 2015).

Balanced and Locality are opposite policies: Balanced optimizes memory access balance over locality, while the Locality policy maximizes local accesses at the expense of memory balance. For this reason, these policies can determine which of the metrics is more important for the performance of parallel applications.

In the **Mixed** mapping, we combine the locality and balance metrics. For each page, if its exclusivity E_{Page} is above a threshold $minExcl$, we allocate the page on the node with the highest number of accesses, as in the Locality policy. If the exclusivity is below the threshold, we allocate the page on the node given by the Interleave mapping. This behavior is formalized in Equation 10.

$$node[p] = \begin{cases} \arg\max(MemAcc[p]), & \text{if } E_{Page}[p] > minExcl \\ \text{addr}(p) \bmod N, & \text{otherwise} \end{cases} \tag{10}$$

This mapping provides a trade-off between locality and balance. For $minExcl = 100\%$, Mixed is equivalent to Locality, for $minExcl = 0\%$, it is identical to the Interleave policy.

The previously presented mapping policies will be compared to a **First-touch** policy, where each page gets allocated on the node that accessed the page for the first time (Marchetti et al. 1995). Pages are not migrated during execution.

Example of policy behavior

As an example of the behavior of the data mapping policies, consider the page usage example presented in Table 2. In

Table 2. Input of the data mapping example for an application consisting of four threads. For each page, the table shows which thread accessed a page first as well as the number of memory accesses to that page by each thread.

		Number of memory accesses			
Page	First access	T_0	T_1	T_2	T_3
0	T_0	1	0	1,000	0
1	T_0	1	1,000	0	0
2	T_0	1,000	0	0	0
3	T_0	1,000	0	0	50
Total	—	2,002	1,000	1,000	50

this example, a parallel application with 4 threads T_0–T_3 accesses 4 pages. All pages are accessed first by thread T_0, and the table shows the total number of memory access by each thread. Further assume that we want to execute this application on a system with 4 NUMA nodes, where each thread is mapped to a different NUMA node with a Compact thread mapping (Intel 2012) (that is, thread 0 executes on node 0, thread 1 on node 1, etc.).

Figure 3 shows the behavior of the different data mapping policies for this example. In the figure, gray squares represent the NUMA nodes, while the black circles represent the pages. The First-touch policy allocates all pages on the first NUMA node, since all pages get accessed first by thread 0, resulting in low memory access locality and a high imbalance. The Interleave and RoundRobin policies result in the same mapping, which has a good balance but a low locality. The Locality policy results in a very high locality, as expected, but due to the imbalanced application behavior does not distribute pages fairly among the nodes (node 3 stores no page, for example). The Remote policy results in a mapping with a low locality and high imbalance. The Balanced mapping distributes pages fairly among nodes, but pages 0–2 are mapped to nodes which have no accesses to them. The Mixed policy (shown for a locality threshold $minExcl$ of 95%), results in a slightly lower locality compared to the Locality mapping, but results in a very high page and memory access balance. The Random policy results in a different random assignment for each execution and is therefore not shown in the figure.

Page Usage of the Benchmarks

For the evaluation of the page usage of the NAS-OMP (Jin et al. 1999) and PARSEC (Bienia et al. 2008) benchmarks in this section, we run the applications in the numalize memory access tracer[†], which is based on the Pin dynamic binary instrumentation tool (Luk et al. 2005). We execute the applications with 64 threads, and record each memory access of each thread on the page granularity, using a page size of 4 KByte. We start with an analysis of the global application behavior, followed by a discussion of the dynamic page usage.

Global Behavior

We begin with a discussion of the global application behavior, that is, the behavior during the whole execution of the application.

Exclusivity The application exclusivity E_{App} is presented in Figure 4 for two configurations: (i) our baseline, with 64 threads on 4 NUMA nodes (threads are assigned to NUMA nodes such that the exclusivity is highest), (ii) 64 NUMA nodes (1 thread per node), to show the inherent exclusivity of the applications. Results are further divided into three page sizes, 4 KByte (which is the default page size in 32-bit and 64-bit x86 architectures (Intel 2013)), 2 MByte (which is the highest page size supported by 32-bit x86 with the Physical Address Extension (PAE) mode (Intel 2013), and 64 MByte (as an example of how the exclusivity changes when increasing page sizes further).

Most applications have a high exclusivity, even for 64 NUMA nodes, which demonstrates the importance of a locality-based mapping policy. Several benchmarks, such as EP-OMP, Canneal, and Streamcluster, have a lower exclusivity however. Even when increasing the page size or the number of NUMA nodes, the exclusivity only decreases slightly for most benchmarks. A slight decrease of exclusivity is expected for larger pages, as more shared data is stored in the same pages.

When increasing the number of NUMA nodes, more shared pages will be located on different nodes, lowering the exclusivity. On average, the application exclusivity for the 4 KByte, 2 MByte and 64 MByte page sizes is 83.3%, 77.1% and 71.3% (for 4 nodes), and 74.0%, 64.4% and 29.0% (for 64 nodes), respectively. Although the reduction is low in most cases, these results indicate that with larger pages or more NUMA nodes, fewer improvements from a locality-based data mapping can be expected. Nevertheless, in the most relevant case for data mapping, where the number of threads is much higher than the number of NUMA nodes, we expect still high improvements even with

[†]Numalize is available at `http://github.com/matthiasdiener/numalize`.

pages that are much larger than the current default of a few KByte.

