Noname manuscript No.
(will be inserted by the editor)

A Flexible and Dynamic Page Migration Infrastructure Based on Hardware Counters

Juan A. Lorenzo-Castillo · Juan C. Pichel · Francisco F. Rivera · Tomás F. Pena · José C. Cabaleiro

Received: date / Accepted: date

Abstract Performance counters, also known as *hardware counters*, are a powerful monitoring mechanism included in the *Performance Monitoring Unit* (PMU) of most of the modern microprocessors. Their use is gaining popularity as an analysis and validation tool for profiling, since their impact is virtually imperceptible and their precision has noticeably increased thanks to the new *Precise Event-Based Sampling* (PEBS) features.

In this paper we present and evaluate a novel user-level tool, based on hardware counters, for monitoring and migrating pages dynamically. This tool supports different migration strategies, being able to attach and monitor a target application without need to modify it whatsoever. The page migration process is performed timely and its overhead is overcome by the benefit of the data locality achieved.

As a case study, an access-based migration algorithm was implemented and integrated into our tool. Performance results on a NUMA system show a noticeable reduction of remote accesses and execution time, achieving speedups of up to ∼21% in a multiprogrammed environment.

Keywords Hardware Counters, Page Migration, NUMA

This work has been partially supported by Hewlett-Packard under contract 2008/CE377, by the Ministry of Education and Science of Spain, FEDER funds under contract TIN 2010-17541 and by the Xunta de Galicia (Spain) under contract 2010/28 and project 09TIC002CT. This work is in the frame of the Spanish network CAPAP-H. The authors also wish to thank the supercomputer facilities provided by CESGA.

Juan A. Lorenzo-Castillo · Juan C. Pichel · Francisco F. Rivera · Tomás F. Pena · José C. Cabaleiro
Centro de Investigación en Tecnoloxías da Información (CITIUS)
University of Santiago de Compostela, Spain
E-mail: {juanangel.lorenzo, juancarlos.pichel, ff.rivera, tf.pena, jc.cabaleiro}@usc.es

1 Introduction

Over the last years, we have witnessed an important evolution in the available computational resources in science and engineering. The line that has traditionally separated multicomputers from multiprocessors is getting blurred, and nowadays most modern supercomputers include several multicore, NUMA (Non-Uniform Memory Access) multiprocessor nodes interconnected by a high-speed network.

As systems have grown in complexity, the need for understanding what is happening inside an application has also increased. Profiling, understood as a performance monitoring technique that records information about a running code, has proven very useful to narrow down its bottlenecks. In this way, the performance monitoring hardware counters, included in the vast majority of modern microprocessors, provide an essential tool to monitor and gain an insight into the system during the execution of a code. Recently, a new player has come on stage. *Precise Event-Based Sampling* (PEBS) is a hardware counter-based profiling technique with a very small overhead and a high precision. This opens the door to the development of new tools and optimization techniques based on the information provided by the hardware counters.

On a NUMA environment, the adequate placement of data is essential to improve the performance of a code. Thread migration and preemption are common events in non-dedicated environments during the life cycle of a parallel application, which may change its page affinity requirements (i.e., the memory module to which a page is bound).

Not until recently did Linux gain NUMA awareness [5]. Its *first-touch* page migration policy has proved beneficial when the OS scheduler initially distributes the threads of a program among the available processors. In this case, a dataset will be allocated in a memory local to the thread that first touches it, and it will remain there regardless of which thread accesses it, with a potential latency increase if it is fetched from a remote thread. This is a common situation in parallel programming languages such as OpenMP [15]. Even in those cases in which the first-touch policy allocates each dataset close to each thread, any page may happen to be accessed by several threads scheduled in different cells. In these situations, a *dynamic migration policy* can mitigate the problem of efficiently accessing data.

In this paper we present a hardware counter-based, dynamic page migration tool for Linux. To our knowledge, nobody has hitherto developed an efficient user-level migration framework for Linux platforms based on the information provided by the hardware counters. The main contributions of our tool are:

- No modifications of the monitored application are required.
- No knowledge about the underlying hardware and application is needed.
- It is a user-level approach, with the implicit easiness of use and portability across platforms.
- It is flexible for the programmers: adding a new migration policy is straightforward.
- Its overhead is overcome by the benefit of the data locality achieved.

As a case study, an access-based migration algorithm was implemented and integrated into our tool. We have evaluated several parallel applications using our migration tool on dedicated and multiprogrammed environments. The experiments shown in the paper have been performed on the FinisTerrae supercomputer [4] installed at Galicia Supercomputing Centre (CESGA), which consists of more than 2500 Itanium2 processors. The PEBS hardware counters present on Itanium2 processors, such as the *Event Address Registers* (EARs) [6], provide event resolution for instruction and data cache misses, allowing an unambiguous localization of latency data and instruction accesses. Experiments show a noticeable reduction of remote accesses and execution time, reaching speedups up to 20.6% in the multiprogrammed environment.

The rest of this article is organized as follows: Section 2 assesses the state of the art in data allocation techniques. Section 3 presents and discusses the design of our monitoring and migration tool. Section 4 performs several tests to evaluate the page migration infrastructure. In Section 5 an access-based migration algorithm is integrated into our tool and evaluated on different scenarios. The results and applicability of our infrastructure are discussed in Section 6. Finally, Section 7 presents the main conclusions derived from this work.

2 Related Work

Research on the first generation of NUMA machines in the 80s and early 90s already addressed the data placement problem [1,8]. Most of these works introduced kernel-level page migration policies based on page fault mechanisms and designed for multiprocessors with large NUMA factors. In addition, these policies were developed on the context of non-cache-coherent NUMA systems.

The NUMA support in Linux lacks some features that have been present for a long time in other systems. Hence, most of the works have studied the proper memory placement of data on systems such as Solaris or Irix. The dynamic page migration infrastructure introduced in this paper tries to fill that gap for Linux systems.

Tikir *et al.* [20] introduce a user-level, profile-driven page migration scheme using performance counters on a Solaris 9 Sun Fire 6800 server. Their migration algorithm is based on the number of times a page is accessed by a processor. Unlike our approach, it requires inserting instrumentation code into the monitored application. The same authors had also previously proposed a dynamic user-level page migration scheme based on an approximate trace of memory accesses obtained by sampling the network interconnect [19].

