Online Thread and Data Mapping Using a Sharing-Aware Memory Management Unit

Eduardo H M Cruz, Matthias Diener, Laércio Lima Pilla, Philippe O A Navaux

Online Thread and Data Mapping Using a Sharing-Aware Memory Management Unit *

Eduardo H. M. Cruz[1], Matthias Diener[2], Laércio L. Pilla[3], and Philippe O. A. Navaux[4]

[1]Campus Paranavaí – Federal Institute of Paraná (IFPR), Paranavaí, Brazil – email: eduardo.cruz@ifpr.edu.br
[2]University of Illinois at Urbana-Champaign, Urbana, USA – email: mdiener@illinois.edu
[3]LRI, Univ. Paris-Saclay, CNRS, Orsay, France – email: pilla@lri.fr
[4]Informatics Institute – Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil – email: navaux@inf.ufrgs.br

Abstract

Current and future architectures rely on thread-level parallelism to sustain performance growth. These architectures have introduced a complex memory hierarchy, consisting of several cores organized hierarchically with multiple cache levels and NUMA nodes. These memory hierarchies can have an impact on the performance and energy efficiency of parallel applications as the importance of memory access locality is increased. In order to improve locality, the analysis of the memory access behavior of parallel applications is critical for mapping threads and data. Nevertheless, most previous work relies on indirect information about the memory accesses, or does not combine thread and data mapping, resulting in less accurate mappings.

In this paper, we propose the Sharing-Aware Memory Management Unit (SAMMU), an extension to the memory management unit that allows it to detect the memory access behavior in hardware. With this information, the operating system can perform online mapping without any previous knowledge about the behavior of the application. In the evaluation with a wide range of parallel applications (NAS Parallel Benchmarks and PARSEC Benchmark Suite), performance was improved by up to 35.7% (10.0% on average) and energy efficiency was improved by up to 11.9% (4.1% on average). These improvements happened due to a substantial reduction of cache misses and interconnection traffic.

1 Introduction

Shared-memory architectures are increasing the thread-level parallelism (TLP) through larger numbers of processors per system and cores per chip [3, 10, 15]. This increase of TLP has influenced the memory architecture. Modern memory hierarchies are complex and include memories with different access latencies and bandwidths. These differences are introduced by multiple

*Author's accepted version. Definitive version available at https://doi.org/10.1145/3433687

cache memory levels, as well as non-uniform memory access (NUMA), and they have an impact on the performance and energy efficiency of parallel applications [31, 49, 24, 51].

The memory hierarchy influences the locality of memory accesses, and thereby presents challenges for mapping threads to cores and data to NUMA nodes [50]. To improve locality in parallel applications, threads that access a large amount of shared data should be mapped to cores that are close to each other in the memory hierarchy, while data should be mapped to the same NUMA node of the threads that are accessing it [43]. An improvement of memory access locality leads to an increase of performance and energy efficiency. We characterize this type of thread and data mapping as *sharing-aware*. Sharing-aware thread mapping uses knowledge about the data that is shared between threads, while sharing-aware data mapping uses information about the threads that access each memory page. Furthermore, these mappings should be performed together to achieve the best results [44].

Improvements due to sharing-aware mapping happen due to a reduction of cache misses, improvement of locality and reduction of traffic in slow interchip interconnections [33]. Cache misses are reduced by decreasing the number of invalidations that happen when write operations are performed on shared data [38]. For read operations, the effective cache size is increased by reducing the replication of cache lines on multiple caches [13], which also reduces cache misses. The locality of main memory accesses is increased by mapping data to the NUMA node where it is most accessed. 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. Due to these optimizations, sharing-aware thread and data mapping is able to improve both performance and energy efficiency [22].

Many previous approaches in the area of sharing-aware mapping focus either on thread mapping [5] or data mapping [4, 37, 46], but perform them separately only. Additionally, some mechanisms rely on execution traces to perform a static mapping [37], which has a high overhead [8] and cannot be used if the sharing behavior of the application changes between executions. Other approaches require source code annotations [11] or use indirect information about the memory access pattern and a small number of samples [5, 22, 29], which can result in less accurate mappings.

In this paper, we propose the *Sharing-Aware Memory Management Unit* (SAMMU), which uses the virtual memory implementation in hardware and software to detect the memory access pattern of a parallel application. SAMMU modifies the memory management unit to analyze the memory access behavior, which is used to perform online sharing-aware thread and data mapping. To the best of our knowledge, SAMMU is the first mechanism that keeps track of the number of memory accesses to each page during TLB access, generating memory access patterns more accurate than related work. SAMMU requires no changes to the application or its runtime system, and needs no previous information about application behavior. Our evaluation shows that SAMMU provides substantial performance and energy consumption improvements. Compared to previous mechanisms, SAMMU achieves better improvements for most applications.

This paper is an extension of our previous work [17]. The main improvements provided in this paper are the the following:

- We provide a more detailed explanation of how SAMMU works.
- We perform experiments in different platforms: a full system simulator and a machine with two NUMA nodes.
- We evaluate the energy consumption improvements that can be achieved by SAMMU.
- We run experiments with more benchmarks (PARSEC in addition to the NPB).

- We include an analysis of how SAMMU behaves under different scenarios, varying several architectural parameters.
- We include the mathematical foundations on which SAMMU is based.

This paper is organized as follows: SAMMU is detailed in Sec. 2. The experimental methodology is explained in Sec. 3. Performance and energy results are presented in Sec. 4. Related work and a performance comparison to four algorithms from the state of the art are discussed in Sec. 5. Concluding remarks and future work are presented in Sec. 6. Appendix A adds an analysis of a specific function in SAMMU.

2 SAMMU: A Sharing-Aware Memory Management Unit

An ideal mapping mechanism would need to know the future memory access behavior of threads to make the best mapping decisions. Nevertheless, knowing with 100% accuracy which pages each thread will access in the future is impossible for most applications. We work to overcome this issue by basing decisions on access locality principles. They give us the idea that a thread has a higher probability to access a page in the future if it accessed this same page in the near past.

We propose to monitor memory accesses in the memory management unit (MMU) of processors, which is present in computer systems that support virtual memory (where the MMU is used to translate virtual addresses to physical ones). To perform the translation, the operating system stores page tables in the main memory, which contain the physical address and metadata of each memory page. In most operating systems, threads of a parallel application share the same page table. A special cache memory, the Translation Lookaside Buffer (TLB), is used to speed up the address translation. In multicore or multithreaded architectures, each (virtual) core contains its own (virtual) MMU and TLB.

In this section, we introduce SAMMU, which adds sharing-awareness to the MMU to optimize the memory accesses of parallel applications. SAMMU works in the same way for multicore and multithreaded architectures, so we will only refer to cores in the text. We begin with an overview of the concepts of SAMMU in Sec. 2.1. Afterwards, we explain how SAMMU gathers information

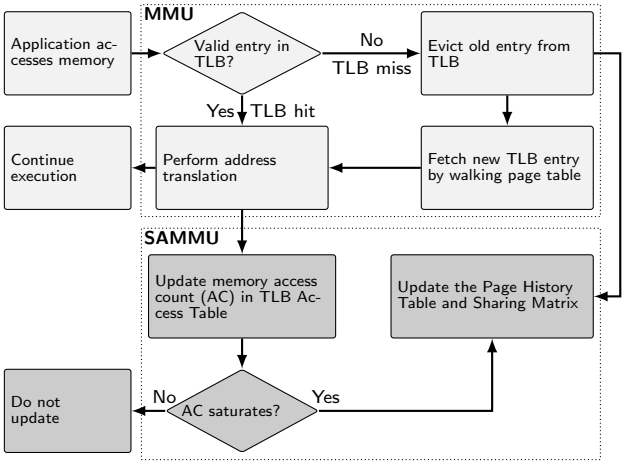


Figure 1: Overview of the MMU and SAMMU.

on memory accesses in Sec. 2.2. We then discuss how we detect the sharing pattern between threads in Sec. 2.3 and the page usage pattern in Sec. 2.4. We present an example of SAMMU's operation in Sec. 2.5, and we discuss implementation and overhead details in Sec. 2.6, Sec. 2.7 and Sec. 2.8, respectively.

2.1 Overview of SAMMU

The default operating system scheduler does not take the sharing pattern of parallel applications into account when assigning threads to cores. Therefore, SAMMU, besides detecting the memory access pattern, also needs the kernel to use this information during scheduling. To solve both issues, SAMMU is implemented partially in hardware and partially in software. The hardware part of our proposal is responsible for detecting the memory access pattern in the MMU, which is explained in Sections 2.2, 2.3 and 2.4.

A high-level overview of the operation of the MMU, TLB and SAMMU is illustrated in Fig. 1. 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.

SAMMU extends the operation of the MMU in two ways, both happening in parallel to the normal operation of the MMU without stalling application execution:
1. SAMMU counts the number of times that each TLB entry is accessed from the local core. This enables the collection of information about the pages accessed by each thread. We store these access counters, one per TLB entry, in a table inside the MMU that we call *TLB access table*.
2. On every TLB eviction or when an access counter saturates, SAMMU analyzes statistics about the page and stores them in two separate structures in main memory. The first structure is the *sharing matrix (SM)*, which estimates how much threads share data. The second structure is the *page history table*, which contains information about the threads and NUMA nodes that accessed each page. This table is indexed by the physical page address, and each of its entries has three fields: (i) *access threshold* (AT), which defines the minimum number of memory accesses required to update the statistics; (ii) *sharers vector* (SV), which contains the ID of the last threads that accessed the page; (iii) *NUMA counters* (NC), which estimates the number of accesses from each NUMA node.