Balance Figure 5 visualizes the page and memory access balance of the first-touch policy and a system with 4 NUMA nodes. For all NAS-OMP benchmarks except EP-OMP, as well as some of the PARSEC benchmarks, the policy distributes the pages fairly between the nodes. On the other hand, comparing the results for the page and memory access balance shows that a high page balance does not necessarily result in a high memory access balance. For example, UA-OMP with first-touch has a very good page balance with an almost equal number of pages on each node. However, the memory accesses of UA-OMP are very imbalanced, with node 0 handling more than 68% of the total accesses.

This intuition is confirmed by the values of the balance metrics, B_{Pages} and B_{Acc}, which are shown in Figures 6 and 7, respectively. In the figures, a value of 0 indicates perfect balance, while higher values indicate an imbalance in the distribution of pages and memory accesses between NUMA nodes. The maximum imbalance, when a single node stores all pages or handles all memory access, results in a value of 300 in this configuration. The RoundRobin policy is not shown in the figures, as its results were almost equal to the Interleave mapping.

Regarding the page balance, Random and Interleave are almost perfectly balanced, as expected. For the NAS-OMP benchmarks, only Remote and First-touch are significantly imbalanced. On the other hand, most PARSEC benchmarks are imbalanced with all policies except Random and Interleave. Regarding the memory access balance, most policies result in higher imbalances. Only the Balance policy results in a high balance for most of the benchmarks. It is important to mention that the Locality policy already results in a much better memory access balance than the First-touch policy. For example, the memory access balance of UA-OMP increases from 47% with First-touch to 95% with Locality. The Mixed policy results in a balance that is between the interleave and locality policies, as expected. Table 3 shows the average values of both metrics over all benchmarks for each policy.

We show a comparison of the B_{Acc} metric of the first-touch policy compared to the main improved mapping policies (Interleave, Locality, Balance, and Mixed) in Figure 8. The values were calculated for each application and policy with the following equation:

$$\Delta B_{Acc} = Policy_{B_{Acc}} - Firsttouch_{B_{Acc}} \qquad (11)$$

The results show that for almost all of the benchmarks, each of the improved mapping policies results in a better balance than first-touch. Only a few of the benchmarks, such as CG-OMP and FT-OMP, are already well-balanced with first-touch and can not profit much from another policy. Most other benchmarks can benefit much more. It is interesting to note that the Balance policy, which results in the highest balance improvements in most cases since it focuses on the memory access balance, shows very similar results to the other policies. This indicates that this policy might not be necessary to achieve a better balance.

Locality To compare different mapping policies in terms of their memory access locality, we evaluate the Loc_{Page} and Loc_{App} metrics defined in Section . Figure 9 shows these metrics for the first-touch page mapping for 4 KByte pages and 4 NUMA nodes. The results for the per-page locality Loc_{Page} show that the first-touch mapping results in a high page locality for most applications. However, when comparing it to the weighed locality Loc_{App}, the locality falls for most applications. This indicates that although the large majority of pages are placed to the NUMA node with the most accesses to them, other pages that are *not* placed on their local node receive an above-average number of memory accesses.

As an example of this behavior, consider the UA-OMP benchmark. Although more than 92% of its pages are placed on the NUMA node with the most accesses to them, these pages receive only 68% of the total memory accesses. The 8% of pages that are mapped to a non-local node receive 32% of memory accesses. Therefore, in order to perform a locality-based data mapping, it is possible to start with a first-touch mapping and then migrate a relatively low number of pages to their local NUMA nodes. These pages have an above average number of memory accesses for many of the benchmarks.

We show a comparison of the Loc_{App} metric of the first-touch policy compared to the main improved mapping policies (Interleave, Locality, Balance, and Mixed) in Figure 10. The values were calculated for each application with the following equation:

$$\Delta Loc_{App} = Policy_{Loc_{App}} - Firsttouch_{Loc_{App}} \qquad (12)$$

The results show that the Interleave policy greatly reduces Loc_{App}, causing a lower memory access locality, despite resulting in a better balance. The other three mapping policies consistently increase the memory access locality for almost all benchmarks, with the highest improvements for the Locality policy, as expected. This value, ΔLoc_{App}, can be used to estimate the performance improvements that can be gained from locality-based mapping policies.

Table 3. Average values of the balance metrics of the mappings policies. Lower values indicate a higher balance.

Metric	First-touch	Locality	Remote	RoundRobin	Interleave	Randor
B_{Pages}	121.2	73.7	177.2	0.007	0.438	1.0
B_{Acc}	171.2	66.4	163.8	28.9	30.2	32.

Dynamic behavior

To evaluate the dynamic behavior of the workloads, we measure the number of page migrations that have to be performed in order to maintain each page on the NUMA node with the most accesses during time windows of 10 ms.

