Hardware-Assisted Thread and Data Mapping in Hierarchical Multi-Core Architectures[1]

Eduardo H. M. Cruz, Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
Matthias Diener, Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
Laércio L. Pilla, Department of Informatics and Statistics – Federal University of Santa Catarina – Florianópolis, Brazil
Philippe O. A. Navaux, Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil

The performance and energy efficiency of modern architectures depend on memory locality, which can be improved by thread and data mappings considering the memory access behavior of parallel applications. In this paper, we propose IPM, a mechanism that analyzes the memory access behavior using information about the time the entry of each page resides in the Translation Lookaside Buffer (TLB). It provides very accurate information with a very low overhead. We present experimental results with simulation and real machines, with average performance improvements of 13.7% and energy savings of 4.4%, which come from reductions in cache misses and interconnection traffic.

CCS Concepts: •**Computer systems organization** → **Multicore architectures;** •**Software and its engineering** → **Main memory;**

General Terms: Architecture, Performance

Additional Key Words and Phrases: Thread mapping, Data mapping, NUMA, Cache memory, Data sharing, Communication

1. INTRODUCTION

As thread-level parallelism (TLP) increases in modern architectures due to larger numbers of cores per chip and chips per system, the complexity of their memory hierarchies also increases. Such memory hierarchies include several private or shared cache levels, and Non-Uniform Memory Access (NUMA) nodes with different access times. In these parallel architectures, the resource management plays a key role in the performance and energy consumption. It influences the data movement between cores, caches and main memory banks, which occurs when a core performs a memory transaction. In this context, the reduction of data movement is an important goal for future architectures to keep performance scaling and to decrease energy consumption [Borkar and Chien 2011; Coteus et al. 2011]. One of the solutions to reduce data movement is to improve memory access locality through *sharing-aware mapping* [Torrellas 2009; Feliu et al. 2012].

State-of-the-art mapping mechanisms try to increase locality by keeping threads that share a high volume of data close together in the memory hierarchy (*sharing-aware thread mapping*), and by mapping data close to where its accessing threads reside (*sharing-aware data mapping*). Thread and data mapping should be performed jointly [Terboven et al. 2008] for maximum improvements. Performance and energy efficiency are increased for three main reasons. First, cache misses are reduced by decreasing the number of cache line invalidations on shared data [Martin et al. 2012] and by reducing the replication of cache lines on multiple caches [Chishti et al. 2005]. Second, the locality of memory accesses is increased by mapping data to the NUMA node where it is most accessed [Diener et al. 2014; Ribeiro et al. 2009]. Third, the usage of interconnections in the system is improved by reducing the traffic on slow and power-hungry interchip interconnections, using more efficient intrachip interconnections instead [Diener et al. 2014]. Failing to identify an application data sharing behavior can lead to poor mappings that reduce performance and energy efficiency.

We propose a mechanism called *Intense Pages Mapping (IPM)*, an online mechanism that analyzes the memory access pattern of parallel applications using information about the residency time of each page table entry in Translation Lookaside Buffers (TLBs). The information about the time that a page table entry resides in the TLB is very relevant because it indicates the affinity between the page and the threads accessing it. In this way, IPM identifies pages that are *intensely* accessed by one NUMA node, along with which threads access these pages. This paper makes the

[1]New Paper, Not an Extension of a Conference Paper.

following contributions:
• Our proposed metric, TLB residency, has a higher accuracy than previous mechanisms because it has access to more accurate information about memory access behavior.
• IPM can be implemented natively in architectures with software-managed TLBs, and with little additional hardware in architectures with hardware-managed TLBs.
• IPM allows the operating system to perform sharing-aware thread and data mapping during the execution of an application in a non-intrusive way with no prior knowledge of application behavior.

We demonstrate the benefits of IPM in a wide variety of applications using simulation and real machines. In a native implementation in a real machine, execution time was reduced by an average of 13.7% (up to 39%). Energy efficiency was improved by an average of 4.4% (up to 12.2%).

2. RELATED WORK

In this section, we describe several related mapping mechanisms, organizing them in mechanisms that perform only data or thread mapping, and mechanisms that perform both together.

2.1. Data Mapping Only

Traditional data mapping strategies, such as *first-touch* and *next-touch* [Löf and Holmgren 2005], have been used by operating systems to allocate memory on NUMA machines. In the case of first-touch, pages are not migrated during execution. Next-touch can lead to excessive data migrations if the same page is accessed from different nodes. The NUMA Balancing policy [Corbet 2012b] was included in more recent versions of Linux. In this policy, the kernel introduces page faults during the execution of the application to perform lazy page migrations, reducing the number of remote memory accesses. A previous proposal with similar goals was AutoNUMA [Corbet 2012a].

Marathe et al. [2010] present an automatic page placement scheme for NUMA platforms by tracking memory addresses from the performance monitoring unit (PMU) of Itanium. Their work requires the generation of memory traces to guide data mapping for future executions of the applications, which may lead to a high overhead [Barrow-Williams et al. 2009]. A similar technique is used by Marathe and Mueller [2006] to perform data mapping dynamically. They enable the profiling mechanism only during the beginning of each application due to its high overhead, losing the opportunity to handle changes in the rest of the execution. Tikir and Hollingsworth [2008] use UltraSPARC III hardware monitors to guide data mapping. Their proposal is limited to architectures with software-managed TLBs, while IPM targets architectures with either hardware-managed or software-managed TLBs.

Carrefour [Dashti et al. 2013] is a mechanism that uses sampling to detect page usage. Their sampling mechanism makes use of instruction-based sampling to identify memory addresses accessed by each thread. However, every delivered sample requires an interrupt, so processing samples at a high rate becomes very costly in Carrefour. Due to its overhead, the authors restrict the mechanism to 30,000 pages, which limits its use to applications with a low memory usage. Ogasawara [2009] proposes a data mapping method that is limited to object oriented languages that use garbage collection. On the other hand, IPM works regardless of the programming language or API.

Piccoli et al. [2014] propose a technique named Selective Page Migration (SPM) that automatically includes instrumentation code at compile time and migrate pages at run time. In the compiler, their proposal analyzes parallel loops to identify their memory access behavior. This behavior is then used to migrate pages to the nodes that most access them during the execution of a loop. Although using the compiler for analysis is interesting, it is only applicable to applications with simple parallelization patterns.

The aforementioned mechanisms do not detect the sharing pattern between threads, losing the opportunity to improve data sharing between them. Additionally, data mapping alone is not able to reduce remote memory accesses effectively in pages accessed by more than one thread, since these threads may be mapped to cores on different NUMA nodes.

2.2. Thread Mapping Only

The usage of the instructions per cycle (IPC) metric to guide thread mapping is evaluated in Autopin [Klug et al. 2008]. Autopin does not detect the sharing pattern, only verifies the IPC of several mappings fed to it and executes the application with the thread mapping that presented the highest IPC. Similarly, Blackbox [Radojković et al. 2013] selects the best mapping by measuring the performance of 1000 random mappings. These methods seldom converge to an optimal mapping because the number of possible mappings is exponential in the number of threads.

Zhuravlev et. al [2012] surveyed the state of the art on thread scheduling algorithms to handle resource sharing and contention. Many algorithms used metrics related to information from last level cache miss rates and IPC to detect scheduling problems and estimate performance degradation [Zhuravlev et al. 2010]. Such statistics do not accurately represent sharing and data access patterns, while IPM has direct access to the memory addresses being accessed.

Azimi et al. [2009] map threads based on information from the hardware counters of Power5 processors that sample the memory addresses resolved by remote caches. Accesses resolved by local caches are not considered, generating an incomplete sharing pattern. When only thread mapping is performed, the locality of memory accesses in NUMA architectures cannot be improved.

2.3. Joint Thread and Data Mapping

A library called ForestGOMP is introduced in [Broquedis et al. 2010a]. This library integrates into the OpenMP runtime environment and gathers information about the different parallel sections of the applications. ForestGOMP only works for OpenMP-based applications. Also, ForestGOMP needs source code annotations to perform the mappings, which can be a burden to the programmer.

The kMAF affinity framework is proposed in [Diener et al. 2014]. It performs both thread and data mapping and gathers information from page faults. To increase its accuracy, kMAF introduces page faults during the execution of the application to identify which threads access each page. Each additional page fault causes an interrupt to the operating system, increasing the overhead.

Gennaro et al. [2016] describe the inaccuracy that comes from using page faults to a single page table to detect the sharing pattern of applications. They explain that the page fault caused by one thread can mask eventual accesses to the same page by other threads. The authors overcome this issue by creating additional thread-specific page tables, and using the collected sharing pattern to map threads and their working sets of pages to the same NUMA node. kMAF and Gennaro et al. generate mapping information based on a very small number of samples compared to IPM due to the overhead of the page faults, such that IPM can provide a higher accuracy.

2.4. Summary of Related Work

We can observe that most related work either perform thread or data mapping, but not both of them together. In the case of thread mapping mechanisms, they are not able to reduce the amount of remote memory accesses in NUMA architectures. On the other hand, data mapping mechanisms are not able to reduce cache misses or correctly handle the mapping of shared pages. The mechanisms we described that perform both mappings together have several disadvantages. ForestGOMP [Broquedis et al. 2010a] is online, but requires hints from the programmer to work properly. kMAF [Diener et al. 2014] uses sampling and has a high overhead when we increase the number of samples to achieve a higher accuracy. In general, other mechanisms rely on indirect statistics obtained by hardware counters, which do not accurately represent the memory access behavior of parallel applications. Several proposals require specific architectures, APIs or programming languages to work, limiting their applicability.

With the analysis of the related work, we can conclude that currently there is no online mechanism that can be applied to any shared memory based application, in which increasing its accuracy does not drastically increases its overhead. Our proposal fulfills this gap of the related work. We perform both thread and data mapping to achieve improvements in terms of cache misses, remote memory accesses and interconnection usage. We implement our proposal directly in the memory

management unit (MMU) of the architecture, which allows us to keep track of many more memory accesses than the related work in a nonintrusive way. In this way, we achieve a much higher accuracy in the detected patterns, while keeping a low overhead. The next sections describe our proposal.

3. DETECTING MEMORY ACCESS PATTERNS VIA TLB RESIDENCY TIME

One of the main challenges of sharing-aware online mapping mechanisms is to detect memory usage information. In this section, we first explain the traditional metrics used to collect such information. Afterwards, we introduce our proposed metric of using the time resided in the TLB. Finally, we perform some experiments to evaluate the accuracy and performance improvement of our proposed metric compared to the traditional ones.

3.1. Traditional Metrics

As online mapping mechanisms require memory access information to infer memory access patterns and make decisions, different information collection approaches have been employed with varying degrees of accuracy and overhead. Although capturing all memory accesses from an application would provide the best information for mapping algorithms, the overhead would surpass the benefits from better task and data mappings. For this reason, this is only done for offline mapping algorithms.