The software part of our proposal uses the information detected by the hardware to perform thread and data migration. Our algorithm calculates thread mapping as explained in Section 2.7.1. To perform thread migration according to the calculated mapping, we use the infrastructure already present in the kernel. For data mapping, pages are migrated to the NUMA node indicated by statistics gathered by the hardware, as explained in Section 2.4, and this also uses the infrastructure already present in the kernel for migration.

2.2 Gathering Information about Memory Accesses

SAMMU gathers memory access information by counting the number of memory accesses to each page in the TLB of each core. To do so, we add a saturating *access counter* (AC) to each TLB entry in the TLB access table. When an AC saturates or its TLB entry gets evicted, SAMMU collects the information and updates the page history table entry of the related page. To filter out the information of threads that perform only a few accesses to a page, each page has an *access threshold* (AT) in the page history table that specifies the minimum number of memory accesses required to update its information. We use an adaptive AT per page in order to handle each

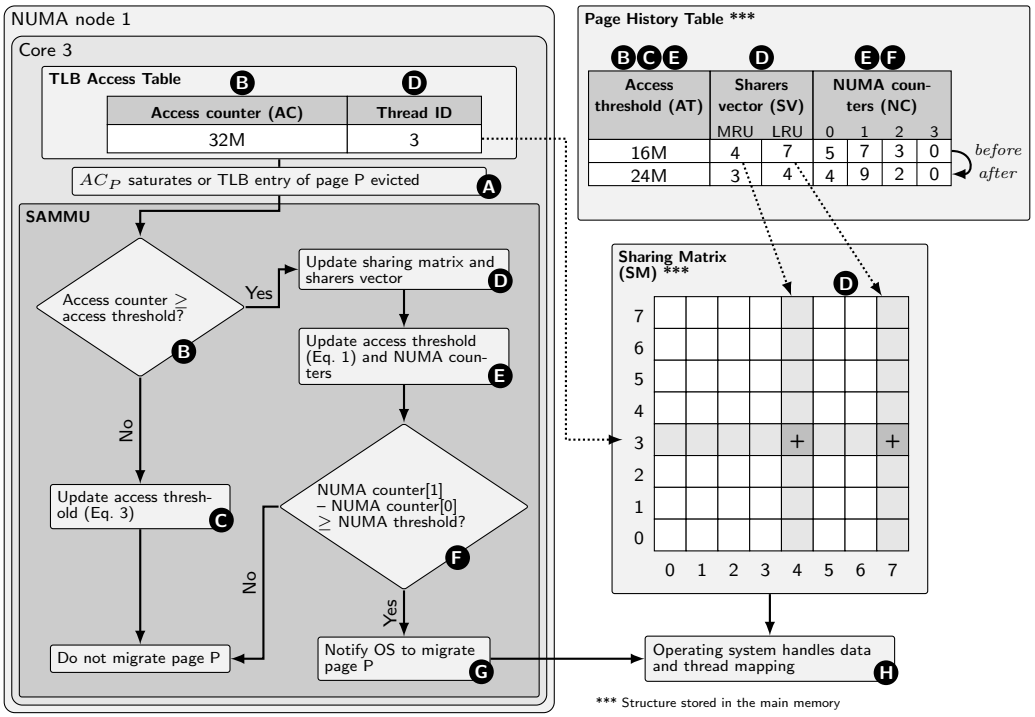


Figure 2: Operation of SAMMU. In this example, the system consists of 4 NUMA nodes with 2 cores each. The NUMA threshold is 4. Page P is initially located on NUMA node 0. In this example, thread 3 (core 3, NUMA node 1) saturates the AC of page P.

page's different access behavior, which would not be possible if a pre-defined or global threshold were to be used.

SAMMU updates the mapping-related statistics of a page only on two situations: when an AC related to a TLB entry containing the page's information saturates; or if the page is evicted from a TLB and the number of memory accesses registered in the AC of this TLB entry is greater than or equal to the AT of the page (a value smaller than AT means that the thread does not use the page enough to influence its mapping).

Fig. 2 shows an example of SAMMU's operation. SAMMU automatically adjusts the access threshold of a given page (which starts at zero) based in two cases (Fig. 2-B):

Case 1 (AC saturates or $AC \geq AT$): When AC saturates or its value is greater than or equal to its AT on a TLB eviction (Fig. 2-B,D), AT is updated with the average value of AC and AT, as illustrated in Eq. 1. Additionally, the mapping statistics are updated, as will be explained in Sec. 2.3 and Sec. 2.4. It is important to note that, since we use the same number of bits to store AC and AT, when AC saturates, it will be greater than or equal to AT.

$$AT_{new} = \frac{AT + AC}{2}, \quad AC \geq AT \tag{1}$$

Case 2 ($AC < AT$): In the second case, when the number of memory accesses registered by AC during a TLB eviction is lower than its AT (Figure 2-B,C), we update AT in such a way that NUMA nodes with a small number of accesses to the page would have a lower influence

on the threshold. In this situation, no mapping statistics are updated, only AT. This situation requires an update function $f(\cdot)$ that subtracts a value from AT:

$$AT_{new} = AT - f(AC), \quad AC < AT \tag{2}$$

Function $f(\cdot)$ must have three properties (we consider AT as a constant):

- $f(0) = 0$, which means that if there were no accesses to a page, its AT would not be changed. In other words, if a thread does not access the page, it should not influence that page's AT.
- $f(AT) = 0$, which means that if a thread were to perform a number of accesses exactly equal to AT, we would still like to keep this AT with no changes.
- $f(AC) = k$, where $0 < AC < AT$ and $0 < k < AT$. This means that, for all values of AC between 0 and AT, we want to reduce the value of AT but still keep its value higher than zero.

As no linear function can provide the properties required by function $f(\cdot)$ (equal to zero in the extremities but larger than zero for other values), we chose a quadratic function. We could use other polynomial functions, but since we need to implement this in hardware, we want to use the simplest function possible. We use Eq. 3, which guarantees that AT will never be decreased by more than 25% at each update. In Appendix A, we provide a detailed explanation on how to find this equation. Further details on how we implement Eq. 3 in hardware are given in Sec. 2.6.1.

$$AT_{new} = AT - \frac{AT - AC}{AT/AC}, \quad AC < AT \tag{3}$$

Lastly, thread and data mapping have complex interactions with each other, which gives rise to multiple questions, such as *should we migrate threads to the core closer to the NUMA node that contains most of the data accessed by the corresponding thread or should we migrate the pages to the NUMA node closer to the cores that most access them?* In order to handle these kinds of questions and to reduce the complexity of mapping decisions, SAMMU handles thread and data mapping separately. These mappings are explained in the next sections.

2.3 Detecting the Sharing Pattern for Thread Mapping

In order to detect the sharing pattern between threads, SAMMU identifies the last threads that accessed each memory page. To obtain this information, SAMMU adds a small *sharers vector* (SV) to each page history table entry. Each SV stores the identifiers of the last threads to access a given page and it overwrites old entries every time new information is available (Fig. 2-D). This provides temporal locality to the detection of the sharing pattern.

SAMMU also keeps a *sharing matrix* (SM) in main memory for each parallel application to estimate the number of accesses to pages that are shared between each pair of threads. This information represents the affinity between threads to be provided to a thread mapping algorithm. In order to be able to update a SM, SAMMU stores the ID of the thread that accessed each TLB entry in the TLB access table. Other additions to the processor's architecture include control registers containing the memory address and dimensions of the sharing matrix, and the ID of the thread being executed, all of which must be updated by the operating system.

When SAMMU is triggered for a certain page by thread T (Fig. 2-A), it accesses the SV of the corresponding page history table entry. If the access counter is greater than or equal to the

access threshold (Fig. 2-B), SAMMU increments the sharing matrix in row T for all the columns that correspond to an entry in the SV (Fig. 2-D).

$$SM[T][SV[i]] = SM[T][SV[i]] + 1 \tag{4}$$

We can notice in Eq. 4 that each line of SM is accessed by its corresponding thread only (in our case, thread T), which minimizes the impact to the memory coherence protocols. Finally, SAMMU inserts thread T into the SV of the evicted page by shifting its elements, such that the oldest entry is removed.

2.4 Detecting the Page Usage Pattern for Data Mapping

In order to identify the page usage pattern of a memory page for mapping reasons, SAMMU requires the addition of a vector to each page history table entry. The vector, which we call *NUMA counters* (NC), has N elements for a system with N NUMA nodes. NC employs saturating counters to count the relative number of accesses from different NUMA nodes to each page. The initial value of each NC is 0.