Nikolopoulos *et al.* present in [12] and [13] two algorithms for moving virtual memory pages to the nodes that reference them more frequently on an IRIX system. The first one, for OpenMP iterative codes, assumes that the page reference pattern of one iteration will be repeated throughout the execution of the program. The second algorithm checks for hot memory areas and migrates the pages with excessive remote references. A more recent work from

the same authors dynamically collocates threads and memory affinity sets of iterative programs in the presence of unpredictable scheduler interventions [14] on a SGI Origin2000. This proposal requires compiler support by linking the monitored program to a page-migration library.

Bull and Johnson study the tradeoffs between page migration, replication and data distribution for OpenMP applications on the Sun WildFire system [2]. They suggest that page replication can be even more beneficial than migration. Tao *et al.* [17] propose three page migration algorithms supported by memory access histograms on a shared memory in a LAN-like environment PC clusters. Their algorithms require a large amount of references issued before a migration decision can be taken. Additionally, they assume that if a page is accessed, the neighboring pages will be also accessed by the same node.

Wilson and Aglietti [22] implement a Dynamic Page Placement (DPP) strategy by a replication/migration decision tree on a cc-NUMA multiprocessor simulator, proposing several ideas for improving DPP.

On Linux systems, Marathe *et al.* [10] introduce a hardware-assisted page placement scheme based on automated profiling on a SGI Altix architecture using the hardware counters of the Itanium2 and the *libpfm* library. Their method needs to run previously a truncated version of an OpenMP program to extract a trace of its memory accesses. The desired page placement is performed by the OS first-touch and, once the data have been placed, they cannot be reallocated. Note that our migration tool does not require the modification of the considered application, and pages can be migrated at any time during the execution. Closely related to the previous work, Thakkar [18] uses *libpfm* on Itanium2-based, SGI Altix and x86-64 Opteron platforms to study the use of hardware counters to assist dynamic page placement. His proposal of a latency-based algorithm considers that the access latencies from a given node are constant. His work on the Itanium2 platform was abandoned due to the instability of the traces obtained, and his results on Opteron show an improvement of 8-15%.

In a recent work [9], the authors study different mechanisms such as in-program data migration and different loop iteration distributions to improve the performance of the NAS benchmarks. They obtain good results with respect to the default first-touch policy used by the Linux kernel. Unlike our proposal, a reasonable amount of knowledge of the underlying architecture is required. In addition, applications have to be modified.

A different approach to kernel-level dynamic page migration is presented by Goglin *et al.* in [5]. The authors develop an implementation of a next-touch memory placement for the Linux kernel in the i386 architecture. Their proposal modifies the kernel to have a page migrated close to the thread that last accessed it. This implementation is yet to be included in the mainstream kernel.

Some authors have applied page migration techniques in the context of virtual machines. For example, Wang *et al.* [21] change the physical location of a logical page owned by a virtual machine to perform a dynamic cache repartition. They observe that the cost of migrating a page is high, and further

studies are required to decide how many pages to migrate and which one among all machine pages should be migrated first.

We must highlight that none of the approaches detailed above fulfill all the requirements of our hardware counters-based page migration infrastructure introduced in the next section.

3 Design Considerations

A first requisite to design our tool was to get insight into a mechanism to obtain a model accurate enough of the memory map of a monitored application. A key advantage to approach this issue has been the availability of the information provided by PEBS hardware counters such as the access latencies and exact memory addresses where events occur.

In order to develop a tool that provides information about page affinity and locality of a multithreaded application, as well as to take decisions to improve its performance, the following prerequisites were defined:

1. Attach **seamlessly** to any multithreaded application given as a parameter, detecting automatically its threads.
2. Most of the page migration techniques found in the literature require the monitored program to be linked to a given library or have its code modified somehow. The aim of our proposal is that **no modifications** of the monitored program are required.
3. Run as a **user-level** tool, so that it can be installed and manipulated by any user and can easily be ported across architectures.
4. Provide enough information and **flexibility** to associate different migration strategies.
5. Perform page migrations **timely**. That is, shortly after the need to migrate is detected.

Our software infrastructure for page migration must provide profiling capabilities –to inspect the data accesses in memory performed by a program in runtime– and effectively modify the allocation of such data so that the whole program locality improves. Hence, it comprises three parts: a ***monitoring stage***, an ***evaluation stage*** and a ***migration stage***. The combo `libpfm/Perfmon2` [3,16] was chosen to write a program that complies with these requirements. Prior to developing the infrastructure, some timing constraints must be discussed.

Consider the time scenario depicted in Figure 1. A program is profiled during a time t_i $(i = 1, 2, ...n)$ by a concurrent *monitoring thread*. The profiled program runs normally during every *monitoring period* (t_p) until the buffer that stores samples gets full and overflows at t_{intj} $(j = 1, 2, ...m)$. For each overflow, the profiled process is stopped and an interruption is raised, with overhead t_{ovj}. Next, while the profiled program keeps on running normally, the interruption is handled by the *monitoring thread* $(hndl_{intj})$. Then, monitoring is resumed and the whole process starts over. When t_p expires, the

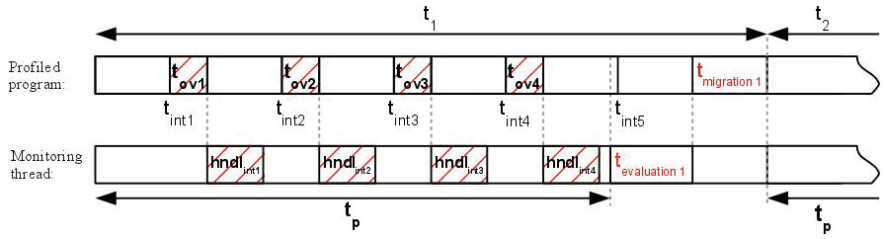


Fig. 1 Time intervals in a profiling process.

evaluation stage begins and all the collected data are processed by applying a migration algorithm (period $t_{evaluation\ i}$). Next, a number of pages are migrated if needed ($t_{migration\ i}$). The overhead imposed in the profiled program on each monitoring/evaluation/migration cycle is given by:

$$overhead = \sum_{j} t_{ovj} + t_{migration\ i} \tag{1}$$

Note that Equation 1 does not include either $hndl_{intj}$ or $t_{evaluation\ i}$ which are computed by the monitoring thread and will be masked by the monitored program execution. Therefore, only the interruption overheads and the migration time are susceptible of slowing down the monitored program. Hence, the runtime consumed by the monitoring thread is:

$$t_{monitor} = \sum_{j} hndl_{intj} + t_{evaluation\ i} \tag{2}$$

Note that the rest of the time the monitor is idle.