The values of the $Page_{dyn}$ metric for this experiment are shown in Table 4, presenting a summary of the number of pages that have to be migrated for each second of execution time of the benchmarks. Although the numbers for some benchmarks appear to suggest that a large amount of data needs to be migrated, the absolute amount of data is actually quite low. For example, for the Vips benchmark, which has the highest dynamicity, about 34,000 pages need to be migrated per second. This corresponds however to only about 130 MByte of data that needs to be copied between NUMA nodes, which is a reasonably low amount of data, even when considering an extremely short time window of 10 ms.

The dynamic page usage behavior of the SP-OMP benchmark is shown in Figure 11a. For each time slice, we show the number of pages that need to be migrated to the NUMA node with the highest number of accesses during that slice, as well as the application exclusivity E_{App}. The results show that during the initialization of the application, the exclusivity varies between 60% and 80%, indicating shared accesses to memory pages. As soon as the parallel part of SP-OMP starts, the exclusivity begins to stabilize and soon reaches its maximum of 92%. For the number of migrations, we can confirm that a significant number of page migrations are necessary only during the time slice at the start of the parallel computation. For the rest of the execution, the page usage is stable and almost no pages need to be migrated.

As an example of an application with a more dynamic behavior, Figure 11b presents the page usage during the execution of the Vips benchmark, which is the application with the most migrations in our experiments. The initialization phase with only a single thread lasts until about 5% of the total execution time. During the whole parallel phase of the application, the exclusivity stays at about 81%. However, in contrast to SP-OMP, pages need to be migrated during the whole execution to maintain them on the node with the highest number of accesses. At each time step, about 3,500 pages need to be migrated, which represents 0.4% of the total number of pages that Vips uses. With this very dynamic behavior, we expect fewer benefits from a data mapping policy.

Evaluating Data Mapping Policies

This section evaluates the performance improvements of the data mapping policies that were proposed in Section .

Methodology

We experiment with the same two parallel benchmark suites as before, the OpenMP implementation of the NAS Parallel Benchmarks (Jin et al. 1999) and the PARSEC suite (Bienia et al. 2008). We evaluate these applications on three real NUMA machines: *Itanium*, *Xeon*, and *Opteron*. The *Itanium* machine represents a traditional NUMA architecture based on the SGI Altix 450 platform. It consists of 2 NUMA nodes, each with 2 dual-core Intel Itanium 2 processors and a proprietary SGI interconnection. The *Xeon* machine represents a newer generation NUMA system with high-speed interconnections. It consists of 4 NUMA nodes with 1 eight-core Intel Xeon processor each. The *Opteron* machine represents a recently introduced generation of NUMA systems with multiple on-chip memory controllers. It consists of 4 AMD Opteron processors, each with 2 memory controllers, forming 8 NUMA nodes in total. The *Itanium* machine can execute up to 8 threads concurrently, while *Xeon* and *Opteron* can execute 64 threads concurrently, and the benchmarks were executed with these numbers of threads. The NUMA factors of the machines, measured with the Lmbench tool, are 2.1, 1.5 and 2.8, respectively.

We evaluate the data mapping policies and compare them to the baseline, the first-touch policy of Linux. Since its results were very close to the Interleave policy, the RoundRobin mapping will not be shown in the figures. As these mapping policies are calculated based on memory access traces, we require that the addresses themselves do not change between multiple executions. Addresses can change in case the parallel application calls `malloc()` from multiple threads. Most NAS-OMP benchmarks and some PARSEC applications have static addresses between executions, and we filter out those applications where the memory addresses change.

Since the behavior on our evaluation systems differs considerably, we present results for all three of them. Threads were pinned to the execution cores with a Compact thread mapping (Intel 2012), to remove the influence of thread migrations during execution. We show the average performance improvement of 10 executions for each benchmark, normalized to the first-touch policy.

Performance Evaluation

The results for the performance improvements (compared to the baseline, the first-touch policy of Linux) for the three evaluated architectures are shown in Figures 12, 13 and 14. For the Mixed policy, we show the improvements for a value of $minExcl$ of 90%.

Results Figure 12 shows the results for the *Itanium* machine. The highest improvements were achieved for the CG-OMP benchmark, of up to 67% for Locality,

Table 4. Dynamic page usage $Page_{dyn}$ of the benchmarks. Number of page migrations per second execution time.

	NAS-OMP										PARSEC											
Metric	BT-OMP	CG-OMP	DC-OMP	EP-OMP	FT-OMP	IS-OMP	LU-OMP	MG-OMP	SP-OMP	UA-OMP	Blackscholes	Bodytrack	Facesim	Ferret	Freqmine	Raytrace	Swaptions	Fluidanimate	Vips	X264	Canneal	Dedup	Streamcluster
$Page_{dyn}$	30	501	1	1	1,228	1,655	2419	2,348	31	221	4	20	104	85	915	5,019	1	4,709	34,124	52	9	123	2,792

Balanced and Mixed. Due to *Itanium*'s relatively slow interconnection, the Interleave policy achieves generally lower improvements than the Locality and Mixed policies. Since it still takes locality into account, the Balanced policy achieves higher improvements than Interleave. As the machine has only a small number of NUMA nodes (2), even the Random policy improves performance in many cases, as it has a higher chance of mapping the page to the local node. The average performance improvements are: 5.2% with Random, 5.7% with Interleave, 20.9% with Locality, 14.4% with Balanced and 19.4% with Mixed. The fact that the improvements with Mixed are lower than the improvements with Locality (even when using a 99% minimum exclusivity for mixed) shows that on this machine, improving memory access locality is more important than balance.