SAMMU's page usage pattern detection proceeds in the following manner: when a TLB entry from a core in NUMA node n is selected for eviction or its AC reaches its maximum value (Fig. 2-A), SAMMU reads the corresponding page history table entry. If the number of memory accesses stored in AC is greater than or equal to the threshold AT (Fig. 2-B), SAMMU increments the NUMA counter of node n by V_{add}, and decrements all other NUMA counters by 1 (Fig. 2-E), as in Eq. 5. V_{add} is a control register configured by the operating system, and its value must be at least 2 for its correct functioning (more on this is discussed later in this section). We set V_{add} to 2 as a compromise between two constraints: low values of V_{add} increase the time to adapt to a new access behavior, while high values require a larger number of bits to store and update the NUMA counters in dedicated hardware. Finally, since the NUMA counters are saturated, they do not overflow nor underflow.

$$NC[x] = \begin{cases} NC[x] + V_{add} & \text{if } x = n \\ NC[x] - 1 & \text{otherwise} \end{cases} \tag{5}$$

After updating the values of NC, SAMMU checks if the corresponding page is stored in NUMA node n. If the page is currently mapped to another NUMA node m, SAMMU evaluates if the difference between their NUMA counters is greater than or equal to a global value *NUMA threshold* (NT) (Fig. 2-F), as in Eq. 6. If that is the case, SAMMU notifies the operating system of the page and its destination node n (Fig. 2-G). The NUMA Threshold is set by the operating system according to the number of migrations. If too many migrations are occurring, the operating system must increase NT to reduce the overhead coming from excessive migrations (the higher the NT, the lower the number of page migrations).

$$NotifyOS = \begin{cases} true & \text{if } NC[n] - NC[m] \geq NT \\ false & \text{otherwise} \end{cases} \tag{6}$$

The operating system then handles the migration of the page (Fig. 2-H). The operating system is also responsible for managing any possible inconsistencies generated by a thread trying to access a page that is being migrated.

2.4.1 Additional information about V_{add}

In order to demonstrate why V_{add} must be at least 2, consider the situation where NUMA nodes n and m present a similar access pattern to a page P, the value of V_{add} is 1, and all NC are set to 0. Nodes other than n and m do not access page P.

Whenever a core from node n evicts page P (or an AC of page P saturates), the value of $NC[n]$ would be incremented to 1 and $NC[m]$ would be decremented to 0. Likewise, whenever a core from node m evicts the same page P, the value of $NC[m]$ would be incremented to 1 and $NC[n]$ would be decremented to 0. In this scenario, the mechanism would not be able to detect that nodes n and m present more accesses than the others. However, if the value of V_{add} is 2, $NC[n]$ and $NC[m]$ would present higher values after multiple accesses. Therefore, by incrementing with a larger number, SAMMU is able to handle this access behavior. If more than two NUMA nodes access a page in similar amounts, a value higher than 2 is required.

2.5 Example of the Operation of SAMMU

In the situation illustrated in Fig. 2, there are 8 cores, 4 NUMA nodes, the NUMA threshold NT is 4, the sharers vector SV has 2 positions, and V_{add} is 2. AC and AT support values up to 32M. SAMMU is configured to support up to 8 threads per parallel application. Consider that page P is initially located on NUMA node 0. If thread 3, which is being executed on core 3 (NUMA node 1), saturates the AC of page P (Fig. 2-Ⓐ), SAMMU accesses the page history table entry of page P.

Since page P was accessed 32M times and its AC saturated, AT (16M) will be updated to 24M following Eq. 1 (Fig. 2-Ⓑ,Ⓔ). For the sharing pattern, SAMMU checks the sharers vector SV, and finds the IDs of threads 4 and 7. It will then increment cells $(3, 4)$ and $(3, 7)$ of the sharing matrix by 1, and store thread 3 in SV (Fig. 2-Ⓓ). This will result in the removal of thread 7 from SV.

Regarding the page usage pattern (Fig. 2-Ⓔ), the NUMA counter of node 1 will be incremented by 2, and all others will be decremented by 1. The counter of node 3 remains as 0 because the counter is saturated. Since the difference between $NC[1]$ (node of core 3) and $NC[0]$ (current node that stores page P) is greater than or equal to the NUMA threshold (4) (Fig. 2-Ⓕ), SAMMU notifies the operating system to migrate page P to node 1 (Fig. 2-Ⓖ,Ⓗ).

Suppose now that page P was accessed 4M times (instead of 32M) and was selected for a TLB eviction. In this situation, no updates to the sharing matrix and NUMA counters would be performed, since AC would be lower than the access threshold (16M). In this case, the access threshold would be updated to 13M following Eq. 3 (Fig. 2-Ⓒ).

2.6 Hardware Implementation Details

2.6.1 Detection Mechanism Implementation

SAMMU can be implemented in several ways. We implemented it in a way to handle only one event at a time to simplify its hardware requirements. If a TLB eviction happens or an AC saturates while SAMMU is already handling another event, the event is ignored. If more events were considered, the accuracy would be higher, but with a trade-off of a more complex hardware and higher overhead and power consumption. However, since these events are very frequent, we do not expect a significant impact on accuracy by discarding some of them. We evaluate this in Section 4.3.

To count the number of accesses to each TLB entry, we added the TLB access table to the MMU. In order to implement the TLB access table as a scratchpad, without the need for

tag comparison, we added to each TLB entry a static unique identifier. When a TLB entry is accessed, its identifier is returned along with the page information. SAMMU then uses this identifier as an index in the TLB access table.

Regarding the updates to the AT in the page history table, since Eq. 3 requires division operations, we update the access threshold using the approximation shown in Eq. 7 instead, where $\gg$ represents the bit shift operation. The H function returns the position of the highest bit set to 1 in its parameter.

$$AT_{new} = AT - \left[\left(AT - AC\right) \gg \left(H(AT) - H(AC)\right)\right] \tag{7}$$

2.6.2 Notifying the Operating System About Page Migrations

In order to notify the operating system when a page needs to migrate, SAMMU can either introduce an interruption or save in memory a list of pages to be migrated and their destination NUMA nodes. To avoid interrupting the operating system too frequently, we chose the latter. The operating system periodically checks this list and performs the migrations.

2.6.3 Handling Context Switches and TLB Shootdowns

In context switches, if the context of a core changes to another process and thereby to another memory address space, the TLB may be flushed depending on the architecture and operating system. In case the TLB gets flushed, SAMMU needs to flush the TLB access table. In case the TLB is not flushed, SAMMU still detects the sharing correctly, since it stores the identifier of the thread in the TLB access table. The content stored in the page history table is not affected. When individual TLB entries are flushed by TLB shootdowns due to changes on its page table entry, the flushed TLB entry can be tracked by SAMMU. Nevertheless, both context switches and changes on page table entries are much less frequent than TLB evictions and have no significant impact on the accuracy of SAMMU.

2.7 Software Implementation Details

2.7.1 Performing the Thread Mapping

How often the thread mapping is performed depends on the operating system, as SAMMU is responsible only for detecting the sharing patterns. A simple implementation consists of analyzing the sharing matrix and then mapping threads from time to time.

We model thread mapping as a graph problem. There are two graphs: the application graph and the machine hierarchy graph. In the application graph, vertices represent threads and edges the affinity between threads. The affinity between each pair of threads is the value of the corresponding pair of threads in the sharing matrix. In the machine hierarchy graph, vertices represent components of the memory hierarchy, such as the cores and caches, and edges represent the links between components. The mapping algorithms of Scotch [41] v6.0 were used to map the application graph to the machine graph, generating the thread mapping. We check for thread migrations every 100ms. To reduce the influence of old values, we apply an aging technique in the sharing matrix every time the mapping mechanism is called by multiplying all of its elements by 0.75. We evaluated values between 0.6 and 0.95, but the results were not sensitive to this value.

After generating the thread mapping, to actually migrate the threads, SAMMU makes use of the thread migration functions already present in all modern operating systems. The Linux kernel provides the function `sched_setaffinity` internally, which migrates threads to different cores and handles all steps required for the migration.

2.7.2 Performing the Data Mapping

To perform the data mapping, the operating system receives from the detection mechanism of SAMMU which pages should be migrated along with the target NUMA node. Since operating systems already implement functions to migrate pages between NUMA nodes, SAMMU only needs to call these functions to perform the migration. The Linux kernel provides the `unmap_and_move` function internally, which moves pages between NUMA nodes and handles all the steps related to to the migration, including copying the data and modifying the page table.

2.7.3 Increasing the Supported Number of Threads

The operating system starts an application configuring SAMMU 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 the supported number of threads during execution. To do that, it must allocate a new sharing matrix SM and copy the values from the old SM. It must also update the contents of all sharers vectors SV to use the new number of bits per entry. Since this is an expensive procedure, we recommend to avoid it by configuring SAMMU to support a large number of threads from the beginning. For all systems and applications evaluated, configuring SAMMU to support 256 threads was enough to avoid this procedure.

2.8 Overhead

SAMMU's overhead consists of storage space in the main memory, circuit area, and execution time. We detail each of them in the next sections.

2.8.1 Memory Storage Overhead

The main memory stores the page history table and sharing matrix. For the configuration used in our experiments and shown in Table 1 (Sec. 3.1), each entry of the page history table would require 8 Bytes. The page history table space overhead would be 0.2% relative to the total main memory. The sharing matrix would require 256 KByte, each of its elements with 4 Bytes. In this configuration, SAMMU can track up to 256 threads per parallel application. To support larger systems, we would need only to use more space per page history table entry, allocating more NUMA counters and sharers vector entries, and a larger sharing matrix.