Despite the implicit overheads of the interruption handling, processing and migrating times, as well as the cache and TLB flushes, an adequate page allocation on each cycle i is expected to amortize these costs by the locality improvement achieved. In fact, accessing pages remotely in a NUMA system can be very expensive, so the difference between the cost of fetching a page to a local cell and the gain achieved by accessing the page locally compensates the migration process. The following section describes the page migration software infrastructure developed.

3.1 Architectural Design

A controlling process (monitoring thread) using `Perfmon2` can simultaneously monitor several threads. To do so, it waits on multiple contexts at the same time, as shown in Figure 2. A context can only be attached to one thread at a time. Therefore, there must be as many contexts as monitored threads. Taking into account this feature, our page migration infrastructure comprises the modules depicted in Figure 3:

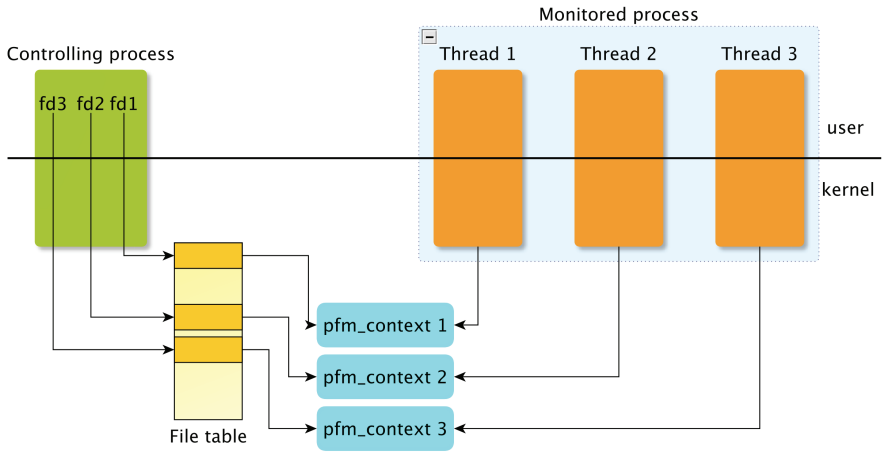


Fig. 2 A monitoring thread waiting on multiple contexts.

- A ***monitoring thread*** is allocated in a *monitoring core*. It is used exclusively to manage the sampling process, information retrieval and page migration decisions. It waits on as many contexts as monitored threads there are at a given time in a dynamic way.
- A ***multithreaded application*** where each *application thread* runs on a different core. A `Perfmon2` context is created and associated to each thread. It retrieves the information provided by the PMU of each core.
- A ***sampling buffer***, attached to each context, stores the samples obtained by each PMU. Its sampled data are processed by the monitoring thread to update a ***sample page map***, which consists of sampled page addresses accessed per monitored thread as well as other additional information collected by the counters.

3.2 Functional Design

The algorithm of the *monitoring stage*, executed by the monitoring thread, is presented in the flow diagrams of Figure 4. It comprises a main function (a) and an interrupt handler (b). The main function configures the monitoring process and launches the monitored process. The interrupt handler attends the interruptions raised by the threads of the monitored process. The algorithm follows these steps:

1. The program receives the name of the application to monitor.
2. An asynchronous notification is installed. When an interruption arrives, the `overflowHandler` function is executed.
3. After initializing the *libpfm* library, a child process is created. Its first action is to execute a `ptrace(PTRACE_TRACEME)` system call, whereby the parent

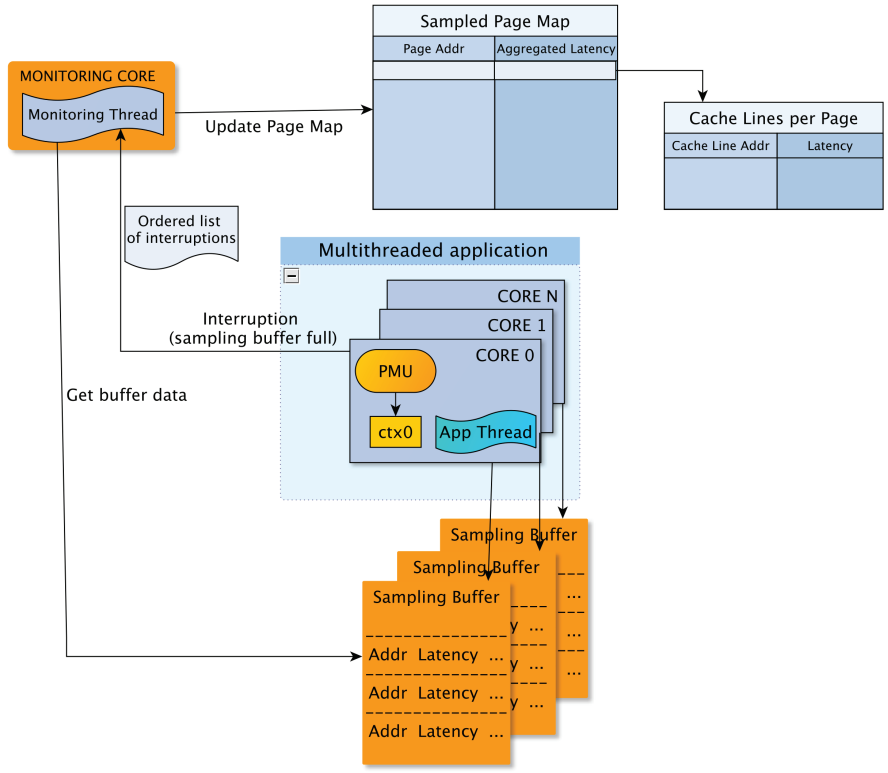
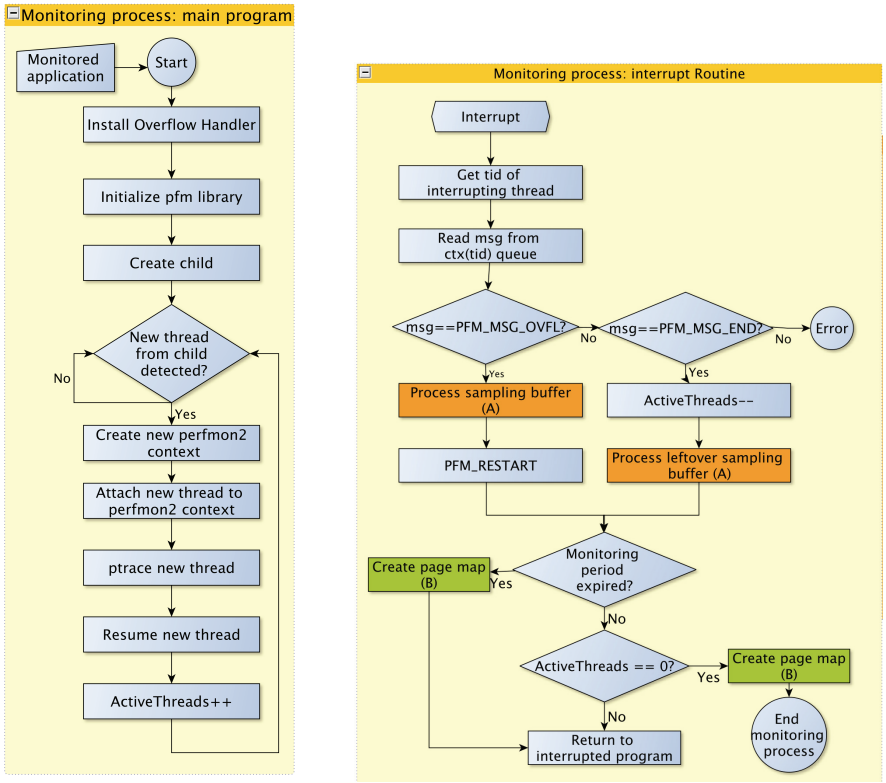