In order to achieve a smaller overhead, most traditional methods for collecting memory access information are based on sampling. Memory access patterns can be estimated by tracking page faults [Diener et al. 2014; Diener et al. 2015; LaRowe et al. 1992; Corbet 2012a; Corbet 2012b], cache misses [Azimi et al. 2009] or TLB misses [Marathe et al. 2010; Verghese et al. 1996], or by using hardware performance counters [Dashti et al. 2013], among others. Still, these sampling-based mechanisms present an accuracy lower than intended, as we show in Section 3.3. This is due to the small number of memory accesses captured (and their representativeness) in relation to all of the memory accesses. For instance, a thread may wrongly appear to access a page less than another thread because its memory accesses were undersampled. In another scenario, a thread may have few accesses to a page, while having cache or TLB misses in most of these accesses, leading to bad mappings in mechanisms based on cache or TLB misses.

3.2. Using TLB Residency as Metric

In this section, we explain our novel method to detect memory access patterns and data sharing based on the time a page table entry resides in the TLB. We refer to it as **TLB residency**. To understand the concept of using TLB residency to estimate the affinity between the threads and pages, we need to explain how the TLB works. A memory address, and thereby a page, can be accessed by an application only when its corresponding page table entry is cached in the TLB. While the application is executing code that is frequently accessing a page, the replacement policy of the TLB should keep its page table entry in the TLB. After the thread stops accessing this page for some time, the replacement policy of the TLB would select its page table entry for eviction, making room for an entry of another page the thread needs to access. Hence, a direct correlation between memory accesses and the time a page entry resides in the TLB can be observed.

Based on this explanation, the behavior of the TLB demonstrates that intensely accessed pages tend to stay for longer times in the TLBs of the cores that make these accesses. Therefore, using TLB residency, we can determine the memory affinity and use it to perform thread and data mapping. We intend to overcome the issues pointed out in the previous section, providing high accuracy with a low overhead. In Section 3.3, we show that the TLB residency metric estimates the memory access behavior of an application with higher accuracy than others. Also, this metric provides a smaller cost to implement in hardware than using the number of memory accesses itself, as the latter requires the addition of counters to every TLB entry [Tikir and Hollingsworth 2008].

3.3. Comparing Mapping Metrics

Following the hypothesis that a more accurate estimation of the memory access pattern of an application can result in a better mapping and performance improvements, we performed experiments

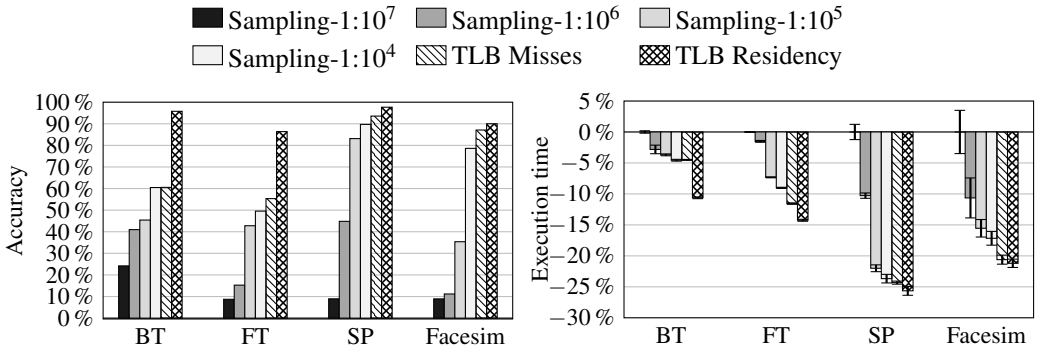
using the TLB residency, memory access sampling, and TLB misses as metrics to guide thread and data mapping. Experiments were run using the NAS parallel benchmarks [Jin et al. 1999] and the PARSEC benchmark suite [Bienia et al. 2008] on a two NUMA node machine with software-managed TLB (more information in Section 5.2). This machine was used because it provides accurate information about TLB misses. TLB residency and TLB misses were captured directly from the TLB miss handler in the operating system.

To analyze sampling based mechanisms, memory accesses were sampled using Pin [Bach et al. 2010]. We vary the number of samples from $1{:}10^7$ to $1{:}10^4$ of the whole for comparison. For instance, when using a sampling rate of $1{:}10^4$, a memory address is sampled at every 10,000 memory accesses. We save the memory address and the ID of the thread that performed the access. After Pin finishes executing the application, we generate a mapping such that a page will be mapped to the NUMA node that has the highest number of samples of the page. It is important to mention that a realistic sampling-based mechanism [Diener et al. 2015] uses a sampling rate of at most $1{:}10^5$ to avoid harming performance. Another sampling-based mechanism [Dashti et al. 2013] specifies that it samples once every $130,000$ cycles.

We use all memory accesses as a baseline for accuracy. The accuracy and execution time obtained with the different methods for benchmarks BT, FT, SP, and Facesim are presented in Fig. 1. To calculate the accuracy, we compare if the NUMA node selected for each page of the applications is equal to the NUMA node that performed most accesses to the page (the higher the percentage, the better). The execution time is calculated as the reduction compared to using $1{:}10^7$ samples (the bigger the reduction, the better).

The accuracy results in Fig. 1a show that the TLB residency provides estimations that are more accurate than all other tested methods. The differences are most noticeable with the BT and FT benchmarks, where the TLB residency shows over 85% of accuracy, while the other methods are all under 60%. One may also notice that accuracy increases with the number of memory access samples used, as expected. Nevertheless, most sampling mechanisms are limited to a small number of samples due to the high overhead of the techniques used for capturing memory accesses [Azimi et al. 2009; Diener et al. 2014], harming accuracy.

The execution time results in Fig. 1b indicate a clear trend: a higher accuracy in estimating memory access patterns translates to a shorter execution time. In this sense, the TLB residency metric provided the best performance among the tested methods. Although its performance was similar to the use of TLB misses for SP and Facesim, the effort to monitor both are very similar, such that improvements over the use of TLB misses can be achieved with negligible overhead.



(a) Accuracy of the final mapping (higher is better). (b) Execution time normalized to $1{:}10^7$ (lower is better).

Fig. 1: Results obtained using different metrics.

3.4. Summary

The results indicate that accurate information about the memory access behavior is important for a good mapping. Furthermore, a good mapping leads to better performance improvements. Our proposed metric provided the best accuracy and performance improvements, and has the same implementation complexity as the TLB misses metric. Given the benefits of using the TLB residency over other methods, the next section presents a mechanism for capturing the TLB residency and employing it for online data and thread mapping.

4. THE IPM MECHANISM

This section details our TLB residency-based mapping mechanism named **IPM**. In architectures with hardware-managed TLBs, IPM is implemented as an addition to the memory management unit (MMU) hardware, because the MMU has direct access to the TLB and all memory accesses. In architectures with software-managed TLBs, IPM can be implemented natively in the TLB miss handler of the operating system.

The text is organized as follows. It first presents an overview of IPM, followed by a discussion on the additional data structures required to capture memory access patterns. A detailed description of IPM is given next, and an account of its overhead completes the section.

4.1. Overview of IPM

An overview of the operation of the MMU, TLB and IPM is illustrated in Fig. 2. On every memory access, the MMU checks if the page has a valid entry in the TLB. If it does, the virtual address is translated to a physical address and the memory access is performed. If the entry is not in the TLB, the MMU performs a page table walk and caches the entry in the TLB before proceeding with the address translation and memory access. IPM modifies the behavior of the MMU during a TLB miss.

On every TLB miss, IPM stores a time stamp in the main memory in a structure called *TLB Access Table*. If a TLB entry must be evicted to store the new entry, IPM loads from the *TLB Access Table* the time stamp corresponding to the evicted TLB entry. By subtracting the loaded time stamp from the current time stamp, it knows how much time the evicted entry resided in the TLB. IPM uses this time difference to update sharing information in two structures: the *Page History Table* and *Sharing Matrix*. Finally, IPM notifies the operating system if a page migration could improve performance.

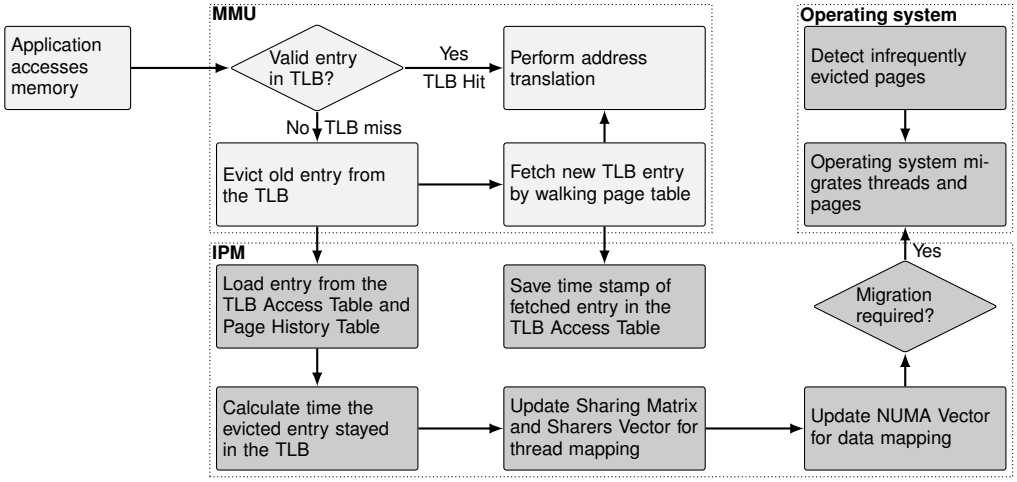


Fig. 2: Overview of the MMU, IPM and the operating system.

4.2. Structures Introduced by IPM

IPM adds new control registers to the architecture, and stores control data in the main memory. The following structures were added to the main memory:

• **Page History Table (PHT)** Contains memory access information related to each page. It stores 2 fields for each page: the Sharers Vector *(SV)* and the NUMA Vector *(NV)*. The Sharers Vector holds the identifier of the last threads that accessed each page. The NUMA Vector holds the affinity of each NUMA node to the page.

• **TLB Access Table (TAT)** For each TLB entry cached on all TLBs, it stores the virtual address of the page and the time stamp related to the moment when the corresponding TLB entry was fetched on a page table walk. It is used to compute the TLB residency used for mapping. The TLB Access Table also stores the ID of the thread that generated the TLB miss, kept by the operating system in a control register CR_{tid}.

• **Sharing Matrix (SM)** Stores the affinity between each pair of threads. The size of the Sharing Matrix is $nt \times nt$, where nt is the number of threads per parallel application supported by IPM, which can be configured by the operating system. This structure is used by the operating system to calculate the thread mapping.

Since each TLB entry has a corresponding entry in the TLB Access Table, IPM adds a unique static identifier to each TLB entry. This identifier is hardwired, not requiring any SRAM. The identifier is returned along with the physical address when the TLB is accessed, and is used as an index into the TLB Access Table (further details in Section 4.3.1).

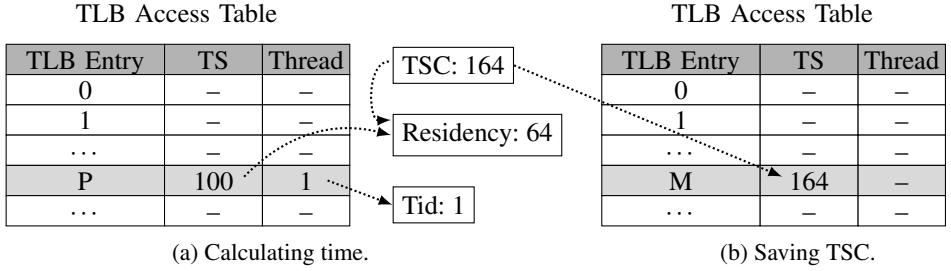
4.3. Detailed Description of IPM in the MMU

IPM is responsible for three tasks: computing the TLB residency for different page entries and threads; detecting the page sharing for thread mapping; and discovering the memory affinity of pages to NUMA nodes for data mapping.