Figure 13 shows the results for the *Xeon* machine. Due to the higher number of threads and NUMA nodes, performance improvements from data mapping are higher than on *Itanium*, despite *Itanium*'s slower interconnection. For the same reason, the balance policies achieve improvements closer to the locality policies. The SP-OMP benchmark achieved the highest improvements, even for the Random policy, of up to 108%. For the other benchmarks, the Random policy can not reduce execution time significantly or even increases it (up to 37% for MG-OMP). On average, performance was improved by 8.7% with Random, 15.8% with Interleave, 18.5% with Locality, 14.6% with Balanced and 23.7% with Mixed. The results show that increasing only locality is not enough to achieve optimal improvements. For several benchmarks (BT-OMP, LU-OMP, and Canneal), the Locality policy actually reduces performance compared to First-touch.

Figure 14 shows the results for the *Opteron* machine. Due to its large number of NUMA nodes (8) and high NUMA factor, the highest improvements were achieved on this machine. In many cases, the Random and Interleave policies reduce performance. Balancing memory accesses is still important though, as shown by the higher improvements for Mixed than for Locality. As before, the Locality policy increases execution time slightly compared to the baseline for some benchmarks (CG-OMP, LU-OMP, and UA-OMP). Similar to the Xeon machine, the highest improvements were achieved for SP-OMP, up to 272% with the mixed policy. On average, performance was improved by 6.3% with Random, 7.2% with Interleave, 50.0% with Locality, 23.9% with Balanced and 61.8% with Mixed.

Summary and Discussion From the performance results, we can draw several interesting conclusions. Most importantly, the first-touch data mapping often has a negative impact on performance. In many cases, even a random assignment of pages to NUMA nodes outperforms first-touch. The reason for this behavior is that in many parallel applications, one thread initializes most data and forces page allocation on a single NUMA node, leading to an increased number of memory accesses to that node, while a random policy can balance the memory access load more equally among the nodes.

Regarding the importance of locality and balance, results depend on the hardware architecture. On traditional NUMA systems with a relatively slow interconnection, such as our *Itanium* machine, memory access locality is the most important metric for performance improvements. For modern NUMA architectures, the importance of balancing the memory accesses between memory controllers is becoming increasingly important, although locality is still the most important metric to optimize, as evidenced by the results of the Interleave and Balanced policies compared to the Locality policy. The reason for this result is that improving locality also improves balance for many applications.

Finally, both the Locality and Balanced policies actually reduce performance significantly in some experiments. Taking both locality and balance into account (as done by our Mixed policy) when mapping pages to NUMA nodes achieves the highest results overall and also avoids the performance decrease of the other policies. In this way, highly exclusive pages can benefit from the locality, while shared pages are distributed among the nodes to balance the memory accesses between memory controllers.

Comparison to Predicted Data Mapping Suitability In Section , we discussed the suitability for data mapping, considering 64 threads and 4 NUMA nodes,

which corresponds to our *Xeon* machine. Comparing the performance results on *Xeon* to the prediction showed that suitability was analyzed correctly in most cases. From the NAS-OMP benchmarks, we only characterized EP-OMP and LU-OMP as being unsuitable for data mapping, which is reflected in their very low improvements and performance reduction in some cases. All other NAS-OMP benchmarks improve performance with data mapping. The PARSEC benchmarks evaluated in this section were all characterized as suitable for data mapping, and they have significant performance gains.

Similar to the analysis for task mapping, although the characterization was very accurate in terms of which benchmarks can benefit from data mapping, the performance improvements still vary widely between benchmarks, which is not well reflected by the metrics. This is due to two main factors. First, not all information about the memory access pattern of the application is collected. For example, the suitability of the pattern for the cache line prefetcher is not taken into account. If the prefetcher can perform an accurate prediction of future memory access, data mapping is less effective, as more memory accesses can be filtered by the caches. On the architectural level, cache size, interconnection speed and contention influences the benefits of data mapping. To provide such a level of precision, much more detailed architectural simulation is required, which would limit the applicability of the characterization to small applications.

Conclusions

In NUMA architectures, data mapping has a high influence on the performance of parallel applications. Two basic strategies, improving the locality and balance of memory accesses to pages, have been proposed. To perform an effective data mapping, the memory access behavior of the parallel application, as well as the characteristics of the NUMA system have to be taken into account.

This paper introduced a set of metrics to comprehensively describe the memory access behavior, focusing on determining the suitability for different data mapping policies. We presented a data mapping policy that takes both locality and balance into account when performing the mapping. In an evaluation with two parallel benchmark suites, this mixed policy showed the highest overall improvements compared to the default first-touch data mapping of most current operating systems. Although our results showed that locality-based policies outperform balance-based policies in most cases, both types result in performance losses in some cases, which can be avoided with the mixed policy.