Fig. 3 Sampling section from our page migration infrastructure.

process can observe and control the execution of the child. This action is followed by an `exec()` system call to start running the application to monitor.

4. The code enters then in a loop in which a notification is received every time a new thread is created by the child process. If this happens, the monitored thread will have previously been stopped by the `ptrace(PTRACE_TRACEME)` call.
5. A new `Perfmon2` context is created and attached to the *tid* of the new thread. A sampling buffer will be allocated and associated to that context. The buffer is configured to send a notification, via the associated file descriptor, every time it is full.
6. Finally, a `ptrace(PTRACE_ATTACH)` system call begins tracing the new thread before it is resumed to start running normally.

This process allows detecting dynamically any new threads created or destroyed during the monitored application's life cycle. Note that the sampling buffer of each context is configured to send a notification every time it overflows. The `overflowHandler` function manages that notification, following the steps in Figure 4(b):



(a) (b)

Fig. 4 Flow diagrams: main function (a) and interrupt handler (b) of the monitoring tool.

1. First, the `Perfmon2` context of the thread that raises the interruption is accessed.
2. `Perfmon2` stores buffer notifications as messages in a queue. If the message indicates that the buffer got full and overflowed (`PFM_MSG_OVFL`), the buffer is processed (*subprocess A*) and the context is restarted to get ready to keep on storing samples. If the message indicates that the thread has finished, then the leftover samples in the buffer are read, and the thread is stopped being traced.
3. When either the monitoring period expires or the monitored program finishes, a *page map* is created or updated with fresh information from the just last read sampling buffer (*subprocess B*).

The *subprocess A* in Figure 4(b) processes the sampling buffer each time a notification arrives to the monitoring tool:

1. Each `Perfmon2` context has a pointer to its associated sampling buffer's header address. The number of stored samples in the buffer is obtained from the buffer header, so that the samples can be iterated.

2. For each sample, the memory address, as well as its associated latency, are retrieved.
3. The cache line is identified from each address.
4. Cache lines are stored or updated in a list associated to the context. Therefore, there will be one list per monitored thread to keep control of each thread accesses.
5. This loop continues until all the samples in the buffer have been read.
6. When sampling is restarted via `PFM_RESTART`, the sampling buffer is marked as empty.

The *subprocess B* in Figure 4(b) creates the page map once all cache lines have been accessed as follows:

1. The list of existing contexts is iterated. For each context, each cache line accessed is read.
2. The page address to which each cache line belongs is identified and appended to the list (which is actually a binary search tree) associated to that context.
3. If the page already exists in the list, the cache line is added to that page. Otherwise, a new page will be created.
4. This process is repeated for all cache lines accessed per context and for each context.

The creation of the page map is the last step of the *monitoring stage*. Next, the *evaluation stage* begins. In this stage, the collected data are processed by a migration algorithm and, then, the memory pages chosen by the algorithm are actually migrated. As case study an access-based migration algorithm is integrated into our tool and evaluated in Section 5. Previously, a set of operating tests carried out to evaluate our page migration infrastructure are presented in the next section.

4 Validation of the Page Migration Infrastructure

We carried out our tests in the FinisTerrae supercomputer [4]. Each node in this machine comprises two SMP cells. Both cells in a node share the main memory in a NUMA configuration. Each cell is composed, in turn, of two sockets with two 1.6 GHz-DualCore Intel Itanium2 Montvale processors each. Hence, there are four dual core processors per cell, which makes sixteen cores per node. Each socket is connected to a cell controller by a 6.8 GB/s bus. Each cell controller connects directly to a 64 GB local memory and to the other cell controller through a crossbar. Our system uses a SuSE Linux ES with a 2.6.29.6 Linux kernel. In our experiments, the main event sampled is "`DATA_EAR_CACHE_LAT4`" which corresponds to all data cache misses with latency higher than 4 cycles. According to the Itanium2 cache latency tables [6], this includes accesses that miss in the L1 data cache and hit L2. To validate our proposal, two performance studies were carried out, detailed in the following sections.

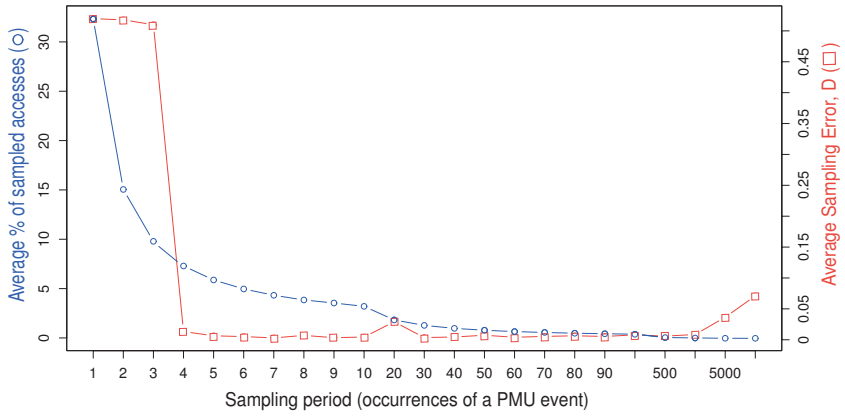


Fig. 5 Average sampling error (D) for 2 threads. The Sampling period is measured as the number of occurrences of an event before a sample is taken.