4.3.1. Calculating the TLB Residency. The time resided in the TLB is the main information required by IPM to analyze the memory access pattern. For this end, the architecture must provide a counter that is incremented at a constant rate over time. Most architectures already provide this counter, such as the x86-64 architecture with its *time stamp counter* (TSC), which can be read using the `rdtsc` instruction. In this paper, we refer to this counter as TSC, although in other architectures it can have a different name. Whenever a TLB miss happens and it requires the eviction of an older entry, IPM fetches from the TLB Access Table (TAT) the entry corresponding to the evicted page, and saves in registers the resident time and thread ID. This is shown in Eq. 1 and 2, where *TLBEntryID* is the ID of the TLB entry introduced in Section 4.2.

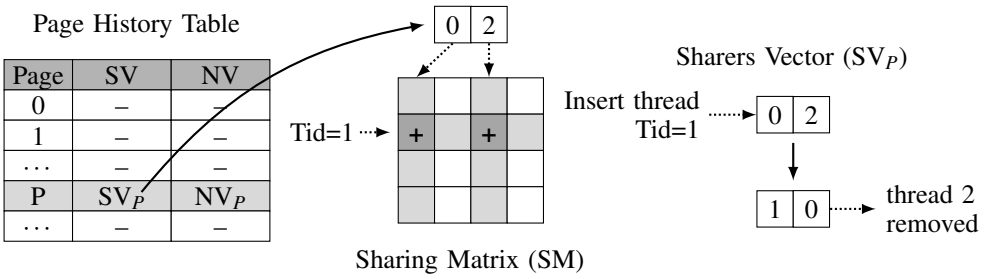
$$Residency \leftarrow (TSC \gg CR_{shift}) - TAT[TLBEntryID].ts \tag{1}$$

$$Tid \leftarrow TAT[TLBEntryID].tid \tag{2}$$



(a) Calculating time. (b) Saving TSC.

Fig. 3: Structures that are updated to calculate the time an entry resided in the TLB.



(a) Updating Sharing Matrix. (b) Updating Sharers Vector.

Fig. 4: Structures that are updated for thread mapping.

Since the time stamp has a clock cycle precision, we can discard the lowest bits of the resident time with a negligible accuracy loss. The operating system can configure the number of bits to be discarded using the CR_{shift} control register. After that, since the TLB Access Table entry of the evicted page is already stored in registers, IPM overwrites the TLB Access Table entry with the current time stamp, also shifted by CR_{shift}, and the thread ID of the current thread running in the core, indicated in the CR_{tid} control register.

An illustration on how the time an entry resides in the TLB is calculated is shown In Fig. 3a. In this example, page M generates a miss in the TLB, which causes an eviction of the TLB entry of page P, of thread 1. Consider for this example that CR_{shift} is 0, and thus no shift is performed in the TSC. To calculate the resident time, the current value of the TSC, 164, is subtracted by the TS field stored in the TLB Access Table entry of page P, 100. The final value, 64, is stored in the *Residency* register, and the thread ID, 1, is stored in the *Tid* register. Finally, the current value of the TSC, 164, is stored in the same TLB Access Table entry, which now belongs to page M, as in Fig. 3b.

4.3.2. Gathering Information for Thread Mapping. To perform thread mapping, we need to know the affinity between the threads. Threads with higher affinity should be mapped to cores close together in the memory hierarchy. To detect the affinity, IPM keeps the last threads that accessed a memory page in the Sharers Vector *(SV)* of the Page History Table. Whenever a TLB entry of page P is evicted, the corresponding Page History Table entry is read. After that, the Sharing Matrix *(SM)* is incremented by the resident time (Eq. 1) in row *Tid* (Eq. 2), for all the columns that correspond to an entry in the SV_P, as shown in Eq. 3. In this equation, $\#SV$, represents the number of elements in the Sharers Vector.

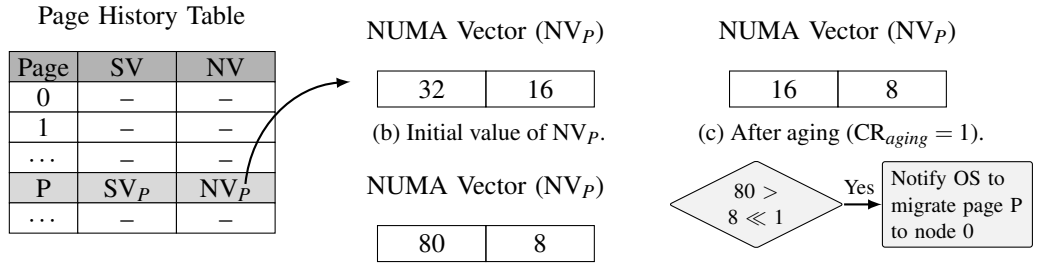
$$SM[Tid][SV_P[i]] \leftarrow SM[Tid][SV_P[i]] + Residency, \qquad 0 \leq i < \#SV \tag{3}$$

Each line of the Sharing Matrix is accessed by its corresponding thread only, minimizing the impact of coherence protocols. The operating system uses the Sharing Matrix as input to a thread mapping algorithm to generate a thread-to-core mapping, as explained in Section 4.4.4. Finally, IPM inserts thread *Tid* (Eq. 2) into the *SV* of the evicted page P by shifting its elements, removing its oldest entry.

$$SV_P[i] \leftarrow SV_P[i-1], \qquad 1 \leq i < \#SV \tag{4}$$

$$SV_P[0] \leftarrow Tid \tag{5}$$

We show how the information regarding the thread mapping is generated in Fig. 4, continuing the example of Fig. 3. To update the Sharing Matrix, as in Fig. 4a, first, the Page History Table entry of page P is read from memory to fetch the Sharers Vector SV_P. Since SV_P contains thread IDs 0 and 2, these two columns of the Sharing Matrix are selected. The row of the Sharing Matrix selected for



(a) Accessing NV$_P$ from the Page History Table.

(b) Initial value of NV$_P$.

(c) After aging (CR$_{aging} = 1$).

(d) After adding *Residency*.

(e) Checking for migration.

Fig. 5: Structures that are updated for data mapping.

update is specified by the *Tid* register, which is 1, as shown in Fig. 3. Due to this, cells $(1,0)$ and $(1,2)$ are incremented by *Residency*, which has the value 64 as in Fig. 3. We also insert *Tid* (1) in the Sharers Vector, as shown in Fig. 4b, and remove thread ID 2, which is the oldest element.

4.3.3. Gathering Information for Data Mapping. To perform data mapping, the operating system needs to know the affinity of each memory page to each NUMA node. In this way, the operating system can map pages to the nodes that most access them. To achieve this, IPM introduces the NUMA Vector *(NV)* in each entry of the Page History Table. The NUMA Vector has one counter per NUMA node. Each counter estimates the affinity of the corresponding page to each node. When a TLB entry corresponding to a page *P* is evicted and its Page History Table entry is loaded, IPM performs aging in each counter of the NUMA Vector, as in Eq. 6. In this equation, *#nodes* represents the number of NUMA nodes. Aging is important to detect changes in the access pattern to a page. The higher the value of the CR_{aging} control register, less aging is performed.

$$NV_P[i] \leftarrow NV_P[i] - (NV_P[i] \gg CR_{aging}), \qquad 0 \leq i < \#nodes \tag{6}$$

After aging, the NUMA Vector element related to the NUMA node where the thread is running is incremented by the resident time (Eq. 1) of the evicted TLB entry, shown in Eq. 7. In this equation, n represents the NUMA node where the thread that generated the TLB miss is running. It is important to note that the counters of the NUMA Vector are saturating counters, such that they keep their maximum possible value in case of overflow.

$$NV_P[n] \leftarrow NV_P[n] + Residency \tag{7}$$

After incrementing the counters, IPM evaluates if the evicted page is intensely used by a specific node and should be migrated. This is shown in Eq. 8, where n is the node of the core executing the thread, m is the node that currently stores page P, and CR_{mig} is a control register to help decide whether a migration is necessary. We left shift $NV_P[m]$ by CR_{mig} to avoid migrating pages that are accessed by several nodes. The higher the value of CR_{mig}, the more difficult it is to migrate a page. If the condition of Eq. 8 is true, then the page is intensely used by a node and IPM notifies the operating system to migrate page P to node n.

$$NotifyOS = \begin{cases} true & \text{if } NV_P[n] > (NV_P[m] \ll CR_{mig}) \\ false & \text{otherwise} \end{cases} \tag{8}$$

We demonstrate how the data mapping is calculated in the example shown in Fig. 5, continuing Figs. 3 and 4. In this example, the TLB evict happened in node 0, and page P is currently stored in node 1. First, the NUMA Vector NV$_P$ is fetched from the Page History Table entry of page P (Fig. 5a and 5b). After that, we perform an aging in NV$_P$, which in this case divides by 2 each of its elements because CR$_{aging}$ is 1 (Fig. 5c). Then, we add *Residency*, which contains the value 64, as shown in Fig. 3, to NV$_P$[0], since the TLB evict happened in node 0 (Fig. 5d). Finally, we evaluate if NV$_P$[0]

is higher than the double of $NV_P[1]$ (Fig. 5e). The value needs to be at least the double because CR_{mig} is 1 in this example. Since the condition is fulfilled, IPM notifies the operating system to migrate page P to node 0.

It is important to mention that the initial value of each counter of the NUMA Vector should not be zero. A zero value would result in too many migrations early during the execution. The initial value was defined as in Eq. 9, which is the maximum value not affected by aging, and is set by the operating system when initializing the page table entry.

$$NV_P[i] = (1 \ll CR_{aging}) - 1, \qquad 0 \leq i < \#nodes \tag{9}$$

4.4. Operating System Support for IPM

We explain some implementation details of the operating system support for IPM.

4.4.1. Dealing With Infrequently Evicted Pages. Depending on the application and architecture configuration, specially the page size and number of entries in the TLB, some pages can present very few TLB evictions (or even none). In this scenario, the procedure described in Section 4.3 alone may not be able to find a good mapping for these pages, as the described procedure uses TLB evictions as source of information. To overcome this issue, we introduce a thread in the kernel to periodically iterate over all entries in the TLB Access Table. Whenever an entry has been in the TLB Access Table for a long period, the kernel thread updates the contents of the Page History Table entry of the corresponding page using the exact same procedure described in Sections 4.3.1, 4.3.2 and 4.3.3. In this way, pages that have very few TLB evictions but reside a long time in the TLB can be correctly mapped by IPM.

4.4.2. Increasing the Supported Number of Threads. The operating system starts an application configuring IPM to support a certain number of threads using a control register. If the parallel application creates more threads than the maximum supported, the operating system can change this during execution. To do that, it must allocate a new Sharing Matrix and copy the values from the old one. It must also update the contents of all Page History Table entries to use the new number of bits per Sharers Vector entry. Since this is an expensive procedure, we recommend to avoid it by configuring IPM to support a large number of threads from the beginning. For all systems and applications we evaluated, configuring IPM to support 1024 threads was enough to avoid this procedure.