Funding

This work was supported by CNPq and Capes.

References

Awasthi M, Nellans DW, Sudan K, Balasubramonian R and Davis A (2010) Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In: *Parallel Architectures and Compilation Techniques (PACT)*. pp. 319–330.

Bienia C, Kumar S, Singh JP and Li K (2008) The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. pp. 72–81.

Blagodurov S, Zhuravlev S, Dashti M and Fedorova A (2010) A Case for NUMA-aware Contention Management on Multicore Systems. In: *USENIX Annual Technical Conference (ATC)*. pp. 557–571.

Corbet J (2012) Toward better NUMA scheduling. URL `http://lwn.net/Articles/486858/`.

Dashti M, Fedorova A, Funston J, Gaud F, Lachaize R, Lepers B, Quéma V and Roth M (2013) Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. pp. 381–393.

Diener M, Cruz EHM and Navaux POA (2015) Locality vs . Balance: Exploring Data Mapping Policies on NUMA Systems. In: *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. pp. 9–16.

Diener M, Cruz EHM, Navaux POA, Busse A and Heiß HU (2014) kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. pp. 277–288.

Intel (2012) Using KMP_AFFINITY to create OpenMP thread mapping to OS proc IDs. URL `https://software.intel.com/en-us/articles/using-kmp-affinity-to-create-openmp-thread-mapping-`

Intel (2013) Intel© 64 and IA-32 Architectures Software Developer's Manual.

Jin H, Frumkin M and Yan J (1999) The OpenMP implementation of NAS Parallel Benchmarks and Its Performance. Technical Report October, NASA.

Kleen A (2004) An NUMA API for Linux. Technical report. URL `http://andikleen.de/numaapi3.pdf`.

LaRowe Jr RP (1991) *Page Placement For Non-Uniform Memory Access Time (NUMA) Shared Memory Multiprocessors*. PhD Thesis, Duke University.

Luk C, Cohn R, Muth R and Patil H (2005) Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. pp. 190–200.

Majó Z (2014) *Modeling Memory System Performance of NUMA Multicore-Multiprocessors*. PhD Thesis, ETH Zurich.

Majo Z and Gross TR (2012) Matching memory access patterns and data placement for NUMA systems. In: *International*

Symposium on Code Generation and Optimization (CGO). pp. 230–241.

Mandal A, Fowler R and Porterfield A (2010) Modeling memory concurrency for multi-socket multi-core systems. In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. ISBN 9781424460229, pp. 66–75. DOI:10.1109/ISPASS.2010.5452064.

Marathe J, Thakkar V and Mueller F (2010) Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces. *Journal of Parallel and Distributed Computing (JPDC)* 70(12): 1204–1219.

Marchetti M, Kontothanassis L, Bianchini R and Scott ML (1995) Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In: *International Parallel Processing Symposium (IPPS)*. ISBN 0-8186-7074-6, pp. 480–485.

Pearce O, Gamblin T, de Supinski BR, Schulz M and Amato NM (2012) Quantifying the effectiveness of load balance algorithms. In: *ACM International Conference on Supercomputing (ICS)*. ISBN 9781450313162, pp. 185–194. DOI:10.1145/2304576.2304601.

Ribeiro CP, Castro M, Méhaut JF and Carissimi A (2010) Improving memory affinity of geophysics applications on NUMA platforms using Minas. In: *International Conference on High Performance Computing for Computational Science (VECPAR)*. pp. 279–292.

Verghese B, Devine S, Gupta A and Rosenblum M (1996) OS Support for Improving Data Locality on CC-NUMA Compute Servers. Technical report.