2.8.2 Circuit Area Overhead

SAMMU's logic is implemented in the MMU of each core. In total it requires 8 adders, 2 subtractors, 5 shifters, and 7 multiplexers of various sizes (from 4 to 32 bits) per core. SAMMU also uses a TLB access table that stores one access counter and one thread ID per TLB entry in static RAM, similar to the TLB itself. The number of bits of each access counter must be enough to store the maximum possible value of the access threshold. In this scenario, the additional hardware required by SAMMU represents $143,000$ transistors per core, which results in an increase in transistors of less than 0.05% in a modern processor.

2.8.3 Execution Time Overhead

The additional hardware of SAMMU is not in the critical path of the processor because it operates in parallel to the MMU, such that application execution is not stalled while SAMMU is operating. Therefore, the time overhead introduced by SAMMU consists of the additional memory accesses to update the page history table and sharing matrix, which depend on the

TLB miss rate. To keep the overhead of these memory accesses low, SAMMU does not lock any structure before its update. The sharing matrix does not need to be locked since each row is updated only by one thread. Even in case of reusing sharing matrices' rows due to dynamic thread creation and execution, no locking mechanism is necessary, since a thread needs to stop executing, and thereby be unscheduled, before it is destroyed. For a page history table entry, a race condition can happen in case threads evict the same page (or the corresponding ACs saturate) at the same time. This race condition would not cause the application to fail, just a slight reduction in accuracy. Since the time SAMMU takes to update a page history table entry is small, this race condition is a rare event. On the software level, the operating system introduces overhead when calculating the thread mapping, and when migrating threads and pages. The measured execution time overhead from both hardware and software are shown in Sec. 4.3.

3 Experimental Methodology

In this section, we describe the experiments we performed to evaluate SAMMU in a full system simulator in Sec. 3.1, and on real machines in Sec. 3.2. Afterwards, we show which benchmarks we used as workloads in Sec. 3.3. Table 1 summarizes the parameters used for SAMMU, the simulated machine, and the real machines.

3.1 Evaluation in a Full System Simulator

We implemented SAMMU in the Simics simulator [34], extended with the GEMS-Ruby memory model [39] and the Garnet interconnection model [1]. 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 [45]

Table 1: Configurations of the two types of experiments that we run, on the Simics full system simulator, and on the real machines with SAMMU traces generated by Pin. The SAMMU configuration is the same for both types of experiments.

Machine	**Parameter**	**Value**
SAMMU	Structure sizes	AC, AT: 32 bits, SV: 2x 8 bits, NC: 4x 4 bits
	Sharing matrix	256 threads, 4 Byte element size
	Control registers	Support up to 256 threads, $V_{add} = 2$, $NT = 10$
Simics	Processors	4x 2 cores, 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 , 8 banks/rank, 2 ranks/DIMM
	Interconnection	1/40 cycles latency (intra/interchip), 64/16 Byte bandwidth (intra/interchip)
Pin	L1 TLB	64 entries, 4-way, shared between 2 SMT-cores
	L2 TLB	512 entries, 4-way, shared between 2 SMT-cores
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	32 GByte DDR3-1600, 4 KByte page size
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	128 GByte DDR3-1333, 4 KByte page size

and the memory timings from JEDEC [27]. 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, as done in [19].

We compare the results in the simulator to its default policy (interleaved page mapping) and to an oracle mapping. The **oracle mapping** directly considers all memory accesses, while SAMMU considers them indirectly on TLB evictions. The oracle performs data and thread mapping online. For data mapping, we count the number of memory access from the threads of each NUMA node to each memory page, and each page is mapped to the NUMA node running the threads that access it the most. For thread mapping, we keep a sharing matrix that is updated on every memory access. The oracle thread mapping is applied following the same methodology described in Sec. 2.7.1. All results in the simulator are normalized to the default mapping. Each experiment in the simulator is executed only once due to the low simulation speed and Simics' deterministic nature.

3.2 Evaluation on Real Machines

The experiments were 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 run version 3.8 of the Linux kernel. Information about the hardware topology was gathered using Hwloc [12]. Besides performance, we measured L3 cache misses per thousand instructions (MPKI), interchip interconnection traffic (QuickPath Interconnection) and energy consumption (RAPL hardware counters [25]) using Intel PCM tool [26].

Since SAMMU is an extension to the current MMU hardware and we are unable to change the circuits inside an Intel processor, we need to find another way to provide to the runtime environment the same information that SAMMU would generate. To generate this information for an application, we execute the application in a simulated environment using the Pin [6] dynamic binary instrumentation tool. 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 SAMMU on real machines, the mapping information generated in Pin is fed into the mapping mechanism in the runtime environment.

The runtime mapping mechanism was implemented in user-space as a dynamic library that is to the application and provides wrappers to `libgomp` functions to track the parallel phases. We use the parallel phases to synchronize the mapping information from Pin to the runtime mapping mechanism. Whenever a new parallel phase begins, the mapping information generated at that point of the execution is used to map threads and data via the `sched_setaffinity` and `move_pages` syscalls, respectively.

All experiments in the real machines were executed 30 times. We show average values as well as a 95% confidence interval calculated with Student's t-distribution. We compare the results of SAMMU to the default mapping performed by the operating system, to random static mappings, and to an oracle mapping. The operating system mapping is the original scheduler and data mapping policy 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 [8]. The oracle mapping is similar to the one described in Section 3.1. The main difference is that the memory accesses are fetched from the trace of a previous execution. Also, since the behavior of the NPB applications used in our evaluation on the real machines is static, the oracle

Table 2: Input sizes of the benchmarks used on each system.

Benchmark suite	Benchmark	Input size in the simulator	Input size on the real machines
NPB	BT, LU, SP, UA	W	A
	CG, EP, FT, IS, MG	A	B
PARSEC	canneal	–	simmedium
	All except canneal	–	simlarge

mapping is applied statically in this experiment. All results in the real machines are normalized to the operating system mapping.

3.3 Workloads

We chose the workloads of the OpenMP implementation of the NAS parallel benchmarks (NPB) [28], v3.3.1, and the PARSEC benchmark suite [9], v3.0, for our experiments. A description of the data sharing patterns of these applications can be found in [23]. We configured the benchmarks to run with one thread per virtual core, although some PARSEC benchmarks execute with multiple threads per virtual core.

For the evaluation on the real machines, the evaluated applications must present the same sharing and page usage patterns across different executions, as well as keep 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. DC and the applications from PARSEC do not keep the same memory address space due to dynamic memory allocations. This makes the information generated in Pin unreliable to guide mapping in future executions.

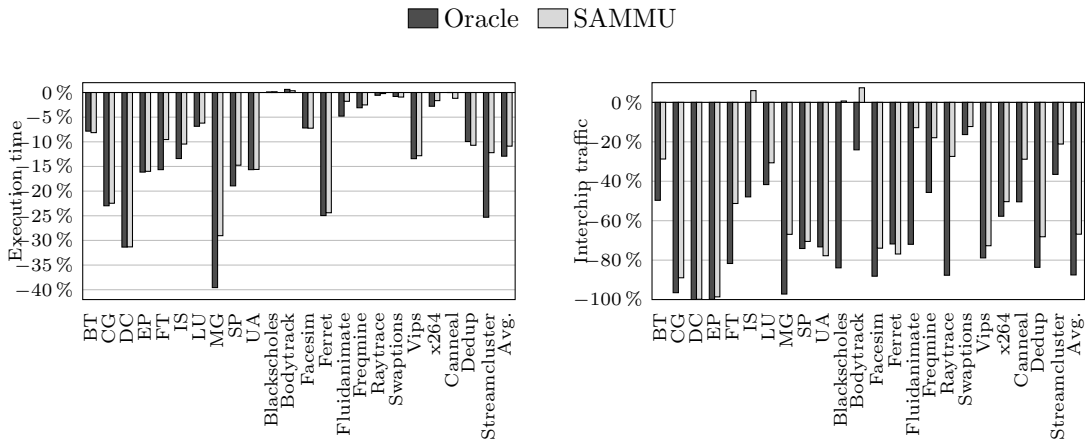
Input sizes were chosen to provide similar total execution times and feasible simulation times. Regarding NAS, benchmarks BT, LU, SP and UA were executed using input size A in Pin and the real machines, and input size W in Simics. Benchmarks CG, EP, FT, IS and MG were executed using input size B in Pin and the real machines, and input size A in Simics. DC was executed with input size W in Simics. Regarding PARSEC, the input size used in canneal was *simmedium*, and *simlarge* for all others. Table 2 summarizes the input sizes used in each system.

4 Experiments and Results

This section presents our collection of experiments with SAMMU. As discussed in the previous section, we perform experiments on two system types, an evaluation in the Simics system simulator with SAMMU implemented in Simics, and an evaluation on real machines where the behavior of SAMMU is simulated with Intel Pin prior to running on the machines. The experiments include performance results related to total execution time and interconnection traffic in Sec. 4.1, energy consumption results in Sec. 4.2, and overhead results in Sec. 4.3. An exploration of the performance of SAMMU over scenarios with different architectural parameters in presented in Sec. 4.4. Finally, a comparison to algorithms from the state of the art in the literature is left to Sec. 5.2.