4.4.3. Selecting the ID of a New Thread. In some applications, although several threads can be created, only a few may be alive at a given time, since threads can finish their execution before the application ends. When a new thread is created, two main solutions can be adopted to select its ID: (i) new threads can always have a unique ID, (ii) or new threads can reuse the IDs of threads that finished their execution. If the first solution is adopted, the Sharing Matrix will have several rows and columns wasting memory. Therefore, we adopt the second solution. Before reusing a thread ID, the corresponding row and column of the Sharing Matrix must be reset, and any reference to that thread ID in the TLB Access Table and Page History Table must be deleted.

4.4.4. Performing the Thread Mapping. To calculate the thread mapping, the operating system applies a mapping algorithm in the Sharing Matrix. In this work, we used the EagerMap [Cruz et al. 2015] mapping algorithm, which receives the Sharing Matrix and a graph representing the memory hierarchy as input, and it outputs which core will execute each thread. The complexity of EagerMap is $O(N^3)$, where N is the number of threads. EagerMap is designed to work with symmetric tree hierarchies. It uses an efficient greedy strategy to group threads that share lots of data. It is important to note that other mapping algorithms that consider the data sharing behavior of the application, such as METIS [Karypis and Kumar 1998], Zoltan [Devine et al. 2006], or TreeMatch [Jeannot and Mercier 2010], could be used without any additional change to IPM.

One of the important characteristics of EagerMap is that it only changes the mapping of threads to cores if the sharing pattern changes significantly. In our experiments, this is done every 200ms. In

order to reduce the influence of old values in the Sharing Matrix, we apply an aging technique every time the mapping mechanism is called by multiplying all of its elements by 0.75. Values between 0.6 and 0.95 were also evaluated, but the results showed no sensitivity to this value.

4.5. Overhead

We implemented IPM in a way to handle only one TLB eviction at a time to keep the hardware modifications limited. If a TLB eviction happens while IPM is already handling another eviction, we do not consider this new eviction. Since TLB evictions are very frequent, we do not expect a significant impact on accuracy even when discarding a fraction of them, which is evaluated in Section 6.3. The overhead of IPM consists of storage space in the main memory, circuit area, and performance and power consumption overhead since it operates during application execution. We calculated the overhead with the configuration shown in Table II in Section 5, considering the *Xeon64* machine.

4.5.1. Memory Storage Overhead. The TLB Access Table, Page History Table and Sharing Matrix are stored in the main memory. Table I contains the storage overhead in Xeon64.

Each entry of the TLB Access Table requires 128 bits (16 bytes): 52 bits to store the page address (right shifted to remove the offset), 60 bits to store the shifted time stamp counter, and 16 bits to store the thread ID. There is one entry per TLB entry in the architecture. Eq. 10 shows how to calculate the space overhead of the TLB Access Table. The TLB Access Table required only 256 KBytes, which represents an overhead lower than 0.0002%.

$$overhead_{TAT} = \frac{TATentrySize \times \#TLBentriesPerCore \times \#Cores}{memSize} \tag{10}$$

Each entry of the Page History Table would require 16 Bytes (we need to roundup the size of the Page History Table entry to a power-of-two). Eq. 11 shows how to calculate the size of each entry (we fixed the number of Sharers Vector elements as 2). There is one entry per physical page. Eq. 12 shows how to calculate the overhead of the Page History Table. The Page History Table would require 512 MBytes of memory, which corresponds to a space overhead of 0.4% relative to the total main memory in Xeon64.

$$PHTentrySize = \#nodes \times NVentrySize + 2 \times \left\lceil \frac{log_2 \#threads}{8} \right\rceil \tag{11}$$

$$overhead_{PHT} = \frac{\frac{memSize}{pageSize} \times PHTentrySize}{memSize} = \frac{PHTentrySize}{pageSize} \tag{12}$$

Table I: Memory storage overhead of IPM in the Xeon64 machine (128 GBytes of main memory, 64 virtual cores and 256 entries per TLB).

Structure	Field	Space per entry	Total space	Overhead
TLB Access Table	Thread ID	16 bits		
(256 × 64 entries)	Time stamp	60 bits		
	Page Address	52 bits		
	Total	128 bits (16 Bytes)	256 KBytes	0.0002%
Page History Table	Sharers Vector	2 × 16 bits		
(33,554,432 entries)	NUMA Vector	4 × 16 bits		
	Unused	32 bits		
	Total	128 bits (16 Bytes)	512 MBytes	0.4%
Sharing Matrix	–	4 Bytes		
(1024 × 1024 entries)	**Total**	4 Bytes	4 MBytes	0.0032%

Each element of the Sharing Matrix has 4 Bytes. There is one element per pair of threads (we configured the Sharing Matrix to support 1024 threads per parallel application running at the same time). Eq. 13 shows how to calculate the overhead of the Sharing Matrix. The Sharing Matrix would require 4 MBytes, which represents an overhead lower than 0.0032%.

$$overhead_{SM} = \frac{SMentrySize \times \#threads^2}{memSize} \tag{13}$$

The total space overhead is lower than 0.5% of the total physical memory. In this configuration, IPM can track up to 1024 threads per parallel application running at the same time, and supports 4 NUMA nodes. In this case, the number of threads is limited by the Sharing Matrix size, since the Sharers Vector uses 16 bits, supporting up to 65536 threads. There are 4 unused bytes in the Page History Table. To support larger systems, we would need only to use these unused bytes, or use more space per Page History Table entry, allocating more NUMA Vector and Sharers Vector entries, and a larger Sharing Matrix.

In the context of multiple applications running, each one requires a separate Sharing Matrix, while only one TLB Access Table and Page History Table is kept for the entire system. In case a memory page needs to be swapped to the disk, two main strategies can be adopted to handle its Page History Table entry: (i) to save a backup copy of the entry in another memory area (or even together in the swap) to be restored when the page is returned to the memory; (ii) or to lose the information and detect the memory access pattern of the page from zero when the page is returned to the memory. As the storage space required by each entry is very small, we recommend the first solution.

4.5.2. Circuit Area Overhead. In architectures with hardware-managed TLBs, IPM is implemented in the MMU of each core. The area overhead can be estimated by analyzing the resources required for implementation. IPM requires the addition of 16 registers, 3 carry look-ahead adders and subtractors, 8 shifters, and 12 multiplexers in the MMU. 64 bit registers are used to store the positions in memory of the Sharing Matrix, Page History Table and TLB Access Table, while control registers and others store values using 16 bits or less.

An implementation of IPM optimized for area and power (by reusing some of the circuits for the different equations described previously) requires less than 30,000 additional transistors per core, which represents an increase of less than 0.02% in a current processor (12% relative to the MMU alone). Additionally, a performance-oriented implementation of IPM (with replicated resources) can be done by using less than double of the number of transistors of the other implementation.

4.5.3. Execution Time Overhead. The additional hardware of IPM is not in the critical path, since it operates in parallel to the MMU, such that application execution is not stalled while IPM is operating. Therefore, the time overhead introduced by IPM consists of the additional memory accesses to update the TLB Access Table, Page History Table and Sharing Matrix, which depend on the TLB miss rate. All memory addresses used by IPM are physical addresses and therefore do not require any translation. They proceed through the cache hierarchy as any other memory access, and could be found in the cache if it was recently used. To keep the overhead of these memory accesses low, IPM does not lock any structure before its update. Any possible race condition would not cause the application to fail, just a slight reduction in accuracy. On the software level, the operating system introduces overhead when checking the TLB Access Table for pages that are infrequently evicted from the TLB, when calculating the thread mapping, and when migrating threads and pages. The measured execution time overhead from both hardware and software are shown in Section 6.3.

4.5.4. Power Consumption Overhead. Since IPM includes additional memory transactions during TLB evicts, it introduces a power consumption overhead. Considering a Sharers Vector with 2 elements, there are 4 read and 4 write additional memory transactions per TLB evict. The amount of power consumption of such transactions will depend if they are resolved by the cache memory or need to go to the main memory.

5. EXPERIMENTAL METHODOLOGY

In this section, we describe the experiments we performed to evaluate IPM, including the environments and workloads employed. We used 3 environments to analyze our proposal in different scenarios. A full system simulator is used to evaluate the operation in machines with hardware-managed TLBs. We used a real machine with a software-managed TLB to prove that IPM can work not only in a simulator, but also in real machines. We performed a trace-based evaluation using real machines with hardware-managed TLBs to gain precise information regarding how our proposals affect the performance, cache misses, interchip traffic, and energy consumption in a real architecture, with more reliable results than in the simulator. Table II summarizes the parameters used.

5.1. Evaluation in a Full System Simulator

We implemented IPM in the Simics simulator [Magnusson et al. 2002], extended with the GEMS-Ruby memory model [Martin et al. 2005] and the Garnet interconnection model [Agarwal et al. 2009]. The simulated machine runs Linux 2.6.15 and has 4 processors, each with 2 cores, with private L1 caches, and L2 caches shared by all cores. Each processor is on a different NUMA node. Cache latencies were calculated using CACTI [Thoziyoor et al. 2008] and the memory timings from JEDEC [JEDEC 2012]. The intrachip and interchip interconnection topologies are bidirectional rings. We simulate the benchmarks with small input sizes due to simulation time constraints. To compensate for the small input sizes, we reduced the size of cache memories and TLBs accordingly,

Table II: Configuration of the experiments.

System	Parameter	Value
IPM	Structure sizes	SV: 2x 16 bits, NV: 4x 16 bits
	Sharing matrix	1024 threads, 4 Byte element size
	Control registers	CR_{shift} : 13, CR_{aging} : 7, CR_{mig} : 2
Pin	L1 TLB	64 entries, 4-way, shared between 2 SMT-cores
	L2 TLB	512 entries, 4-way, shared between 2 SMT-cores
Itanium	Processors	4x Intel Itanium 9030 (Montecito), 2 cores
	Caches/proc.	2x 16 KByte L1, 2x 256 KByte L2, 2x 4 MByte L3
	Main memory	2 NUMA nodes, 16 GByte DDR-400, 16 KByte page size
	Environment	Linux kernel 2.6.32, GCC 4.4
Xeon32	Processors	2x Xeon E5-2650 (SandyBridge), 8 cores, 2-SMT
	Caches/proc.	8x 32 KByte L1, 8x 256 KByte L2, 20 MByte L3
	Main memory	2 NUMA nodes, 32 GByte DDR3-1600, 4 KByte page size
	Environment	Linux kernel 3.8, GCC 4.6
Xeon64	Processors	4x Xeon X7550 (Nehalem), 8 cores, 2-SMT
	Caches/proc.	8x 32 KByte L1, 8x 256 KByte L2, 18 MByte L3
	Main memory	4 NUMA nodes, 128 GByte DDR3-1066, 4 KByte page size
	Environment	Linux kernel 3.8, GCC 4.6
Simics	Processors	4x 2 cores, SPARC instruction set, 2.0 GHz, 32 nm
	L1 cache/proc.	2x 16 KByte, 4-way, 1 bank, 2 cycles latency
	L2 cache/proc.	1 MByte, 8-way, 2 banks, 5 cycles latency
	TLB/proc.	2x TLBs (64 entries), 4-way, 1 TLB per core
	Cache coherency	Directory-based MOESI protocol, 64 Byte lines
	Main memory	8 GByte DDR3-1333 9-9-9, 4 KByte page size
	Interconnection	1/40 cycles latency (intra/interchip)
		64/16 Byte bandwidth (intra/interchip)
	Environment	Linux kernel 2.6.15, GCC 4.3

similar to previous work [Cuesta et al. 2013]. We compare the results in the simulator to a first-touch page mapping, interleaving and to an oracle mapping, which generates mappings considering all memory accesses. Results in the simulator are normalized to the first-touch mapping.