4.1 Reliability of sampling

An experiment was conducted to compare how representative our sampling method is of all memory accesses. A distance metric D, similar to the one described in [20], was used. If we consider the *ratio of accesses* from a processor to the total number of accesses, D measures the difference, or error, between ratios of accesses for a given program. An OpenMP program that accesses randomly positions of an array was used. A variable C_p counts the number of accesses by processor p. Another variable, C_a, counts all accesses performed by all processors. The program is monitored by our page migration infrastructure, using EARs to sample the accesses to the array by each processor (S_p) and the accesses by all processors (S_a). The ratios of actual and sampled accesses by each processor are given by $R_{all} = \frac{C_p}{C_a}$ and $R_{sample} = \frac{S_p}{S_a}$, respectively. D indicates how much a set of accesses deviates from another set and is defined as $D = \frac{|R_{sample} - R_{all}|}{R_{all}}$. Therefore, the closer the distance to 0 is, the more representative the set of sampled values is to the set of all accesses.

The experiment was performed considering 2, 4, 8 and 16 threads on a FinisTerrae node and different sampling periods, defined as the *number of occurrences of an event before a sample is taken*. As an example, results for two threads are shown in Figure 5. They combine the average distance of every processor (in red) with the percentage of sampled accesses (in blue) for every case. All the considered cases show a similar pattern in which a low sampling period (1 to 5, approximately) obtains a high number of samples, but at the cost of a noticeable distance error. From a sampling period of 5, the values of D decrease dramatically and stay steady until it increases again for values higher than 1000, due to the fact that the number of samples is too low to characterize accurately the monitored program. The choice of an appropriate sampling

period must attend at a low value of D and, simultaneously, a sufficient number of samples. Regarding D, an appropriate choice for any number of threads may be any sampling period between 5 and 1000. However, the choice of the percentage of sampled values is far from being trivial. Indeed, at first sight, a sensible range of sampling periods could be any between 6 and 10. These periods are enough to obtain between 5% and 10% of sampled values. However, this experiment evaluates a constant monitoring stage. The overhead associated to the evaluation stage, in which the sampled data are processed, is not considered. Empirically, a period of 1000 has been verified to be enough to acquire a representative number of samples with a small overhead, so that was the sampling period used in the experiments presented in the remaining sections. The monitoring period chosen has been 1 second, which also proved to obtain the best tradeoff overhead-number of useful samples.

4.2 Migration Throughput

Page migration is performed using the `move_pages()` system call [11]. Whereas it permits page migration in user-space, it suffers from a limited performance. Each invocation has a large initialization overhead and, among other side effects, flushes the TLB. Although `move_pages()` was fixed in kernel 2.6.29 to have a linear complexity [5], its overhead to migrate large buffers might constrain the final performance of a page migration policy. In our tool, the evaluation stage is run every time the monitoring stage ends, applying a migration policy to evaluate which of the sampled pages must be migrated. The larger the buffer of pages to migrate is, the higher the potential migration overhead. This section evaluates and quantifies the throughput of `move_pages()` regarding the amount of pages transferred.

The study has been carried out on a FinisTerrae node. We have used a code in C that allocates P pages in a buffer, places a thread in a given core and calls `move_pages()` from that thread. The throughput of the buffer migration from one node to another was measured. Since both cells are symmetrical, and considering that there are no other processes interfering, it seems reasonable to pose that the time to migrate a page from Cell 0 to Cell 1 must be the same as from Cell 1 to Cell 0. Additionally, this study has evaluated whether there is any difference in performance ordering the migration from either a local or a remote cell. Figure 6 shows the throughput for different number P of pages and migration strategies. The legend *"Data in cell X. Thread in cell Y. Migration from X to Z"* means that the data to migrate is initially allocated in cell X and the thread that invokes `move_pages()` is in cell Y. The pages to migrate are in cell X and are moved to cell Z.

The figure shows two different trends. In two out of the four cases, in which the thread that invokes `move_pages()` is in the same cell as the data, the throughput is identical to each other and higher than the opposite cases, specially for 32 pages (maximum use of the L1 DTLB). In all experiments, the peak throughput is achieved for 128 pages (number of entries of the L2

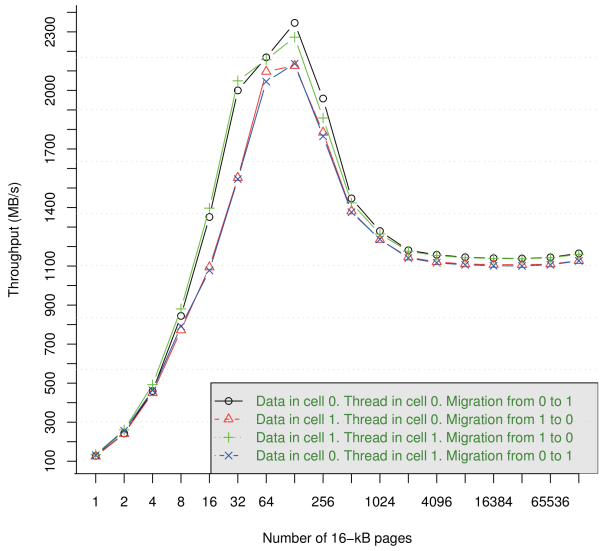


Fig. 6 Migration throughput between cells.

DTLB). From then on, the performance falls until a buffer size of 2048 pages is reached, staying approximately steady from that value on. In preliminary tests the number of pages to migrate was practically always higher than 1024 pages per monitoring period, coinciding with the steady region in Figure 6. There, the difference of throughput is just about 3%. Taking into account that in the tests the number of pages migrated on each monitoring period was in the range between 1024 and 16384 pages, the migration time can be assumed to fall, approximately, in the range between 14 and 229 milliseconds at most. It is not straightforward to state whether such an overhead is affordable or not, since it will depend on the number of page migration actions that occur during the execution of the monitored program, and how much these and the remaining overheads are compensated thanks to the improvement achieved by those migrations.