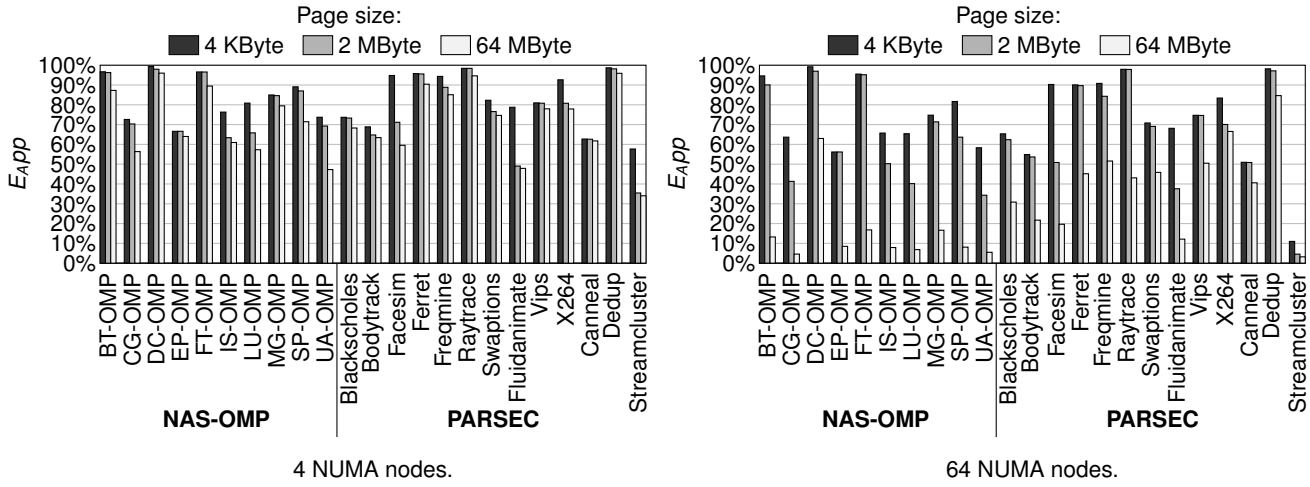
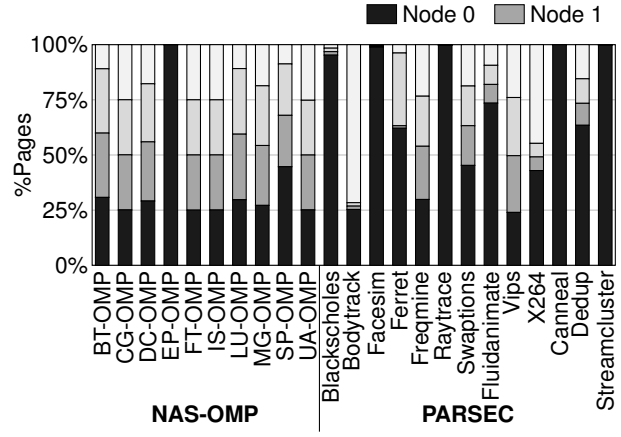


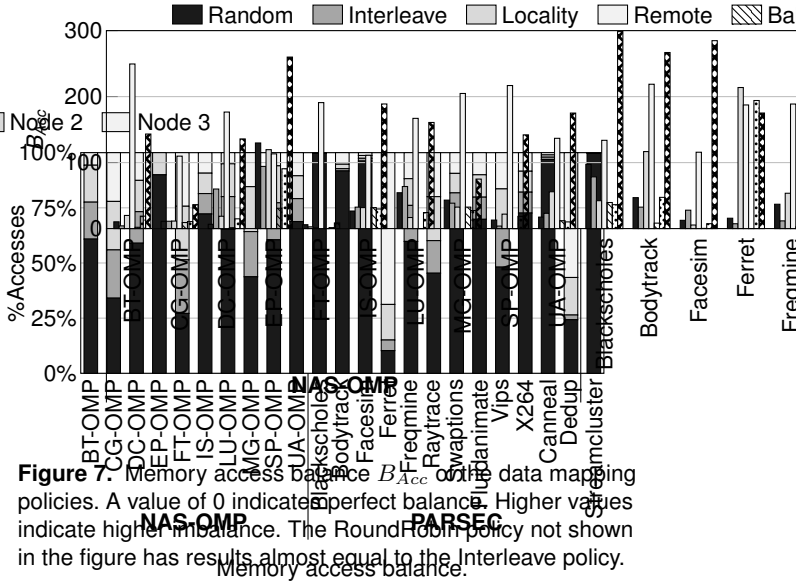
Figure 4. Application exclusivity E_{App} for 64 threads and different numbers of NUMA nodes and page sizes.

Table 5. Overview of the systems used in the evaluation.

Name	Properties
Itanium	2 NUMA nodes; Proc.: 4× Intel Itanium 2 9030, 2 cores, no SMT; Mem.: 16 GB DDR-400, page size 16 KB
Xeon	4 NUMA nodes; Proc.: 4× Intel Xeon X7550, 8 cores, 2-way SMT; Mem.: 128 GB DDR3-1066, page size 4 KB
Opteron	8 NUMA nodes; Proc.: 4× AMD Opteron 6386, 8 cores, 2-way SMT; Mem.: 128 GB DDR3-1600, page size 4 KB



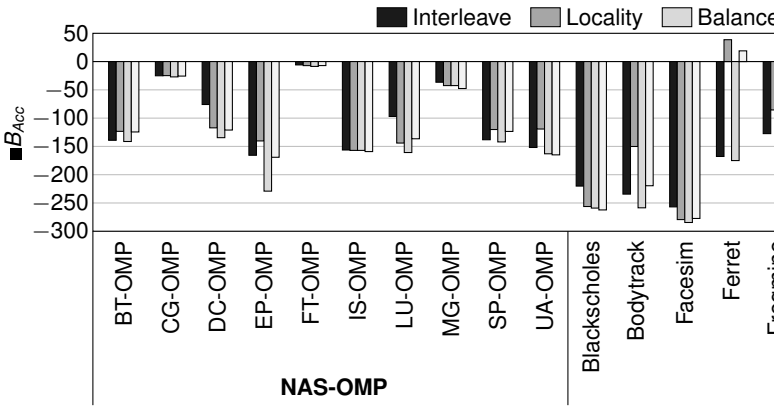
Page balance.