4.1 Performance Results

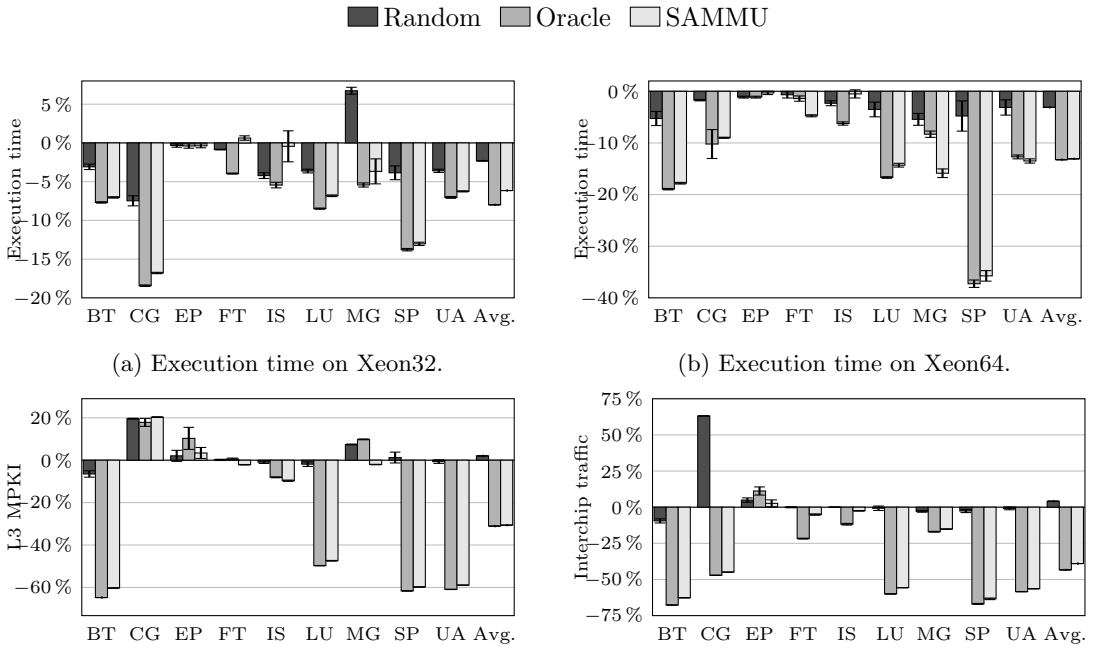
The execution time and interchip interconnection traffic measured in Simics are shown in Fig. 3a and 3b. For the real machines, the results regarding execution time can be found in Fig. 4a



(a) Execution time.

(b) Interchip interconnection traffic.

Figure 3: Performance results in Simics, normalized to the OS. Lower values are better.



(a) Execution time on Xeon32.

(b) Execution time on Xeon64.

(c) L3 MPKI on Xeon64.

(d) Interchip interconnection traffic on Xeon64.

Figure 4: Performance results, normalized to the OS. Lower values are better.

and 4b, L3 cache misses per thousand instructions (MPKI) in Fig. 4c, and interchip traffic in Fig. 4d. In these figures, we also show the average improvements as the geometric means. For all of these results, the biggest the reduction, the better. The sharing patterns of a subset of our workloads are illustrated in Fig. 5. These applications whose sharing patterns are shown here were selected because they represent the different patterns found in all applications.

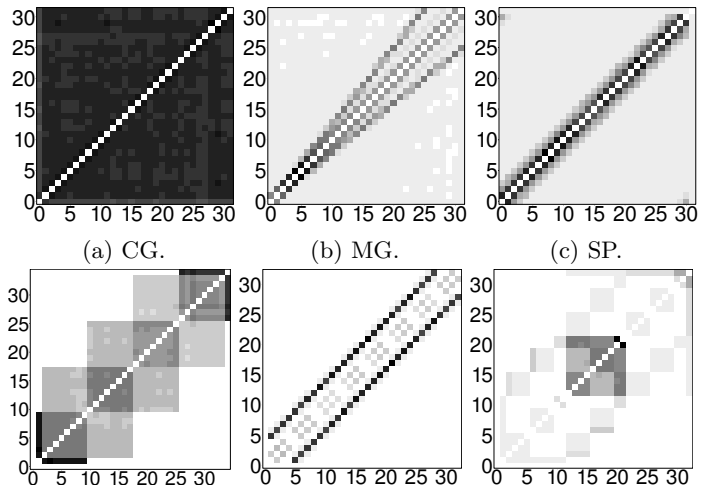
In applications whose pages are shared within a small subgroup of threads, mapping presents a high potential for performance improvement. As can be seen for SP in Fig. 5(c), most sharing happens between neighboring threads, which is very common in parallel applications that use domain decomposition. In MG and Fluidanimate, the sharing between more distant threads is more evident than in the other applications. In Ferret and Dedup, which have a pipeline sharing pattern, threads that share data in the pipeline form sharing clusters.

The threads of tested benchmarks were able to benefit from the shared cache memories and faster interconnection when mapped nearby in the memory hierarchy, as well as from the faster access to the shared pages now mapped to their local NUMA nodes. In general, the effect of the sharing-aware thread and data mapping of SAMMU is a reduction of cache misses and interchip traffic, observed both in LU and SP. SP presented the highest improvements in the real machines, with an execution time reduction of up to 35.7% on Xeon64.

Thread and data mapping interact synergically [21], such that the benefits of thread and data mapping alone are often lower than of the combined mapping. This happens because, if threads that share a lot of data are not mapped to the same NUMA node, even if the private data of each thread is mapped to their local NUMA node, all the shared data will require remote memory accesses from the threads that are not local. By mapping these threads to the same NUMA node, the shared data will be local to all threads, providing a higher performance improvement. These interactions between mapping types are a well-researched phenomenon and we therefore do not perform an in-depth evaluation of them in this paper. Due to these reasons, in our evaluation, we perform both mappings together, benefiting from the positive interaction between thread and data mapping. To illustrate how thread mapping also affects data mapping, consider MG. MG's sharing pattern indicates that it has a high potential for thread mapping. However, its reduction of interchip traffic is higher than its reduction of cache misses. The reason is that the threads that share data were mapped on the same NUMA node, thus reducing interchip traffic as their shared pages were also mapped to their NUMA nodes. Cache misses were not reduced to the same degree. Therefore, although MG shows a high potential for thread mapping, we are able to observe this by looking at interchip traffic, not at cache misses.

Some applications do not present a sharing pattern that benefit from thread mapping. Examples of this type of application are CG and Vips. The sharing pattern of CG is illustrated in Fig. 5a(a), where we can observe that each pair of threads has a similar affinity. Therefore, no thread mapping is able to improve the usage of cache memories. This is the reason that SAMMU, or even the Oracle mapping, did not decrease the number of cache misses for CG. However, due to the data mapping, SAMMU improved the memory access locality in CG such that the volume of interchip traffic was decreased by up to 88.9%, leading to a performance improvement of up to 22.5%. Likewise, interchip traffic and execution time of Vips were reduced by 72.7% and 12.8%, respectively.

No performance improvements are expected for some applications, either by thread or data mapping. For instance, Swaptions has a sharing pattern similar to the one of Vips. However, the memory usage of Swaptions is much lower than the one of CG, such that almost all its data fits into the caches. Therefore, although we decrease interchip traffic by 12.2% in Swaptions, the absolute reduction is very small, since its interchip traffic corresponds to only about 9.5% of CG's, and is not affected by data mapping. Another example of application that does not usually benefit from different mappings is EP. It is a CPU bound application [28] with almost



(a) CG. (b) MG. (c) SP.

(d) Ferret. (e) Fluidanimate. (f) Dedup.

Figure 5: Sharing patterns of some applications. Axes represent thread IDs. Cells show the number of accesses to shared pages for each pair of threads. Darker cells indicate more accesses.

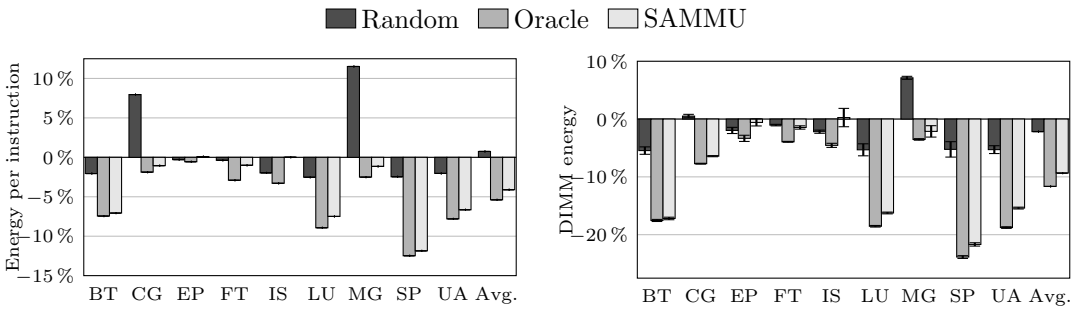
no data sharing among its threads. The reason that no performance improvement was achieved in the real machine but we achieved significant gains in Simics is that the cache memory space in Simics is much smaller. Similarly to Swaptions, all of EP's data fits into the caches of the real machine. However, this was not the case in Simics, such that the data mapping provided some performance improvements to the application.

SAMMU was able to reduce the number of cache misses and the traffic in the interconnections significantly. L3 MPKI (misses per thousand instructions) was reduced by an average of 30.6%. Interchip traffic was reduced by an average of 66.8% and 39.0% on Simics and Xeon64, respectively. The execution time was reduced by an average of 6.1% on Xeon32, 13.1% on Xeon64 and 10.9% on Simics. This smaller reduction in execution time when compared to MPKI happens because a better mapping directly influences the number of cache misses and traffic on the interconnections, while the execution time is influenced by several other factors.