5 Case Study: an Access-based Migration Algorithm

Next, as a case study, a page migration algorithm has been implemented and integrated into our hardware counter-based infrastructure. Consider a generic NUMA node whose memory modules are arranged in a hierarchy with two latency levels: $|G|$ *groups*, interconnected to each other at the same distance (understood as the same latency) and $|G_j|$ *cells* inside each group, also at the same distance from each cell inside its own group. The proposed migration algorithm takes into account uniquely the number of times a page is accessed from each cell during the execution of a program. Note that, because of the use of the hardware counters, only incomplete information from the sampled data,

provided by our page migration framework, is used. Assuming that it is able to sample a representative number of accesses from each page, our algorithm can be formally stated as follows:

Let us suppose that our system is composed of a set of $|G|$ groups $G = \{G_1, G_2, \dots G_{|G|}\}$. A group G_j is composed of $|G_j|$ cells $G_j = \{C_{j1}, C_{j2}, \dots\}$, with $1 \leq j \leq |G|$.

Let $P = \{p_1, p_2 \dots p_p\}$ be the set of pages accessed by a program during its execution. Let us suppose that, during a monitoring period of our framework, the page p_i resides in the local memory of cell $C_{ks} \in G_k$.

Let $a_i^{jm} \geq 0$ be the total number of sampled accesses to p_i from the threads running in cell $C_{jm} \in G_j$, being $1 \leq j \leq |G|$ and $1 \leq m \leq |G_j|$. The total number of accesses to p_i from all the cells in group G_j is then

$$\alpha_i^j = \sum_{m=0}^{|G_j|} a_i^{jm} \tag{3}$$

Let h and d be the indexes such that

$$\begin{aligned} \alpha_i^h &= \max_{1 \leq j \leq |G|}\{\alpha_i^j\} \\ a_i^{hd} &= \max_{1 \leq m \leq |G_h|}\{a_i^{hm}\} \end{aligned} \tag{4}$$

Then, the page p_i is migrated to the cell $C_{hd} \in G_h$. Notice that if $h = k$ and $d = s$, p_i is not migrated (because p_i is in the memory of C_{ks}), and that if $h = k$ but $d \neq s$ an intra-group migration is carried out.

In other words, our algorithm evaluates the number of accesses to each page from each cell. After a monitoring period, each page is migrated to the cell which most accessed it under the assumption that, if a page has been accessed most from a cell during a monitoring period, it will keep on doing it in further periods. Note that this strategy can be generalized for systems with a more sophisticated memory hierarchy.

To evaluate the effectiveness of our proposal, a series of tests were conducted on a FinisTerrae node. The migration algorithm was particularized for $|G| = 1$ groups and $|G_j| = 2$ cells. Note that the total number of sampled accesses to a page p_i from a given cell, defined as a_i^{jm} in Equation 3, are provided by the EAR hardware counters.

Eight out of the nine OpenMP NAS Parallel benchmarks [7] v3.3, C size, were used. Namely, BT, CG, FT, IS, LU, MG, SP and UA. EP was not evaluated since it does not have significant sharing of data. The goal of the experiment was to compare the execution time of each benchmark with and without our page migration tool on a dedicated and a multiprogrammed environment. To do so, each benchmark was initially executed with 4, 8, 12 and 16 threads without any specific constraints, allowing the OS scheduler to allocate and migrate the threads on any core. Their wallclock execution time was measured. Note that thread-to-core binding was not used, since one of the aims of our proposal is that no knowledge about the monitored application and hardware platform is required by the users. In fact, if an advanced user had a deep

Benchmark	*Speedup*		*Benchmark*	*Speedup*	
	Na_D	Na_M		Na_D	Na_M
BT.C			*LU.C*		
4 *threads*	**0.2**	**0.2**	4 *threads*	**0.2**	**20.6**
8 *threads*	-5.6	**4.8**	8 *threads*	-5.5	**3.4**
12 *threads*	-2.7	0.0	12 *threads*	-3.2	**3.3**
16 *threads*	**0.1**	**0.6**	16 *threads*	**10.9**	**2.8**
CG.C			*MG.C*		
4 *threads*	-14	-3.6	4 *threads*	**8.8**	**0.3**
8 *threads*	-30.1	-2.2	8 *threads*	**0.1**	**0.9**
12 *threads*	-26.5	-5.5	12 *threads*	**1.2**	**6.4**
16 *threads*	-32.1	-0.3	16 *threads*	-2.3	**6.4**
FT.C			*SP.C*		
4 *threads*	-9.2	-7.4	4 *threads*	-2.4	**7.9**
8 *threads*	-17.1	-5.5	8 *threads*	-5.2	**7.7**
12 *threads*	-0.8	-4.1	12 *threads*	-0.2	**6.6**
16 *threads*	**0.3**	-1.5	16 *threads*	**2.1**	**5.5**
IS.C			*UA.C*		
4 *threads*	-3.7	**0.4**	4 *threads*	-0.5	**1.2**
8 *threads*	-1.2	-2.3	8 *threads*	-2.0	**1.9**
12 *threads*	-1.9	**0.3**	12 *threads*	-1.2	**1.6**
16 *threads*	-0.7	-0.8	16 *threads*	-1.5	**0.6**

Table 1 Speedup (%) of our page migration strategy with respect to the OS first-touch policy. Tested in a dedicated (Na_D) and a multiprogrammed (Na_M) environment.

knowledge of the application memory pattern, (s)he could pin threads that are prone to share data to neighbour cpus in the same socket or cell. The data access inside a cell is SMP, which is faster than accessing a remote memory. This will probably reduce the need for a page migration algorithm, but implies that the user knows the memory access pattern of the application. Remember that the overhead associated to the monitoring stage is considered and must be amortized.

5.1 Evaluation in a dedicated environment

In this scenario each thread will be scheduled in a free core, so few or no thread migrations or preemptions are expected. Since the benchmarks are already optimized, the first-touch policy should be sufficient to execute the benchmarks efficiently. Therefore, only little improvements are expected.

Column Na_D in Table 1 shows the speedup of the access-based migration algorithm for each benchmark compared to the first-touch policy. Results show that the dynamic migration technique obtains relevant improvements only for LU and MG, whereas it is clear that the rest of them are negatively affected. Particularly bad cases are CG and FT, in which the performance is noticeably worse –about a -17% for FT and a -32% for CG in the worst cases–. These results are discussed in Section 6.

5.2 Evaluation in a multiprogrammed environment

A more realistic scenario is the one in which the hardware resources are shared among user applications. In this case, threads may be preempted or migrated at any moment by the OS scheduler. Hence, the chances that a thread is migrated far from its dataset are higher than in a dedicated environment. Additionally, the presence of numerous programs accessing memory simultaneously will cause a higher bus load. In these situations, our page migration strategy is expected to achieve a better performance.