Memory access balance.

Figure 7. Memory access balance B_{Acc} of the data mapping policies. A value of 0 indicates perfect balance. Higher values indicate higher imbalance. The RoundRobin policy not shown in the figure has results almost equal to the Interleave policy.

Figure 5. Page balance and memory access balance of the first-touch policy for a system consisting of 4 NUMA nodes. The color of each bar indicates how many pages or memory accesses are handled by each node.



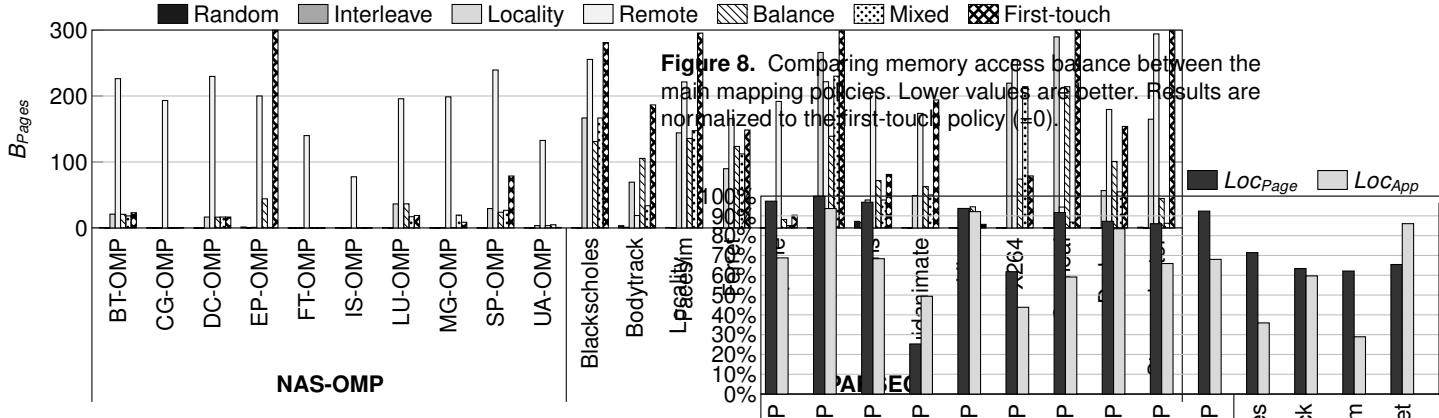


Figure 6. Page balance B_{Pages} of the data mapping policies. A value of 0 indicates perfect balance. Higher values indicate higher imbalance. The RoundRobin policy not shown in the figure has results almost equal to the Interleave policy.

Figure 8. Comparing memory access balance between the main mapping policies. Lower values are better. Results are normalized to the first-touch policy (=0).

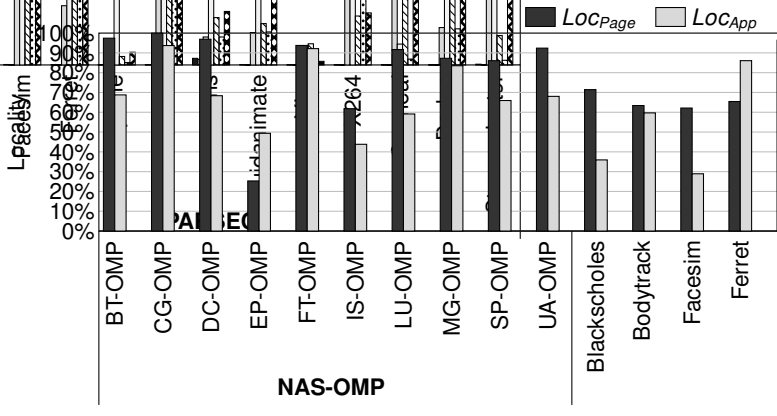


Figure 9. Locality of the first-touch page mapping for 4 KByte pages and 4 NUMA nodes.