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, in case of pages shared by multiple threads.

It can be observed that improvements in Simics and Xeon64 are higher than on Xeon32. This comes from the fact that Simics and Xeon64 have memory hierarchies with more cache memories and NUMA nodes, so the probability of correctly mapping a page to a node without any knowledge of the page usage pattern is only 25% on them, while it is 50% on Xeon32. For this reason, a thread and data mapping that takes the memory hierarchy into account has a higher performance impact on machines with more complex memory hierarchies.

SAMMU demonstrated its effectiveness with results similar to the oracle mapping. In most cases, it performed significantly better than the random mapping. This shows that the gains



(a) Energy / instruction on Xeon32.

(b) DIMM energy on Xeon32.

Figure 6: Energy consumption results, normalized to the OS. Lower values are better.

compared to the operating system are not due to the unnecessary migrations introduced by the operating system, but due to a more efficient usage of resources.

4.2 Energy Consumption Results

We have measured the energy consumed per instruction and DIMM energy consumption, which are shown in Fig. 6. Energy was only measured in Xeon32 because Xeon64 does not support RAPL. We also measured the total energy consumption, whose behavior is similar to the execution time results, with the biggest reductions for BT, CG, LU, SP and UA. The other applications show no difference or a small reduction of energy consumption. Additionally, we observed that the DIMM energy was reduced more than the processor energy, 9.3% and 5.0% on average respectively, because a sharing-aware mapping has more influence in the memory than in the processor. The energy per instruction results show that our mechanism not only saves energy by reducing the execution time, but also by providing a more efficient execution, which is an important goal for future Exascale architectures [47]. Energy per instruction was improved by 4.1% on average, and up to 11.9% for SP. The average energy per instruction of the applications using SAMMU was 3.7nJ per instruction. The lowest and highest energy per instruction happened for BT and CG, respectively, which required 2.1nJ and 7.4nJ of energy per instruction.

4.3 Performance Overhead of SAMMU

As discussed in Sec. 2.8.3, SAMMU causes an overhead on the execution of the parallel application on the hardware and software levels. We evaluate the hardware overhead by running SAMMU without actually performing thread and data migration, and compare the application execution time to the baseline without SAMMU. For the software overhead, we measure the time spent in the mapping and migration routines. We only show the performance overhead in Simics because the real machines do not physically implement SAMMU.

The overhead caused by the hardware and software required by SAMMU in Simics is shown in Fig. 7. The performance overhead caused by the hardware was 0.27%, due to the introduction of 1.43% additional memory transactions, on average. As explained in Sec. 2.6.1, our implementation of SAMMU is able to handle only one event per core simultaneously, such that SAMMU was able to handle 85.5% of the total number of events. SAMMU required an average of 109 cycles to handle each event. Nevertheless, application execution is not stalled while SAMMU handles an event. IS has the highest overhead due to its large number of TLB misses, introducing more

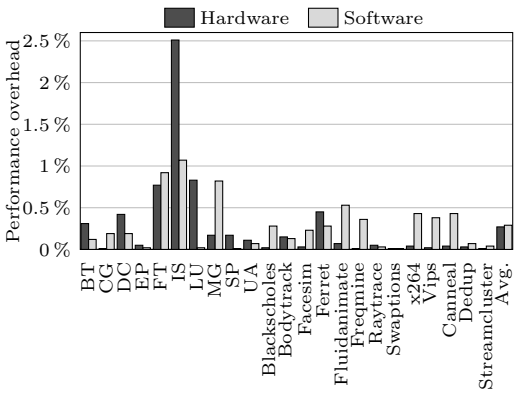


Figure 7: Performance overhead in Simics (% of total execution time). Lower values are better.

memory accesses. Meanwhile, the overhead in the software level was 0.29%, on average. These results show that SAMMU has a very small performance overhead that is easily overcome by its performance benefits, as shown previously.

4.4 Exploring Architectural Parameters

We have analyzed how SAMMU behaves on environments with configurations different from the previous experiments using Simics/GEMS. For that, we varied three important parameters: cache memory size, memory page size and interchip interconnection latency. In the next sections, we show results only for the applications whose sharing patterns are illustrated in Fig. 5, which present different data sharing behaviors.

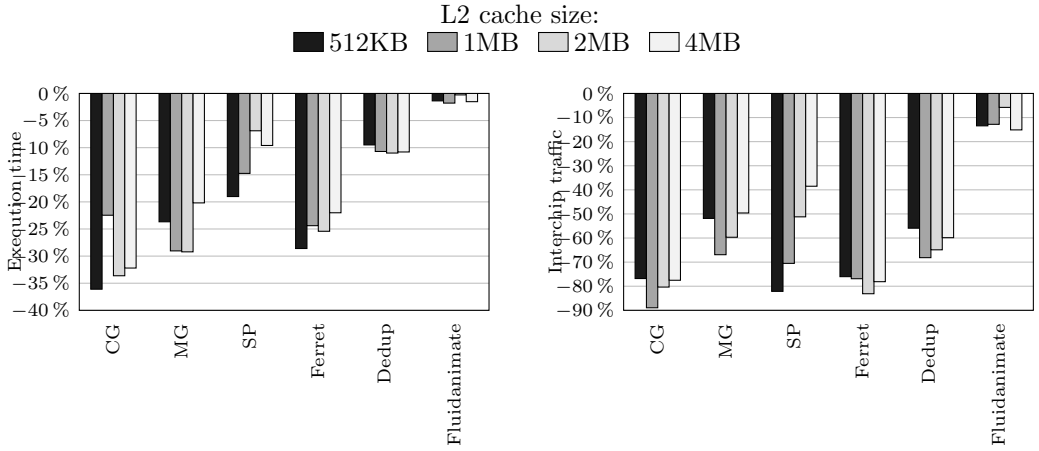
4.4.1 Cache Size Variations

The cache memory size influences the performance improvements obtained with sharing-aware mapping because it affects the amount of shared data that can be cached, and the number of accesses to the main memory. The previous experiments were performed with L2 cache memories with 1 MByte of capacity and a 5 cycles latency. We now show results for caches with 512 KBytes, 2 MBytes, and 4 MBytes, and latencies of 4, 6, and 6 cycles, respectively, as calculated using CACTI [45].

The results of varying the cache size are shown in Fig. 8. They follow the same pattern of the previous experiments: the reduction of execution time depends on the reduction of interchip interconnection traffic. We can observe that SAMMU was able to improve performance for all applications and all cache sizes. This shows that SAMMU can benefit architectures regardless of their cache sizes.

4.4.2 Page Size Variations

Another important parameter for SAMMU is the page size, since it influences the number of TLB misses and, thereby, evictions. In this context, the normalized execution time and interchip interconnection traffic measured with different page sizes are shown in Fig. 9a and Fig.9b (the previous experiments were performed with a 4 KBytes page size). We also show the TLB miss rate



(a) Execution time.

(b) Interchip traffic.

Figure 8: Varying L2 cache memory sizes. Results are normalized to the default mapping with the corresponding cache size. Lower values are better.

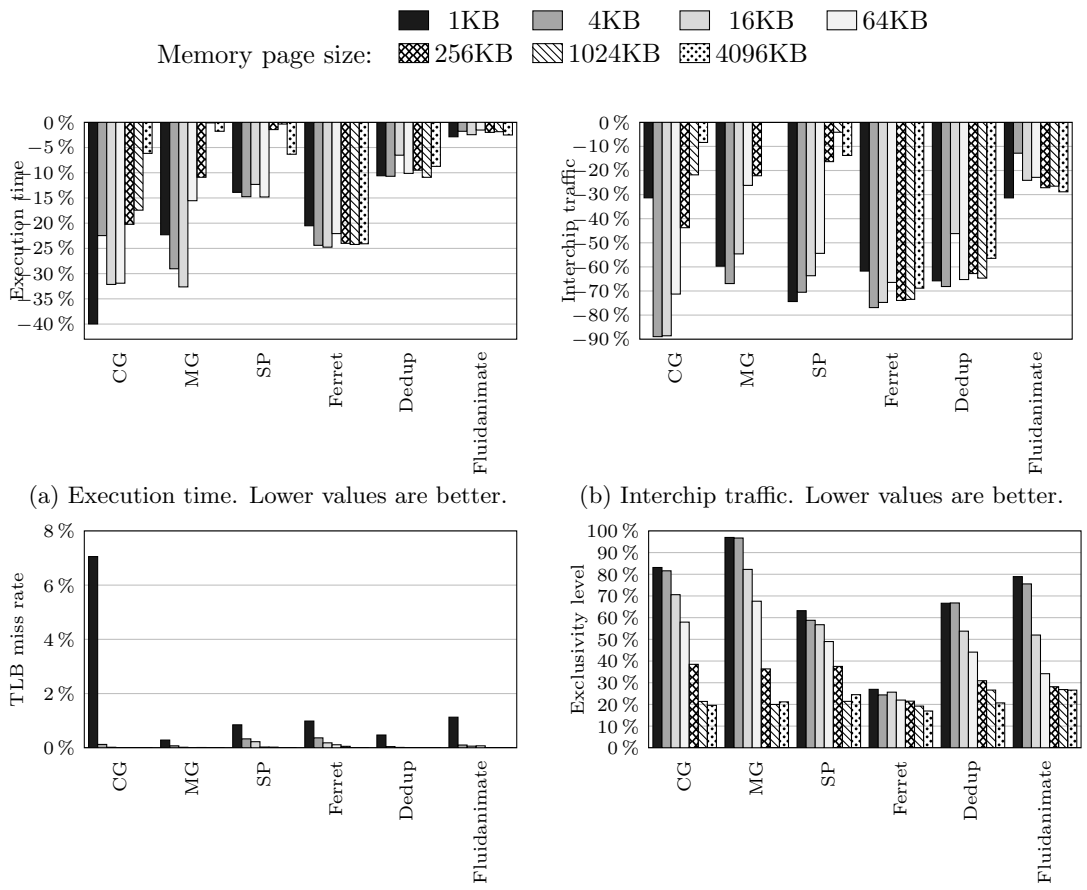
in Fig. 9c, and a metric called exclusivity level, introduced in [22], in Fig. 9d The exclusivity level of a page corresponds to the highest number of memory accesses to the page from a single thread in relation to the number of accesses from all threads. The exclusivity level of an application is a weighted average of the exclusivity of all pages considering the number of memory accesses. The higher the exclusivity level of an application, the higher its potential for sharing-aware data mapping.