For this experiment we wrote a parallel program called *NomadicNoise* to simulate, in a controlled way, a multiprogrammed environment in which a second program shares resources with the NAS benchmarks. *NomadicNoise* allocates an array locally and accesses it during a given period. After the period expires, the array is freed and the program migrates its threads to a new cell, resuming the process there. For our tests, *NomadicNoise* was executed using 7 threads, a 10 GB array and a 15-second period. The interference of this program was expected to force some threads of the NAS benchmarks to lose their affinity, being moved to other cells by the OS scheduler. Two cases were anticipated when the scheduler tried to reallocate a thread: a first one when there are enough available cores to migrate the threads (NAS using ohly 4 or 8 threads), and when there are not (12 or 16 threads). In this last case, some of the NAS threads will be preempted until they get a new time slot. As in previous tests, there was no binding of threads to processors or any other intervention that alters the behavior of the scheduler. The monitoring thread was configured to share the cores in which the NAS benchmark was running.

Column Na_M in Table 1 shows the speedup of each benchmark for the original case and the page migration technique in the multiprogrammed environment. There is a general performance improvement compared to the native first-touch policy. BT, LU, SP and UA were the most benefited by the migration policies. As in the dedicated environment, FT and CG were slowed down (-7.4% and -5.5% in the worst case, respectively), although the performance decrease was noticeably lower than in a dedicated environment. The results also confirm a better behavior when there are available cores where threads can be migrated (4 and 8 threads) than those cases in which the threads are preempted (12 and 15 threads). See later discussion in the next section.

6 Discussion

In order to discuss our findings we must go through the characteristics of some of the benchmarks used. BT, SP, LU and UA are simulated *Computational Fluid Dynamics* (CFD) applications with a high rate of data usage and computation. BT (*Block Tri-diagonal*) solves a discretized version of unsteady, compressible Navier-Stokes equations in three spatial dimensions. The factorization used decouples data in the x, y and z dimensions. The resulting systems are Block-Tridiagonal and are solved sequentially along each dimen-

sion. Note that BT touches the data pages in the beginning of the program in favor of the x and y dimensions, so only the z dimension is likely to trigger data movement. The OpenMP version is particularly well tuned to reduce the memory usage and improve the cache performance which explains why, in general, there is no performance improvemement in the Na_D column. On the other hand, improvement in the multiprogrammed environment depends very much on how threads are preempted or migrated in the presence of *NomadicNoise*. Only when *NomadicNoise* pushes enough BT threads to the same cell and our page migration method makes their datasets follow, a gain can be expected. This explains why only the 4 and 8-thread cases produce a positive speedup. For 12 and 16 threads, some of them will be preempted, given that there is no room for all of them in the other cell.

The iteration procedure for the SP (*Scalar Penta-diagonal*) algorithm is similar to BT, although the approximate factorization is different. Therefore, at first sight it may be expected that the performance scales for any number of threads in the multiprogrammed environment, whereas in BT does not. However, there are a couple of factors that justify the results. Firstly, BT is similar to SP in structure but there are several additional parallel procedure calls to resolve each dimension. Secondly, the number of iterations performed by SP is 400, double than BT for the same problem size. This gives more time for the page migration cost to be amortized.

LU also solves a system resulting from a finite-difference discretization of the Navier-Stokes equation in 3D, but does it by splitting the system into block lower and upper triangular systems. LU is implemented in parallel using pipelining. This is done by distributing the matrix data rows in blocks. Being a pipeline, thread $i+1$ has to wait for data calculated by thread i to be available. Once the pipeline is full, all threads work concurrently until they reach the end of the matrix. This means that, opposite to BT and SP, there is a continuous flow of data from one dataset to another during the execution of the benchmark. This type of code is likely to be benefited from dynamic page migration techniques to increase locality, as performance improves when a dataset is close to the previous one in the pipeline structure. Indeed, Table 1 shows a positive speedup in LU for all cases of the multiprogrammed environment. We noticed an important improvement as well in the dedicated environment for 16 threads. The reason for this is a non-optimal initial thread allocation by the first-touch policy. Once LU starts, each thread will read its neighbor datablock to do its own calculations. A good initial allocation will keep datablocks in the same cell. However, a poor allocation will require threads to access other datablocks in a remote cell, situation that can be relieved by our page migration method. This can happen as well for fewer threads, but the effect is not so noticeable because there are fewer datablocks to rearrange. Note that in this applications in which remote datablocks are required as read-only blocks, a page replication policy could even result in better performance. We are studying this possibility as a future work.

UA (*Unstructured Adaptive mesh*) features a dynamic and irregular memory access problem. As in most of problems with irregular and continually

changing patterns, a page migration policy can even become detrimental if a page keeps on being migrated back and forward. Despite all, our technique achieves a modest improvement in the multiprogrammed scenario, as the datasets are moved close to their threads when those are migrated.

The remaining benchmarks (CG, FT, IS and MG) are not applications but kernels that mimic the computational core of numerical methods used by CFD applications. Little or no improvement is found here. CG and FT are benchmarks in which the performance slow down is particularly noticeable. The former shows little movement of data –since about half of its address space are temporary data for the initialization of the program–, and an important number of L1/L2 cache misses due to the irregularity of its operation with sparse matrices [7]. FT is a computational-bound kernel that calculates a Fourier Transform using the FFT-based spectral method, with little data movement. IS and MG also show a low percentage of remote memory accesses (3% and 2% of the total bandwidth, respectively [9]) which make them difficult to get any improvement.

In those cases with such a high data locality the use of a migration policy, particularly in a dedicated environment, is detrimental to the performance, as we expected. In a multiprogrammed environment, the benefit of having datasets migrated to follow their threads does not compensate for the overhead of the process, given that these kernels run only for 20 iterations.

By and large, we observed that the OS scheduler, although it tends to spread threads across cells, does not follow a fixed policy to map threads to cores. For example, for 4 and 8 threads, some executions of the same benchmark had their threads evenly collocated in cores of both cells and some had all threads in the same cell. Only those executions whose threads were spread out in both cells could benefit of our page migration method improving the performance and overcoming the implicit overhead of the monitoring stage.