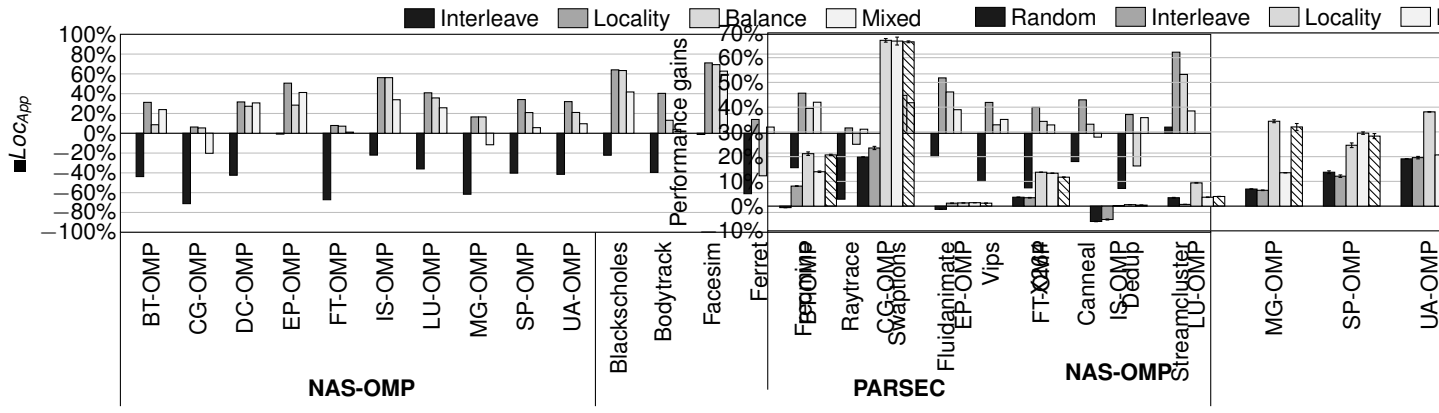
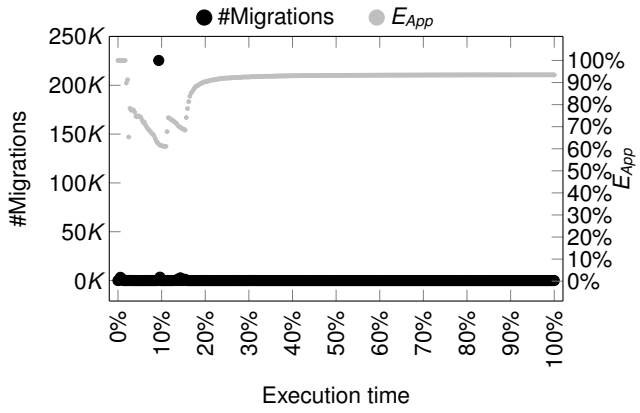
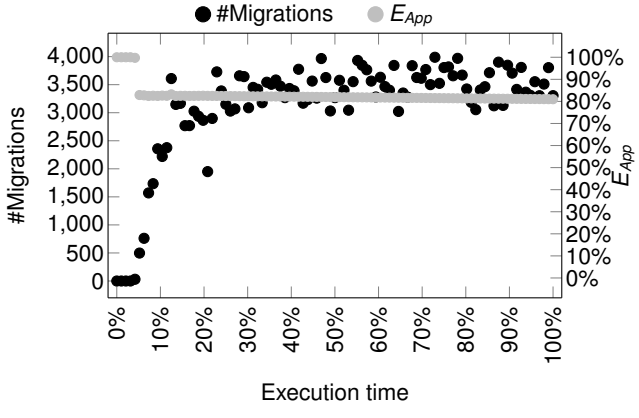


Figure 10. Comparing memory access locality between the main mapping policies. Values are normalized to the first-touch policy (=0%).



(a) SP-OMP.



(b) Vips.

Figure 11. Dynamic page usage of two benchmarks.

Figure 12. Performance improvements on *Itanium*, compared to the first-touch mapping.

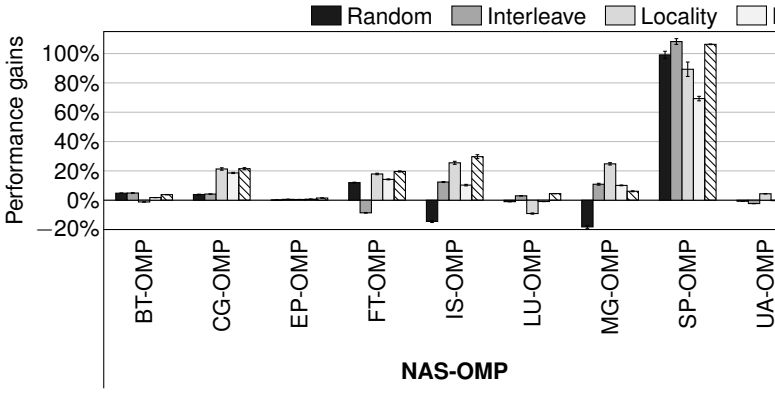


Figure 13. Performance improvements on *Xeon*, compared to the first-touch mapping.

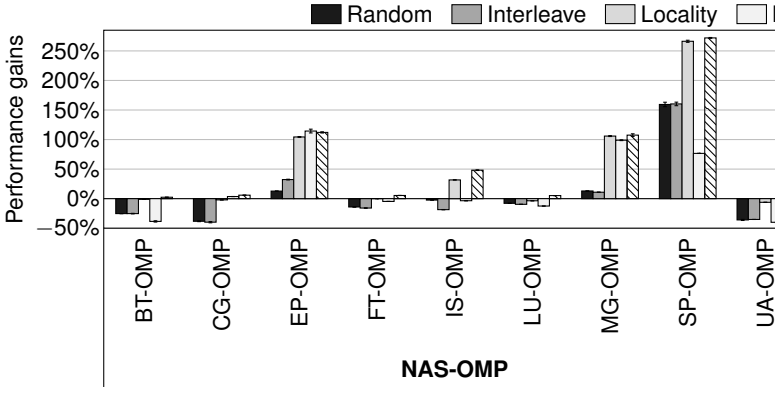


Figure 14. Performance improvements on *Opteron*, compared to the first-touch mapping.