5.2. Evaluation in a Real Machine with Software-Managed TLB

Although IPM is an extension to the current hardware of MMUs, it is possible to implement its behavior in software in architectures that have a software-managed TLB. In these architectures, whenever a TLB miss happens, there is no hardware page table walker to fetch the page table entry from the main memory. Instead, the architecture generates a TLB miss interrupt that is handled by the operating system, which walks through the page table in software.

The Itanium architecture [Intel 2010] has a hybrid mechanism to handle TLB misses, where we can choose to use a hardware-based manager called Virtual Hash Page Table (VHPT) walker, or handle the TLB misses in software. We implemented IPM in the Linux kernel version 2.6.32 for the Itanium architecture configured to use a software-managed TLB. Although the architecture does not allow the software to access the contents of the TLBs, as all the TLB misses are handled by the operating system, we can manage a virtual TLB in software in the main memory. The architecture also provides access to a cycle accurate time stamp by reading the `ar.itc` register. The code that performs the same procedure of IPM was added to the data TLB miss interrupt handler of the kernel. We refer to this machine as *Itanium*. Hardware performance counters can be accessed in Itanium using Perfmon2 [Eranian 2006].

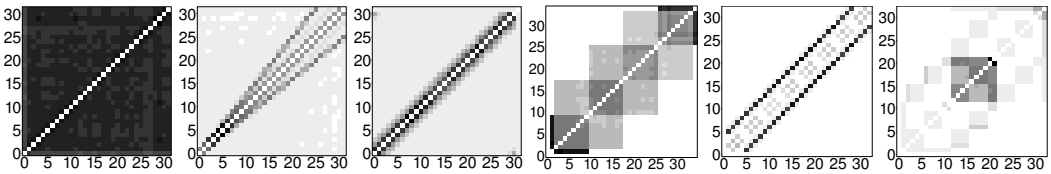
In Itanium, we compare the results obtained with IPM to the original scheduler of Linux, an oracle mapping, an interleaved data mapping, NUMA Balancing [Corbet 2012b], kMAF [Diener et al. 2014] and the usage of TLB misses. For the oracle mapping, for most benchmarks, we generated traces of all memory accesses for each application and performed an analysis of the sharing and page usage patterns, similar to [Barrow-Williams et al. 2009]. For one of the applications evaluated in Itanium, we generate the oracle mapping by modifying its source code, as detailed in Section 5.4. NUMA Balancing and kMAF had to be ported to work in Itanium. Regarding the comparison to TLB misses, we compare to two strategies: (i) similar to NUMA Balancing, we migrate pages to the NUMA node that generated a TLB miss on that page; (ii) does the same procedure of IPM, keeping a Sharing Matrix, Sharers Vector and a NUMA Vector, but using TLB misses instead of TLB residency. Each experiment in Itanium was executed 30 times, and we show average values as well as a 95% confidence interval calculated with Student's t-distribution. Results are normalized to the operating system mapping.

5.3. Trace-Driven Evaluation in Real Machines with Hardware-Managed TLB

Experiments were also performed using two different real machines. The first machine, *Xeon32*, consists of two NUMA nodes with one Intel Xeon E5-2650 processor per node, with a total of 32 virtual cores. The second machine, *Xeon64*, consists of four NUMA nodes with one Intel Xeon X7550 processor per node, with a total of 64 virtual cores. The machines are running version 3.8 of the Linux kernel. Information about the hardware topology is gathered using Hwloc [Broquedis et al. 2010b]. Besides performance, we measured L3 cache misses per thousand instructions (MPKI), interchip interconnection traffic (QuickPath Interconnection) and energy consumption (RAPL hardware counters [Intel 2012a]) using the Intel PCM tool [Intel 2012b].

Since IPM is an extension to the current MMU hardware, we simulate its behavior with the Pin dynamic binary instrumentation tool [Bach et al. 2010]. The simulated hardware uses the same TLB configuration as the real machines. We used Pin because it is faster than a full system simulator. To make it possible to evaluate IPM in real machines with hardware-managed TLBs, the mapping information generated in Pin is fed into the mapping mechanism during run time.

Similar to *Itanium*, all experiments in the two Xeon machines were executed 30 times, and we show average values as well as a 95% confidence interval calculated with Student's t-distribution. We compare the results of IPM to the default mapping performed by the operating system, to random static mappings, and to an oracle mapping. The operating system mapping uses the original



(a) CG. (b) MG. (c) SP. (d) Ferret. (e) Fluidanimate. (f) Dedup.

Fig. 6: Sharing patterns of some applications. Axes represent thread IDs and cells the amount of accesses to shared pages between threads. Darker cells indicate more accesses.

scheduler and data mapping policy (first touch) of Linux. For the random mapping, we randomly generated a thread and data mapping for each execution. For the oracle mapping, we generated traces of all memory accesses for each application and performed an analysis of the sharing and page usage patterns, similar to [Barrow-Williams et al. 2009]. All results are normalized to the operating system mapping.

5.4. Workloads

As workloads, we used the OpenMP implementation of the NAS parallel benchmarks (NPB) [Jin et al. 1999], v3.3.1, and the PARSEC benchmark suite [Bienia et al. 2008], v3.0. We configured the benchmarks to run with one thread per virtual core, although some applications of PARSEC execute with multiple threads per virtual core. Some PARSEC applications were not executed in Itanium due to software FPU errors.

For the evaluation in the real machines with hardware-managed TLBs, the evaluated applications must present the same sharing and page usage patterns across different executions, as well as keeping the same memory address space, since the trace generated in Pin is used to guide mapping decisions. For this reason, only the NAS applications (except DC) were executed on the real machines with hardware-managed TLBs. DC and applications from PARSEC usually do not keep the same memory address space due to dynamic memory allocation. This makes the information generated in Pin unreliable to guide mapping in future executions.

Input sizes were chosen to provide similar total execution times and feasible simulation times. Regarding NAS, benchmarks BT, LU, SP and UA were executed using input size *A* in Pin, Xeon32 and Xeon64, and input size *W* in Simics. Benchmarks CG, EP, FT, IS and MG were executed using input size *B* in Pin, Xeon32 and Xeon64, and input size *A* in Simics. DC was executed with input size *W* in Simics. Regarding PARSEC in Simics, the input size used in Canneal was *simmedium*, and all others *simlarge*. In the Itanium machine, DC was executed with input size *A* and all other NAS benchmarks with input size *B*, and the ones of PARSEC with the *native* input size.

We also experiment with a real world scientific application, Ondes3D [Dupros et al. 2008]. Ondes3D simulates the propagation of seismic waves using a finite-differences numerical method. The oracle mapping in Ondes3D was not generated by analyzing memory traces, as explained in Section 5.2, because the execution time of Ondes3D is too large to enable the generation of a trace. Therefore, the oracle mapping in Ondes3D is performed by manually adding the mapping routines in its source code. By exploiting the regular memory access pattern of finite-differences applications, we guarantee that the memory accessed by each thread is mapped nearby [Dupros et al. 2010].

6. EVALUATION OF IPM

The evaluation comprehends results regarding performance, energy consumption, and overhead. They are presented in this section following this order. A comparison with previous mapping mechanisms is shown in Section 6.1.2.

6.1. Performance Results

One of the key aspects to understand the performance improvement obtained by sharing-aware mapping is the sharing pattern between the threads. Fig. 6 illustrates the sharing patterns of some of the evaluated applications detected by IPM. In the illustrations, axes represent the threads of the application, and each cell shows the amount of data sharing between each pair of threads. Darker cells indicates more shared data.

Applications with threads that share more data within a small group present more potential for performance improvements when mapping the threads with most sharing to cores nearby in the memory hierarchy. In SP, whose sharing pattern is illustrated in Fig. 6c, neighboring threads present intense sharing. The patterns of MG and Fluidanimate, shown in Fig. 6b and 6e, are similar, but also present data sharing between more distant threads. Ferret and Dedup, illustrated in Fig. 6d and 6f, have a pipeline sharing model, forming thread clusters that share data.

In some applications, the amount of data shared between threads is very similar. The sharing pattern of CG, shown in Fig. 6a, falls in this category. For these applications, thread mapping does not affect performance, since there is no mapping that optimizes access to shared data. However, data mapping is still able to improve their performance by mapping the private data each thread uses to its NUMA node.

In the remainder of this section, we explain the performance improvements obtained in each platform.

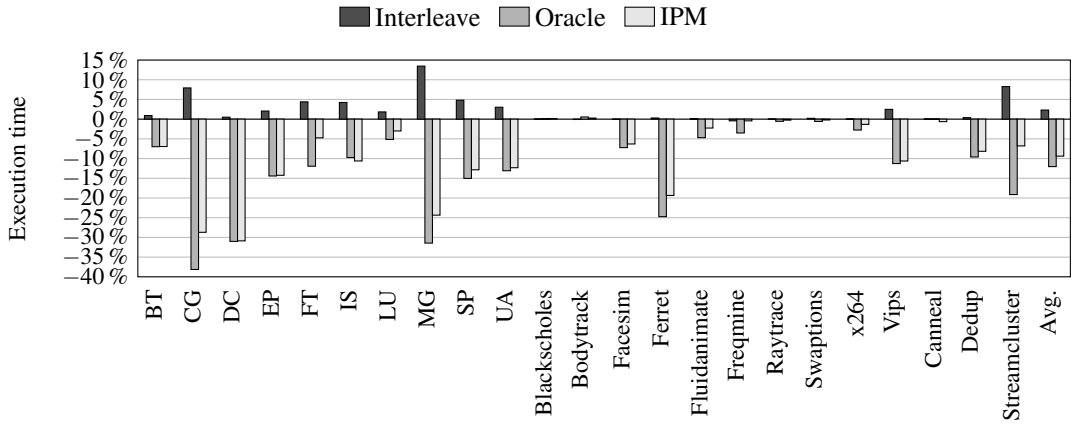
6.1.1. Results Generated in Simics. The execution times obtained in Simics are shown in Fig. 7a, and the interchip interconnection traffic is shown in Fig. 7b. We can observe that the NAS applications presented better improvements than the PARSEC ones. IPM achieved the best improvements with CG, reducing execution time by 28.7%. This is a result of a better data mapping, as IPM reduced the interchip interconnection traffic by 55.3%. Similar behaviors of interchip traffic and execution time reductions are seen in DC, MG, SP and Ferret. We can note a slight decrease in performance in Blackscholes and Bodytrack due to the overheads of monitoring the TLB and thread migrations. On average, execution time and interchip interconnection traffic in Simics were reduced by 9.4% and 52.0%, respectively.