From our observations, some characteristics emerge as being representative of the kind of codes that can benefit from our proposal. So a user that wants to determine its applicability to his/her codes might want to consider the following recommendations:

- The monitoring and migration overhead must be masked by the speedup achieved by having pages close to the thread that references the data. Therefore the number of pages should be high enough to reach an acceptable migration throughput and should stay next to the referencing thread for long enough. According to our tests, this can be translated into two features: codes should not be extremely irregular and the memory usage must be large enough with relation to the system memory. A value that works fine in our experiments is in the range of, at least, 1/8 of the available memory.
- In a dedicated environment, codes with low locality can expect a performance improvement, since the pages will be adequately pushed to the requesting threads during the execution of the code.

– In a shared environment, a user can expect an important performance improvement as long as the memory usage is large enough, as mentioned in the first point.
– The problem size and, consequently, the run time are also important. Our tests were performed using the C size of the NAS benchmarks. Sizes D or E would have likely yielded better results, as the migrations have more time to be amortized.

7 Conclusions

This paper has introduced a profile-driven, user-level monitoring and page migration infrastructure for Linux. It relies on the PEBS hardware counters to obtain a sampled profile of the exact data addresses issued by an application. This tool attaches and samples the monitored program without modifying it whatsoever. In addition, it is flexible to support different migrating strategies. The page migration process is performed timely, and its overhead is overcome by the benefit of the data locality achieved. A series of tests has been carried out to evaluate its performance and reliability, exploring the benefits and drawbacks of this approach.

As case study, an access-based page migration algorithm has been implemented on top of our tool. Its effectiveness has been evaluated using the OpenMP parallel NAS benchmarks on a two-cell NUMA node. Two scenarios have been considered. First, a dedicated one, in which the main reason for non-local memory accesses is a poorly data allocation by the OS first-touch policy. We observed that data misplacing by the first-touch is more likely to happen for a high number of threads, giving more room for improvement. A maximum execution speedup about 11% was achieved, although the general improvement was scarce and irregular. The second scenario was a multiprogrammed environment in which applications share the hardware resources. In this case, pages were migrated more efficiently, achieving speedups up to 20.6%, in a more regular improvement trend.

We observed that, in a poor initial data allocation, threads can be (1) allocated far from their datasets or, (2) have their datasets close to them but happens that the data are tightly related to another thread's dataset. Our experience shows that, in a dedicated environment, there will typically be little data migration, enough to reach an equilibrium and minimize the number of remote accesses. In these situations the continuous overhead related to the monitoring and evaluation process may become higher than the little page migration needed, resulting in a performance decrease in some cases. However, in the multiprogrammed environment, in which thread migrations are often performed, the benefit of moving data to the same cell where a recently migrated thread was sent clearly compensates the overhead.

By and large, the infrastructure performed correctly, migrating pages in a reliable way when the use case algorithm stated to do it. In a multiprogrammed environment, this algorithm improved the performance, compared

to the standard first-touch policy of the Linux kernel, for codes with significant memory usage and not extremely irregular pattern access.

Our migration platform allows virtually any algorithm to be implemented on top of it. Future work will take advantage of the latency-related events provided by the EARs to study the effect of latency-based migration strategies in the memory accesses. We are also considering the possibility of making the migration decision process more automatic by pushing it to kernel level, so that a sensible choice would be a kernel module that might be enabled or disabled under certain criteria. We have also considered sustainability in other platforms: The monitoring standard now for hardware counters access in the x86-64 platform is *Perf_events*. However, access to x86-64 PEBS, the closest to the EARs hardware counters provided in IA-64, is still in an early development stage. In fact, the Perfmon2 main developer is collaborating with the Perf_events team for this purpose. We are convinced that, given the adequate kernel infrastructure to access hardware counters, our proposal may be applied to the x86-64 architecture, although the efficiency and reliability of PEBS for x86-64 remains to be tested.

References

1. W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA policies and their relation to memory architecture. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, 1991.
2. J. M. Bull and C. Johnson. Data distribution, migration and replication on a ccNUMA architecture. In *Proceedings of the Fourth European Workshop on OpenMP*, 2002.
3. S. Eranian. *The Perfmon2 Interface Specification. Technical Report HPL-2004-200R1.* HP Labs, February 2005.
4. Galicia Supercomputing Centre (CESGA). http://www.cesga.es.
5. B. Goglin and N. Furmento. Enabling high-performance memory migration for multithreaded applications on Linux. In *Proc. of the IEEE Int. Symposium on Parallel & Distributed Processing*, pages 1–9, 2009.
6. Hewlett Packard. *Dual-Core Update to the Intel Itanium 2 Processor Reference Manual*, 2006. Technical paper.
7. H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, 1999.
8. R. P. Larowe, Jr. and C. Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, Nov. 1991.
9. Z. Majo and T. R. Gross. Matching memory access patterns and data placement for NUMA systems. In *Proc. of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 230–241, New York, NY, USA, 2012.
10. J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In *Proc. of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 90–99, 2006.
11. move_pages manual. http://linux.die.net/man/2/move_pages.
12. D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A case for user-level dynamic page migration. In *Proceedings of the Int. Conf. on Supercomputing*, pages 119–130, 2000.
13. D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *Proc. of the Int. Conf. on Parallel Processing*, pages 95–, 2000.

14. D. S. Nikolopoulos, C. D. Polychronopoulos, T. S. Papatheodorou, J. Labarta, and E. Ayguadé. Scheduler-activated dynamic page migration for multiprogrammed DSM multiprocessors. *Journal of Parallel and Distributed Computing*, 62(6):1069–1103, 2002.
15. OpenMP: Simple, Portable, Scalable SMP Programming. http://openmp.org.
16. Perfmon2 monitoring interface and Pfmon monitoring tool. http://perfmon2.sourceforge.net.
17. J. Tao, M. Schulz, and W. Karl. Improving data locality using dynamic page migration based on memory access histograms. In *Proc. of the International Conference on Computational Science-Part II*, pages 933–942, 2002.
18. V. Thakkar. Dynamic Page Migration on ccNUMA Platforms Guided by Hardware Tracing. Master's thesis, Graduate Faculty of North Carolina State University, 2008.
19. M. M. Tikir and J. K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *Proc. of the ACM/IEEE conference on Supercomputing*, SC '04, pages 46–, 2004.
20. M. M. Tikir and J. K. Hollingsworth. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing*, 68:1186–1200, 2008.
21. X. Wang, X. Wen, Y. Li, Y. Luo, X. Li, and Z. Wang. A dynamic cache partitioning mechanism under virtualization environment. In *Proc. of the 11th International Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1907 –1911, june 2012.
22. K. M. Wilson and B. B. Aglietti. Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 98–107, 2001.