TLB miss and eviction rates drop considerably when the page size increases, decreasing the number of updates to the sharing matrix and page history table. Since SAMMU detects the sharing pattern in the page level granularity, increasing page size can also lead to the detection of different patterns. The analysis of the exclusivity level shows that, for all applications except Ferret, the interference of accesses in different offsets is low when the page size is 1 KByte or 4 KBytes, but increases noticeably in larger page sizes. In relation to data mapping, the lower exclusivity level with larger page sizes tends to limit performance improvements. This happens because there are more threads executing in cores from different NUMA nodes that share a page when it is larger, which is reflected in the lower reduction of interchip traffic and, thereby, in the lower performance gain. In some cases, the best performance improvements happened with an intermediate page size, such as for MG. This happened because pages were migrated earlier during the execution, such that the benefits of the improved data mapping were taking effect earlier too.

It is important to note that the memory footprint of the applications used in the simulations is very small due to the small input size required by simulation time constraints. Applications with higher memory footprints keep similar exclusivity levels with larger page sizes [22]. Therefore, SAMMU would maintain similar performance improvements with larger page sizes for applications that use large amounts of memory.

4.4.3 Interchip Interconnection Latency Variations

The interchip interconnection latency influences the time it takes to send cache coherence messages, such as cache line invalidations, as well as cache line transfers between caches, affecting



(a) Execution time. Lower values are better.

(b) Interchip traffic. Lower values are better.

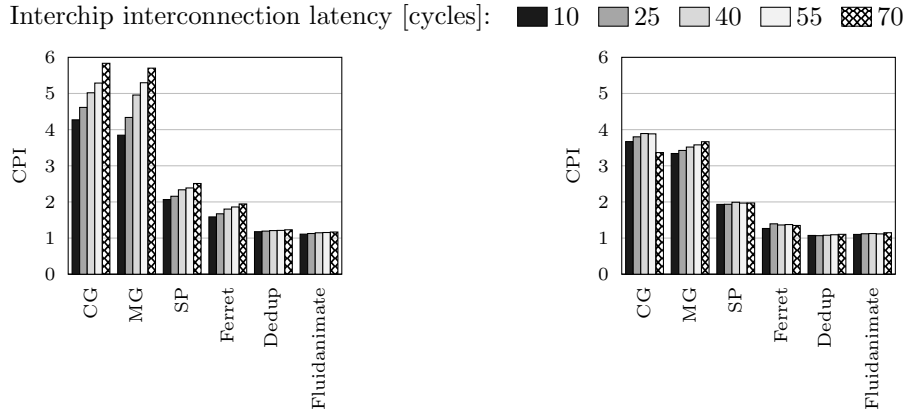
(c) TLB miss rate. Lower values are better.

(d) Exclusivity level. Higher values are better.

Figure 9: Varying memory page sizes. Results of Fig. 9a and 9b are normalized to the default mapping with the corresponding page size.

thread mapping. Regarding data mapping, the interchip interconnection latency influences the time it takes to perform remote memory accesses. Local memory accesses are not affected. In this section, we briefly evaluate how SAMMU behaves with latencies between 10 and 70 cycles (previous experiments used a latency of 40 cycles).

The results obtained with the interchip interconnection latency variation are shown in Fig. 10. In the absolute values, we chose to show cycles per instruction (CPI) instead of execution time due to the high difference of execution time between the applications. We can observe that the CPI measured using the standard operating system mapping increases significantly with the increase of the interchip latency for most applications. On the other hand, the CPI obtained by SAMMU suffers a much lower impact from the latency increase. SAMMU is less influenced by this latency because it is able to reduce the number of coherence messages, cache-to-cache transfers and remote memory accesses.



(a) Absolute CPI of OS.

(b) Absolute CPI of SAMMU.

Figure 10: Varying interchip interconnection latency. Lower values are better.

5 Related Work

In this section, we first describe several related mapping mechanisms and then compare some of them to SAMMU.

5.1 State-of-the-art Thread and Data Mapping

Traditional data mapping policies, such as *first-touch* and *next-touch* [32], 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. A more advanced policy named NUMA Balancing [14] was included in Linux 3.8. 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. However, NUMA Balancing does not detect the sharing pattern between the threads.

Marathe et al. [37] 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 [8]. Trahay et al. [48] propose a tool that generates and analyzes memory traces using hardware counters. A similar technique is used in Marathe and Mueller [36] 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 application's execution. Tikir and Hollingsworth [46] use UltraSPARC III hardware monitors to guide data mapping, but do not perform thread mapping. Their proposal is limited to architectures with software-managed TLBs, while SAMMU focuses on architectures with the more common hardware-managed TLBs, such as x86. Data mapping alone is not able to improve locality when more than one thread accesses the same pages, since threads may be mapped to cores of different NUMA nodes.

Azimi et al. [5] 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. Only thread mapping

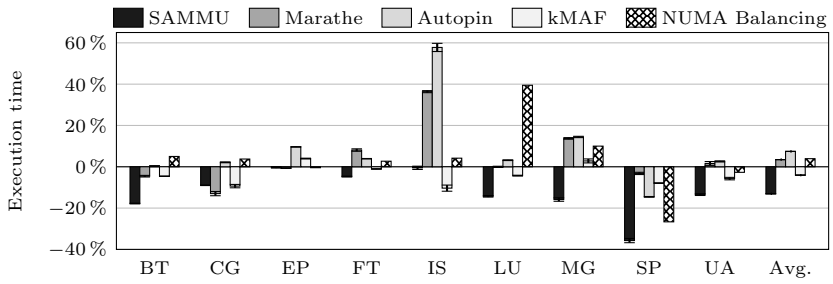


Figure 11: Execution time comparison to related work on Xeon64, normalized to the OS. Lower values are better.

was performed, which does not improve the locality of memory accesses in NUMA architectures. Barghi and Karsten [7] propose a locality-aware scheduler for actor model-based applications which tries to avoid migrating actors to a different NUMA node, without any data migration.

The kMAF affinity framework is proposed in [22]. It performs both thread and data mapping and gathers information from page faults. Lepers et al. [30] developed AsymSched, a thread and memory mapping algorithm that optimizes the bandwidth for communicating threads. They also show that the asymmetry of the interconnections has a big impact on performance. Carrefour [20] is a similar mechanism that uses sampling to detect page usage. Due to its overhead, the authors restrict the mechanism to 30,000 pages, which limits its use to applications with a low memory usage. These mechanisms generate thread mapping information based on a very small number of samples compared to SAMMU, as all memory accesses are handled by the MMU. LAPT [16] and IPM [18] also add hardware to monitor the memory accesses, but do not track all memory accesses as SAMMU, obtaining less accurate information. Some techniques, such as ForestGOMP [11], require annotations in the source code and depend on specific parallelization libraries. Similarly, Ogasawara [40] proposes a data mapping method that is limited to object oriented languages, and Majo and Gross [35] propose a mechanism that works only in Intel Thread Building Blocks. Anbar et al. [2] propose PHLAME, an execution model in which programmers can provide a description of the locality between data and threads, which the runtime uses to perform locality-aware mapping.

The usage of the instructions per cycle (IPC) metric to guide thread mapping is evaluated in Autopin [29]. Autopin itself does not detect the sharing pattern, as it only verifies the IPC of several mappings fed to it and executes the application with the thread mapping that presented the highest IPC. In [42], the authors propose BlackBox, a scheduler that, similar to Autopin, selects the best mapping by measuring the performance that each mapping obtained. When the number of threads is low, all possible thread mappings are evaluated. When the number of threads makes it unfeasible to evaluate all possibilities, the authors execute the application with 1000 random mappings to select the best one. These mechanisms that rely on statistics from hardware counters take too much time to converge to an optimal mapping, since they need to first check the statistics of the mappings. The convergence is usually not possible because the number of possible mappings is exponential in the number of threads. Also, these statistics do not accurately represent sharing and data access patterns.