In some applications, no performance improvements are expected by either thread or data mapping. Swaptions is an example of such an application. Although its sharing pattern is similar to the one of CG, its memory usage is much smaller, as almost all its data fits in the caches. Therefore, although we decrease interchip traffic in Swaptions, the absolute reduction is very small (about 9.5% of CG's) and not affected by data mapping.

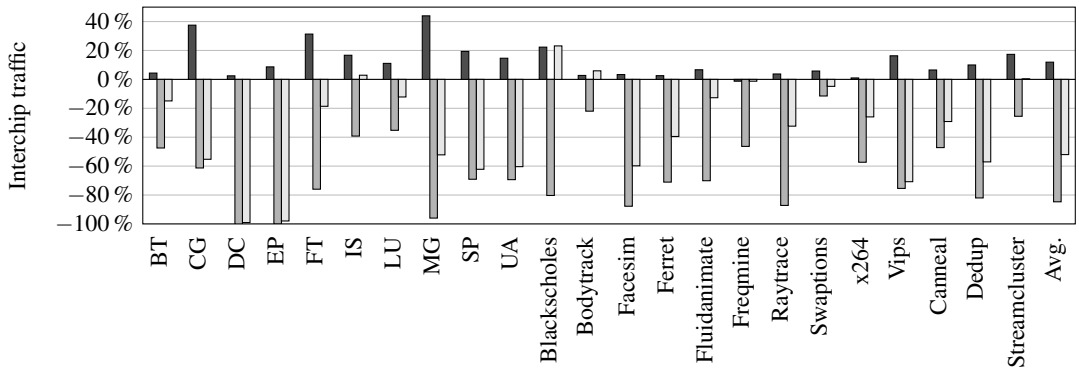
6.1.2. Results Generated in Itanium. The execution times in Itanium can be found in Fig. 8. CG was the application with the best reduction in execution time (39.0%). Since the memory hierarchies are very different between platforms, some applications present different results, such as DC and Ferret. The lack of hardware counters to monitor interchip traffic in Itanium makes it difficult to better understand some results, but one of the main reasons for differences comes from the absence of shared caches in Itanium. Nevertheless, most results are very similar to the ones from Simics, where NAS applications such as CG, MG, SP and UA presented the best improvements.

The results generated in Itanium show that IPM reduced execution time by an average of 13.7%, while the oracle mapping reduced execution time by 14.2% on average. The experiments with the Ondes3D application showed an execution time reduction of 30.8% using IPM, while the oracle mapping reduced execution time by 30.5%. It is important to emphasize that IPM was developed as a hardware extension and that this machine is emulating IPM's behavior with its software-managed TLB. This emulation ends up stalling the execution of the thread that generated the TLB miss, as IPM is executed in its core. This would not be an issue with the hardware implementation of IPM, as the MMU would operate in parallel to the application.

We compare IPM on Itanium to several previously mentioned techniques: Interleave, NUMA Balancing [Corbet 2012b], the kMAF affinity framework [Diener et al. 2014], and to mapping using



(a) Execution time.



(b) Interchip interconnection traffic.

Fig. 7: Performance results in Simics, normalized to the operating system, represented as 0%.

TLB misses as source of information. The results of kMAF are lower than IPM due to its sampling mechanism, as kMAF needs more time to detect the memory access behavior, losing opportunities for improvements. NUMA Balancing performed a little worse than kMAF mainly because of unnecessary page migrations, since NUMA Balancing keeps no history regarding page migrations, in contrast to kMAF. The overhead of adding more page faults in kMAF and NUMA Balancing to generate a better profile is high compared to the usage of TLB residency in IPM.

The usage of TLB misses alone with no history, similarly to NUMA Balancing, resulted in too many page migrations, harming performance in several applications. The amount of page migrations using this policy is much higher than in NUMA Balancing, since the amount of TLB misses is also much higher than the amount of page faults introduced by NUMA Balancing. On the other hand, using TLB misses as a source of information to the same procedures performed by IPM results in a very good performance, since it reduces the amount of unnecessary page migrations. Nevertheless, the usage of TLB residency provided a clear advantage over TLB misses in some applications, such as BT, FT, SP and Facesim. As the implementation complexity of TLB residency is almost the same as TLB misses, we believe the usage of TLB residency is recommended.

The comparison to the related work shows that mechanisms that perform both thread and data mapping are able to achieve better improvements than mechanisms that perform these mapping separately. It also shows that simple policies do no guarantee a good performance for all applica-

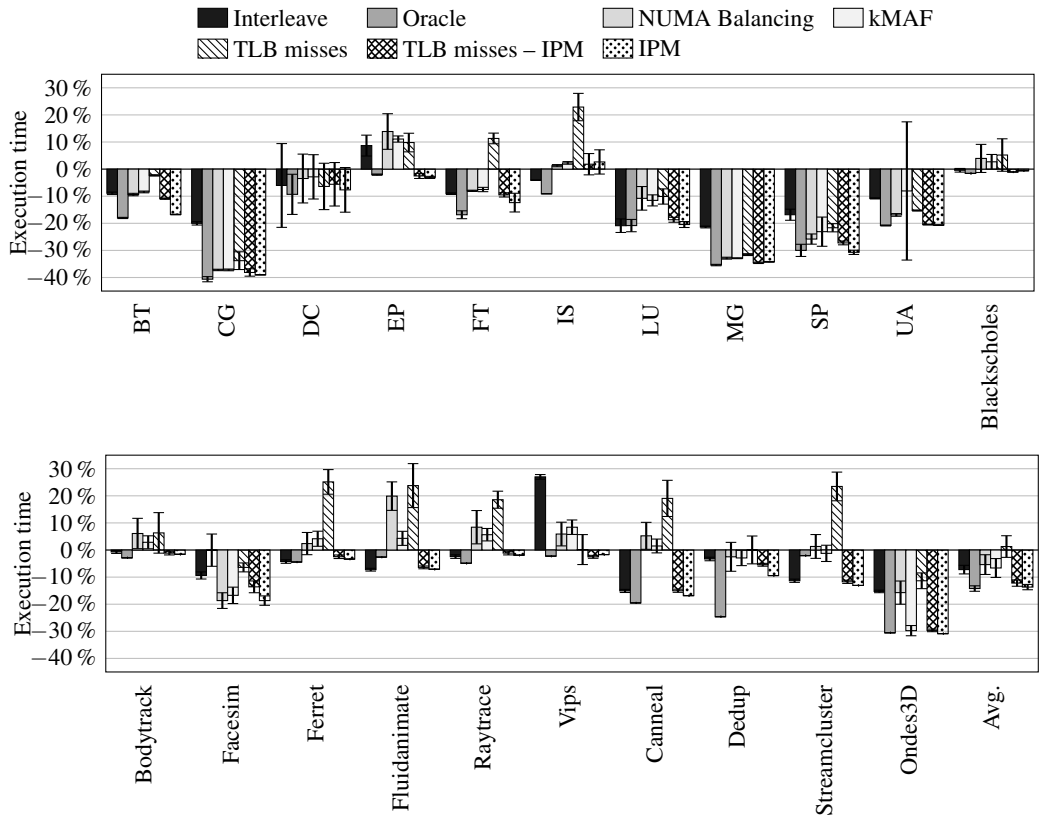


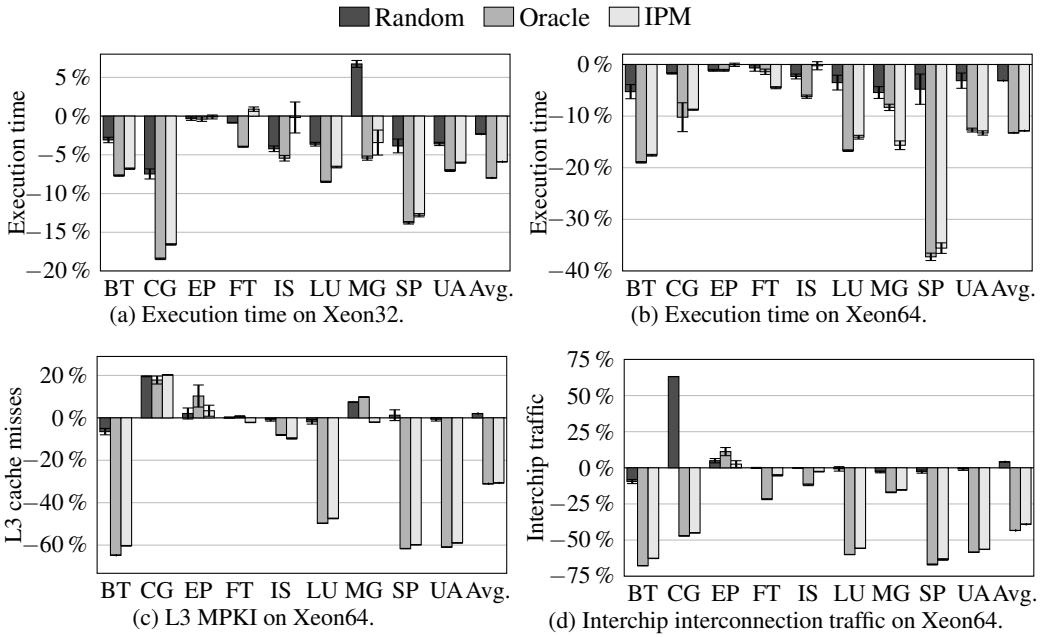
Fig. 8: Execution time in Itanium, normalized to the operating system, represented as 0%.

tions. As we can observe in the interleaved mapping policy, it provided a high variance in the results. Furthermore, the results shows that IPM, with its TLB residency-based metric, is able to achieve a higher accuracy and better performance than other mechanisms.

Finally, the results obtained in Itanium demonstrate that IPM works not only in a simulator, but also in a real machine. Furthermore, these results indicate that IPM is able to improve performance not only for traditional benchmarks, but also for real applications (Ondes3D). Also, as explained in Section 5.4, the oracle mapping in Ondes3D was generated by inserting mapping routines directly in the source code, which shows that IPM achieves improvements as good as manually optimized code.

6.1.3. Results Generated in Xeon32 and Xeon64. The execution time in Xeon32 and Xeon64 can be found in Figs. 9a and 9b, respectively. We also show results of L3 cache misses and interchip interconnection traffic obtained in Xeon64 in Figs. 9c and 9d. In Xeon32, CG was the application with the highest improvements, reducing execution time by 16.6%. CG, as explained before, has a sharing pattern in which thread mapping is not able to improve performance, which is the reason why we only reduce interchip interconnection traffic (cache misses were actually increased). In Xeon64, SP was the application with the highest improvements, with a reduction of execution time of 35.6%. Due to its sharing pattern, both cache misses and interchip traffic were reduced, by 59.8% and 63.4%, respectively.

We can observe how thread mapping affects data mapping by analyzing MG. The sharing pattern of MG is similar to the one of SP, thereby thread mapping affects its performance. However, the memory usage of MG is much higher than SP, such that its amount of data shared by threads is



(a) Execution time on Xeon32.
(b) Execution time on Xeon64.
(c) L3 MPKI on Xeon64.
(d) Interchip interconnection traffic on Xeon64.

Fig. 9: Performance results, normalized to the operating system, represented as 0%.

much higher than the L3 cache size, resulting in no reduction in L3 cache misses. Thread mapping improves data mapping in cases where pages are shared by multiple threads. In this way, the more appropriate thread mapping puts threads that share pages on the same NUMA node, thus reducing interchip traffic. Therefore, we are able to observe MG's potential for thread mapping by looking at interchip traffic, and not at cache misses.

Comparing the different metrics analyzed, we observe that the highest reduction occurred in the interchip interconnection traffic, with an average reduction of 39.1% in Xeon64. Meanwhile, execution times were reduced by an average of 12.9% in Xeon64 and 5.9% in Xeon32. The effects in total execution time are smaller because they are influenced by several factors, as opposed to interchip traffic and cache misses, which are directly influenced by better mappings.

6.1.4. Summary. The results obtained in Simics, Itanium and Xeon64 are better than the results obtained in Xeon32. As Simics and Xeon64 have 4 NUMA nodes (while Xeon32 has 2), the probability of finding the correct node to a page without any knowledge of the memory access pattern is only 25% on them (while it is 50% on Xeon32). Regarding Itanium, although it also has only 2 NUMA nodes, a better mapping has a bigger impact because the latency of a remote memory access is much higher than in Xeon32.

Most applications are more sensitive to data mapping than thread mapping, which can be observed in the results by the fact that the interchip traffic presented a higher reduction than cache misses. This happens because, even if an application does not share much data among its threads, each thread will still need to access its own private data, which can only be improved by data mapping. It is important to note that this does not mean that data mapping is more important than thread mapping, because the effectiveness of data mapping depends on thread mapping for shared pages.

IPM presented results similar to the oracle mapping, demonstrating its effectiveness. It also performed significantly better than the random mapping for most cases. This shows that the gains over the operating system are not due to the unnecessary migrations introduced by the operating system, but due to a more efficient usage of resources. Results were also significantly better than other related work.

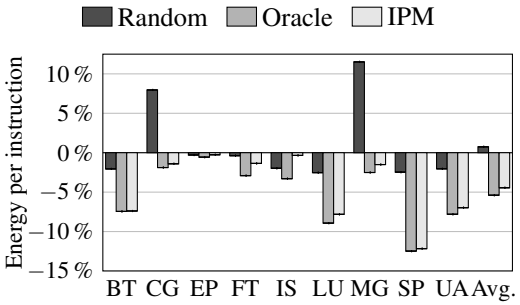


Fig. 10: Energy per instruction on Xeon32, normalized to the operating system, represented as 0%.

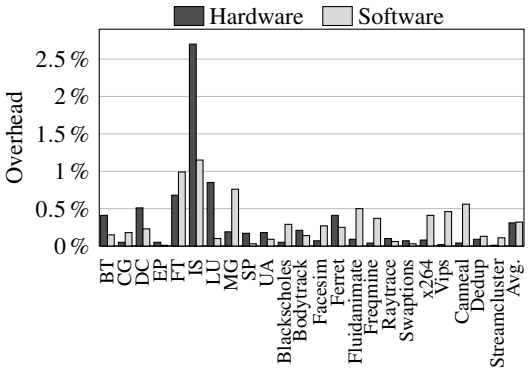


Fig. 11: Performance overhead in Simics.

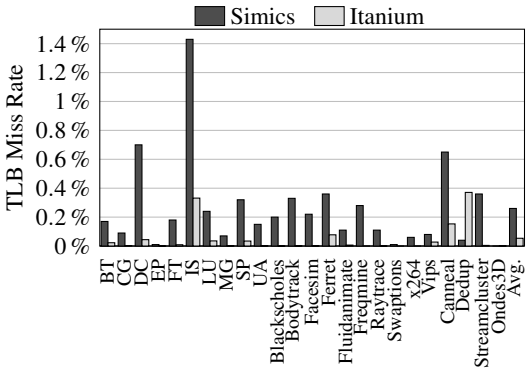


Fig. 12: TLB miss rate in Simics and Itanium.

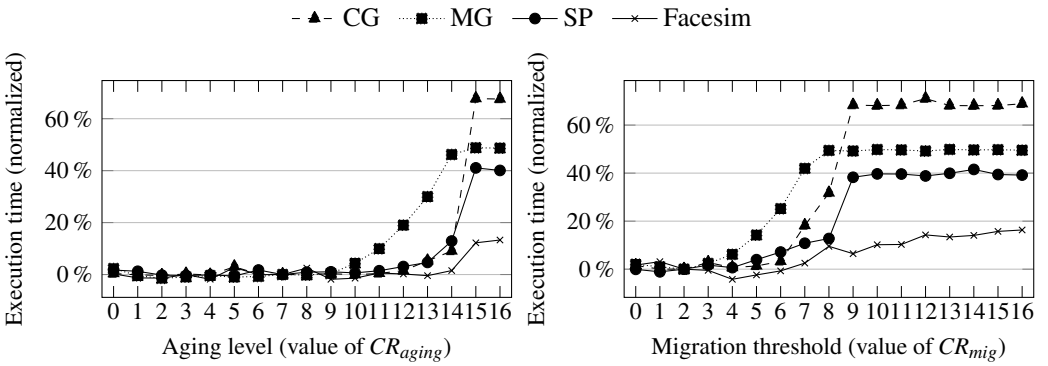
6.2. Energy Consumption Results

Results of the total amount of energy per instruction for Xeon32 are shown in Fig. 10. Energy per instruction was improved by 4.4% on average, and up to 12.2% in SP. This shows that our mechanism not only saves energy by reducing the executing time, but also by providing a more efficient execution, which is an important goal for future Exascale architectures [Torrellas 2009]. We also measured the DIMM and processor energy separately, but do not show them due to space constraints. These measurements show a higher reduction of DIMM energy than processor energy, 9.6% and 5.2% on average, respectively, because a sharing-aware mapping has more influence in the memory than in the processor.

6.3. Performance Overhead

IPM causes an overhead on the execution of the application on the hardware and software levels, as discussed in Section 4.5.3. We evaluate the hardware overhead by running IPM without performing any migration, and compare the execution time to the baseline without IPM. For the software overhead, we measure the time spent to calculate the mapping and perform migrations. We only show the performance overhead in Simics because the real machines with hardware-managed TLBs do not implement IPM. The performance overhead in Simics is shown in Fig. 11. We also show the TLB miss rate of the experiments running in Simics and Itanium in Fig. 12.

The average performance overhead caused by the hardware was 0.31%, due to the introduction of 1.2% additional memory transactions, on average. IS has the highest overhead due to its large TLB miss rate, 1.43% in Simics, introducing more memory accesses. The average TLB miss rate was 0.26% and 0.05% in Simics and Itanium, respectively. As explained in Section 4.5, our im-



(a) Varying the aging factor (the higher CR_{aging}, less aging). Normalized to $CR_{aging} = 7$.

(b) Varying the multiplication threshold (the higher CR_{mig}, less migrations). Normalized to $CR_{mig} = 2$.

Fig. 13: Normalized execution time of the applications when varying configurable parameters.

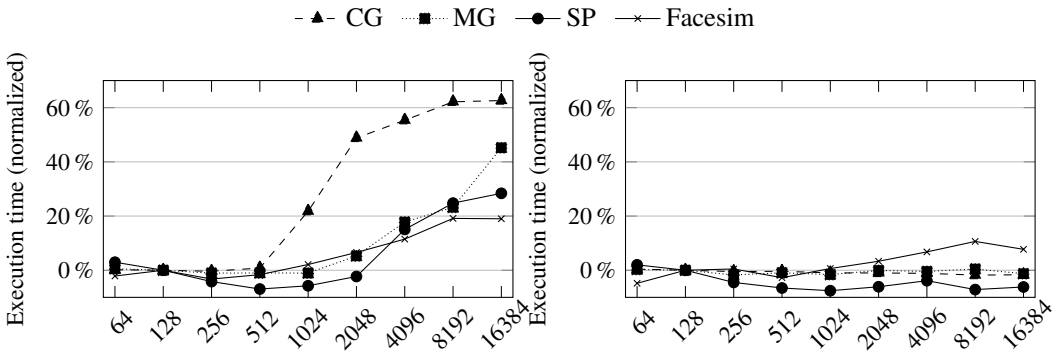
plementation of IPM is able to handle only one event per core simultaneously, such that IPM was able to handle 67.6% of the total amount of TLB evicts. IPM required an average of 255 cycles to handle each TLB eviction. Application execution is not stalled while IPM handles a TLB eviction. The overhead in the software level was 0.32%, on average. These results show that IPM presents only a minor overhead.

As explained in Section 4.5.3, race conditions can happen when updating the Page History Table. In Simics, on average, only 0.12% of the TLB evictions caused this race condition. The impact of these race conditions on the accuracy was very low. On average, IPM maps 91.3% of the total pages to the correct node. If there were no race conditions, the accuracy would be increased to 91.6%. In the worst case, in SP, there were 0.5% of race conditions, dropping the accuracy by 0.9%. No impact was observed in the thread mapping accuracy. Statistically, no performance impact was observed due to the race conditions.

6.4. Design Space Exploration

The configurable parameters of IPM control migrations. The size of each element of the NUMA Vector, CR_{shift} and CR_{aging} depend on each other. We decided to fix the size of each NUMA Vector element to 2 bytes because it has a large range, and at the same time does not impose a high memory usage in the Page History Table. Regarding CR_{shift}, a low value would make the resident time (Eq. 1 in Section 4.3.1) too large, such that it would not fit in the NUMA Vector. On the other hand, a high value for CR_{shift} would make the resident time lose too much precision. Based on that, we empirically determined that a good value for CR_{shift} is 13. After fixing the size of the NUMA Vector and CR_{shift}, we analyzed how the aging (CR_{aging}) and the migration threshold (CR_{mig}) affect performance. We also check the impact of the number of TLB entries and page size. For this analysis, we ran the experiments in the Itanium machine and used the applications CG, MG, SP and Facesim, because they present very different characteristics. Each experiment is normalized to a different baseline, represented as 0% in all graphics.

6.4.1. Impact of Aging (CR_{aging}). The execution time varying the aging value is shown in Fig. 13a. The results are normalized to the results of CR_{aging} set to 7. The higher the value of CR_{aging}, less aging is performed (Eq. 6 in Section 4.3.3). In our experiments, we observed that an aggressive aging (low CR_{aging}) did not have a negative impact on performance, because the migrations did not increase significantly, as, in these applications, most memory pages are accessed by a single NUMA node [Diener et al. 2014]. On the other hand, we observed a negative impact on performance when using a conservative aging (high CR_{aging}) because it increased too much the time to start



(a) Without verifying infrequently evicted pages. (b) Verifying infrequently evicted pages.

Fig. 14: Execution time varying the number of TLB entries, normalized to 128 entries per TLB.

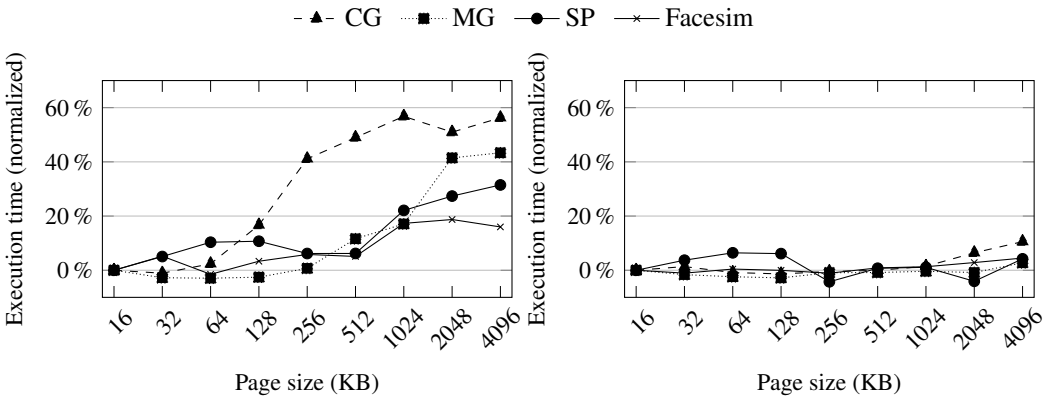
migrating data, since the initial values of the NUMA Vector are set according to the aging (Eq. 9 in Section 4.3.3).

6.4.2. Impact of the Migration Threshold (CR_{mig}). The execution time varying the migration threshold value is shown in Fig. 13b. The results are normalized to the results of CR_{mig} set to 2. The higher the value of CR_{mig}, the more difficult it is to migrate a page (Eq. 8 in Section 4.3.3). We observed that an aggressive threshold (low CR_{mig}) did not lead to too many migrations, not harming performance. The reason is the same as in the evaluation of the aging: in these applications, most memory pages are accessed by a single NUMA node [Diener et al. 2014]. However, conservative values of the threshold (high CR_{mig}) made it too difficult to migrate the pages, losing opportunities to improve performance.

6.4.3. Impact of the Number of TLB Entries. The number of TLB entries per core directly influences the number of TLB misses and evictions. Since TLB evictions are the primary source of information of IPM, it is important to evaluate how the number of TLB entries affects IPM. Fig. 14 contains the execution time when varying the number of TLB entries in the Itanium machine. Although the number of entries in the hardware TLB is fixed, we can perform this evaluation in Itanium because, as we explain in Section 5.2, we manage a virtual TLB in software. We perform two different sets of experiments: the first using only TLB evictions as source of information, and the second also enabling the verification of infrequently evicted pages, as described in Section 4.4.1. Results are normalized to the execution time using 128 entries per TLB, which is the value used in the previous experiments.

Analyzing Fig. 14a, where IPM does not verify infrequently evicted pages, we can observe that the execution time increases when the number of TLB entries per core is higher than 512. This happens because the amount of TLB evictions drastically decreases, such that pages have their entries in the TLB for too much time, and IPM does not consider these pages for migrations. However, when enabling the support for infrequently evicted pages, IPM is able to handle this scenario and the execution time remains stable, even with an unrealistic TLB size, as can be seen in Fig. 14b.

6.4.4. Impact of the Page Size. Another important architectural parameter that has a high influence in the number of TLB evictions is the page size. The higher the page size, the lower the number of evictions. Also, the page size represents the granularity of the memory block used to detect the data sharing. With larger pages, it is likely that more threads and NUMA nodes will access the same page [Diener et al. 2014]. To analyze how this influences our proposal, we run experiments varying



(a) Without verifying infrequently evicted pages. (b) Verifying infrequently evicted pages.

Fig. 15: Execution time varying the page size, normalized to a 16 KBytes page size.

the page size from 16 KBytes to 4 MBytes. We perform experiments disabling and enabling the support for infrequently evicted pages.

Results can be found in Fig. 15. They follow the same tendency of the experiment of Section 6.4.3. When we increase the page size and thereby decrease the number of TLB evictions, the execution time increases if the support for infrequently evicted pages is disabled. On the other hand, IPM is able to handle large pages when the support for infrequently evicted pages is enabled.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed IPM, a mechanism that uses information about the time each page entry resides in the TLB to perform thread and data mapping. The TLB residency metric proved to have a better accuracy than other metrics, such as TLB misses and sampling of memory addresses, and has a lower implementation cost than metrics that consider all memory accesses. In this way, our proposed metric has the best trade-off between accuracy and implementation cost in the state of the art. Architectures with hardware-managed TLBs require very little extra circuit area to implement IPM, while architectures with software-managed TLBs can implement our proposal directly in software, without any hardware changes.

We performed an extensive evaluation of IPM, including two benchmark sets and one scientific application. The experiments were performed in one simulated machine, one real machine with software-managed TLB and two real machines with hardware-managed TLB. The execution time was reduced by an average of 9.4%, 13.7%, 5.9% and 12.9% on the machines, respectively. Results were highly dependent on the sharing-pattern of each application, as well as the characteristics of each machine, reducing execution time by up to 39.0% in applications whose sharing patterns had high potential for mapping. To better understand the performance improvements, we also measured cache misses and interchip interconnection traffic, with an average reduction of 30.7% and 39.1% in the real machine. Energy efficiency was improved by an average of 4.4%. We compared our proposal to related work, in which IPM had the best performance.

As future work, we intend to extend our mechanism to balance the load in memory controllers.

Acknowledgment

This research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement n.º 689772. Additional funding was provided by CNPq and Capes.

REFERENCES

Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 33–42. DOI: http://dx.doi.org/10.1109/ISPASS.2009.4919636

Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. 2009. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. *ACM SIGOPS Operating Systems Review* 43, 2 (April 2009), 56–65. DOI: http://dx.doi.org/10.1145/1531793.1531803

Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. 2010. Analyzing Parallel Programs with Pin. *IEEE Computer* 43, 3 (2010), 34–41.

Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A Communication Characterisation of Splash-2 and Parsec. In *IEEE International Symposium on Workload Characterization (IISWC)*. 86–97.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (2011), 67–77.

François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-Andr Wacrenier, and Raymond Namyst. 2010a. Structuring the execution of OpenMP applications for multicore architectures. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. 1–10.

Franois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010b. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*. 180–186.

Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2005. Optimizing Replication, Communication, and Capacity Allocation in CMPs. *ACM SIGARCH Computer Architecture News* 33, 2 (May 2005), 357–368.

Jonathan Corbet. 2012a. AutoNUMA: the other approach to NUMA scheduling. (2012). http://lwn.net/Articles/488709/

Jonathan Corbet. 2012b. Toward better NUMA scheduling. (2012). http://lwn.net/Articles/486858/

P. W. Coteus, J. U. Knickerbocker, C. H. Lam, and Y. A. Vlasov. 2011. Technologies for exascale systems. *IBM Journal of Research and Development* 55, 5 (Sept. 2011), 14:1–14:12. DOI: http://dx.doi.org/10.1147/JRD.2011.2163967

Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. 2015. An Efficient Algorithm for Communication-Based Task Mapping. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 207–214.

Blas Cuesta, Alberto Ros, Maria E. Gomez, Antonio Robles, and Jose Duato. 2013. Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Non-Coherent Memory Blocks. *IEEE Trans. Comput.* 62, 3 (2013), 482–495. DOI: http://dx.doi.org/10.1109/TC.2011.241

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 381–393.

Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. 124–133. DOI: http://dx.doi.org/10.1109/IPDPS.2006.1639359

Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 277–288.

Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2015. Communication-Aware Process and Thread Mapping Using Online Communication Detection. *Parallel Comput.* 43, March (2015), 43–63.

Fabrice Dupros, Hideo Aochi, Ariane Ducellier, Dimitri Komatitsch, and Jean Roman. 2008. Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. In *IEEE International Conference on Computational Science and Engineering (CSE)*. 253–260. DOI: http://dx.doi.org/10.1109/CSE.2008.51

Fabrice Dupros, Christiane Pousa, Alexandre Carissimi, and Jean-François Méhaut. 2010. Parallel simulations of seismic wave propagation on NUMA architectures. In *Parallel Computing: From Multicores and GPU's to Petascale*. 67–74.

Stephane Eranian. 2006. Perfmon2: a flexible performance monitoring interface for Linux. In *Proceedings of the Linux Symposium*.

Josue Feliu, Julio Sahuquillo, Salvador Petit, and Jose Duato. 2012. Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

Ilaria Di Gennaro, Alessandro Pellegrini, and Francesco Quaglia. 2016. OS-based NUMA Optimization: Tackling the Case of Truly Multi-Thread Applications with Non-Partitioned Virtual Page Accesses. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID)*. 291–300. DOI: http://dx.doi.org/10.1109/CCGrid.2016.91

Intel. 2010. *Intel Itanium Architecture Software Developer's Manual*. Technical Report.

Intel. 2012a. *2nd Generation Intel Core Processor Family*. Technical Report September.

Intel. 2012b. Intel Performance Counter Monitor - A better way to measure CPU utilization. (2012). http://www.intel.com/software/pcm

Emmanuel Jeannot and Guillaume Mercier. 2010. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Euro-Par Parallel Processing*. 199–210.

JEDEC. 2012. DDR3 SDRAM Standard. (2012).

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

George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (Jan. 1998), 359–392.

Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. 2008. autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. *High Performance Embedded Architectures and Compilers (HiPEAC)* 3, 4 (2008), 219–235.

Richard P. LaRowe, Mark A. Holliday, and Carla Schlatter Ellis. 1992. An Analysis of Dynamic Page Placement on a NUMA Multiprocessor. *ACM SIGMETRICS Performance Evaluation Review* 20, 1 (1992), 23–34.

Henrik Löf and Sverker Holmgren. 2005. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *International Conference on Supercomputing (SC)*. 387–392.

P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A Full System Simulation Platform. *IEEE Computer* 35, 2 (2002), 50–58.

Jaydeep Marathe and Frank Mueller. 2006. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 90–99.

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

Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why On-Chip Cache Coherence is Here to Stay. *Commun. ACM* 55, 7 (July 2012), 78. DOI: http://dx.doi.org/10.1145/2209249.2209269

M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, Min Xu, A. R. Alameldeen, K. E. Moore, M .D. Hill, and D. A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News* 33, 4 (2005), 92–99.

Takeshi Ogasawara. 2009. NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. *ACM SIGPLAN Notices* 44, 10 (Oct. 2009), 377–389.

Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, Fernando M. Quintão Pereira, and Fernando Magno. 2014. Compiler support for selective page migration in NUMA architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 369–380.

Petar Radojković, Vladimir Cakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2013. Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 24, 12 (2013), 2513–2525.

Christiane Pousa Ribeiro, Jean-François Méhaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. 2009. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 59–66.

Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. 2008. Data and Thread Affinity in OpenMP Programs. In *Workshop on Memory Access on Future Processors: A Solved Problem? (MAW)*. 377–384.

Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, Norman P Jouppi, and Palo Alto. 2008. *Cacti 5.1*. Technical Report.

Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing (JPDC)* 68, 9 (Sept. 2008), 1186–1200.

Josep Torrellas. 2009. Architectures for extreme-scale computing. *IEEE Computer* 42, 11 (2009), 28–35.

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. *OS Support for Improving Data Locality on CC-NUMA Compute Servers*. Technical Report February.

Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 129–142.

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Computing Surveys (CSUR)* 45, 1, Article 4 (2012).