5.2 Comparison to Related Work

We compare SAMMU on Xeon64 to four previously mentioned techniques: the Marathe [36] data mapping mechanism, Autopin [29], the kMAF affinity framework [22] and NUMA Balancing [14].

We selected these techniques because they can be reasonably run on real hardware. Autopin was executed with 5 mappings: the Oracle mapping and 4 random mappings. After a warm-up time of 500ms, every mapping was evaluated for 150ms. Then, the mapping with the highest IPC was used for the rest of the execution. Autopin, kMAF and NUMA Balancing were directly executed on the real machine. We implemented Marathe using a long latency load profile [36] in Pin and fed the information during the execution of the application in the real machine, as in Sec. 3.2.

Fig. 11 shows the execution time of SAMMU and related work on Xeon64. Values are normalized to the results of the operating system. For CG, Marathe presented slightly better results than SAMMU. This happens because, as previously explained, CG is an application only affected by data mapping, such that SAMMU introduces thread migrations during execution that add overhead. These unnecessary migrations could be avoided if our mapping algorithm presented features to allow migrations only if the detected sharing pattern has high potential for mapping. Autopin, in several executions, selected a mapping different from the Oracle, which shows that indirect metrics are not accurate. Also, its performance improvement is lower than ours because it needs to evaluate several other mappings. The results of NUMA Balancing and kMAF are lower than SAMMU for most of the benchmarks. Due to their sampling mechanism, these mechanisms need more time to detect the memory access behavior and therefore lose opportunities for improvements.

The comparison to the related work shows that mechanisms that perform both thread and data mappings are able to achieve better improvements than mechanisms that perform these mappings separately. Also, it shows that SAMMU, with its hardware support, is able to achieve a higher accuracy than other mechanisms due to the high number of memory accesses considered when detecting the memory access pattern. Thereby, it is able to detect the pattern faster, providing better performance improvements.

6 Conclusions and Future Work

The locality of memory accesses has a high influence on the performance and energy efficiency of shared-memory architectures. In order to improve locality, we need a mechanism to analyze the memory access behavior of parallel applications. In this paper, we presented SAMMU, a mechanism that is implemented as an extension of the memory management unit of the processors that analyzes the memory access behavior during run time. Using the information analyzed by SAMMU, the operating system can perform the sharing-aware mapping of threads and data during the execution of the applications without any prior knowledge of their behavior. It detects the memory access pattern completely in hardware, directly considering the memory accesses. SAMMU requires only minimal changes to the hardware and operating system, adding less than 0.05% of circuit area in modern processors.

We performed experiments with the NAS OpenMP benchmarks and PARSEC on a full system simulator and two real machines. The results showed performance improvements of up to 35.7% (10.0% on average) and improved processor energy efficiency by up to 11.9% (4.1% on average). The L3 cache MPKI was reduced by an average of 30.6% in the real machines. The interchip interconnection traffic was reduced by an average of 39.0% and 66.8% in the real machines and simulator, respectively. The performance overhead represented only 0.27% and 0.29% on average on the hardware and software level, respectively. We also demonstrated that SAMMU is able to handle a wide range of architectures by varying the simulation parameters. Compared to previous work, SAMMU presented the best performance improvements for most applications.

For the future, we plan to evaluate SAMMU using parallel applications with several processes that do not necessarily share the same virtual address space.

Acknowledgment

This research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement n.º 689772. It was also supported by the Coordination for the Improvement of Higher Education Personnel (CAPES), the National Council for Scientific and Technological Development (CNPq), and Intel.

References

[1] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, apr 2009.

[2] Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed A. Badawy, and Tarek El-Ghazawi. Exploiting Hierarchical Locality in Deep Parallel Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–25, 2016.

[3] Krste Asanovic, Bryan Christopher Catanzaro, David A. Patterson, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, 2006.

[4] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 319–330, 2010.

[5] Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65, apr 2009.

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

[7] S. Barghi and M. Karsten. Work-stealing, locality-aware actor scheduling. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 484–494, May 2018.

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

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

[10] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

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

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

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

[14] Jonathan Corbet. Toward better NUMA scheduling, 2012.

[15] P. W. Coteus, J. U. Knickerbocker, C. H. Lam, and Y. A. Vlasov. Technologies for exascale systems. *IBM Journal of Research and Development*, 55(5):14:1–14:12, sep 2011.

[16] Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, and Philippe O. A. Navaux. LAPT: A Locality-Aware Page Table for thread and data mapping. *Parallel Computing*, 54(May):59–71, 2016.

[17] Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. A Sharing-Aware Memory Management Unit for Online Mapping in Multi-core Architectures. In *Euro-Par Parallel Processing*, pages 659–671, 2016.

[18] Eduardo H. M. Cruz, Matthias Diener, Laercio L. Pilla, and Philippe O. A. Navaux. Hardware-Assisted Thread and Data Mapping in Hierarchical Multicore Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(3):1–25, 2016.

[19] Blas Cuesta, Alberto Ros, Maria E. Gomez, Antonio Robles, and Jose Duato. Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Non-Coherent Memory Blocks. *IEEE Transactions on Computers*, 62(3):482–495, 2013.

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

[21] Matthias Diener, Eduardo H. M. Cruz, and Philippe O. A. Navaux. Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 9–16, 2015.

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

[23] Matthias Diener, Eduardo H. M. Cruz, Laércio L. Pilla, Fabrice Dupros, and Philippe O. A. Navaux. Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping. *Performance Evaluation*, 88-89(June):18–36, 2015.

[24] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 508–519, 2012.

[25] Intel. 2nd Generation Intel Core Processor Family. Technical Report September, 2012.

[26] Intel. Intel Performance Counter Monitor - A better way to measure CPU utilization, 2012.

[27] JEDEC. DDR3 SDRAM Standard, 2012.

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

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

[30] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *Annual Technical Conference (USENIX ATC)*, pages 277–289, Santa Clara, CA, July 2015. USENIX Association.

[31] Tan Li, Yufei Ren, Dantong Yu, and Shudong Jin. Analysis of numa effects in modern multicore systems for the design of high-performance data transfer applications. *Future Generation Computer Systems*, 74:41 – 50, 2017.

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

[33] Hao Luo, Pengcheng Li, and Chen Ding. Thread data sharing in cache: Theory and measurement. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 103–115, New York, NY, USA, 2017. ACM.

[34] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[35] Zoltan Majo and Thomas R. Gross. A library for portable and composable data locality optimizations for numa systems. *ACM Transactions on Parallel Computing (TOPC)*, 3(4):20:1–20:32, March 2017.

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

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

[38] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why On-Chip Cache Coherence is Here to Stay. *Communications of the ACM*, 55(7):78, jul 2012.

[39] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[40] Takeshi Ogasawara. NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA*, pages 377–390, 2009.

[41] François Pellegrini. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Scalable High-Performance Computing Conference (SHPCC)*, pages 486–493, 1994.

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

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

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

[45] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, Norman P Jouppi, and Palo Alto. Cacti 5.1. Technical report, 2008.

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

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

[48] François Trahay, Manuel Selva, Lionel Morel, and Kevin Marquet. Numamma: Numa memory analyzer. In *International Conference on Parallel Processing (ICPP)*, pages 19:1–19:10, New York, NY, USA, 2018. ACM.

[49] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás. Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(10):3007–3020, Oct 2017.

[50] Wei Wang, Tanima Dey, Jason Mars, Lingjia Tang, Jack W Davidson, and Mary Lou Soffa. Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012.

[51] Jidong Zhai, Tianwei Sheng, and Jiangzhou He. Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 22(11):1862–1870, 2011.

A Analysis of the Equation that Updates the Access Threshold

In the **Case 2** explained in Sec. 2.2, we defined three properties for the function $f(\cdot)$ that subtracts a value from AT:

$$AT_{new} = AT - f(AC) \tag{8}$$
$$f(0) = 0 \tag{9}$$
$$f(AT) = 0 \tag{10}$$
$$f(AC) = k \tag{11}$$

Where we consider AT as a constant, $0 < AC < AT$ and $0 < k < AT$.

No linear equation can provide the properties required by function $f(\cdot)$. We then chose to use a quadratic equation. We could use other polynomial equations, but since we need to implement this in hardware, we want to use the simplest equation possible. The quadratic equation is concave down. Therefore, to find function $f(\cdot)$:

$$f(AC) = (-1) \times (AC) \times (AC - AT) \times c \tag{12}$$
$$= AC \times AT \times c - AC^2 \times c \tag{13}$$

The roots of the equation are 0 and AT. We multiply by -1 so the equation is concave down. We also multiply by a constant c to help us control the concavity of the function to keep $f(AC) < AT$. To find the possible values of c, we need to differentiate $f(\cdot)$ to find its critical point.

$$f'(AC) = AT \times c - 2 \times c \times AC \tag{14}$$

To find the critical point, we equal the derivative to 0.

$$f'(AC) = 0 \tag{15}$$
$$AT \times c - 2 \times c \times AC = 0 \tag{16}$$
$$2 \times c \times AC = AT \times c \tag{17}$$
$$AC = \frac{AT \times c}{2 \times c} \tag{18}$$
$$AC = \frac{AT}{2} \tag{19}$$

Now, we know that, regardless the value of c and AT, the maximum value of $f(AC)$ happens when $AC = {}^{AT}/_{2}$. That is, when the Access Counter is half of the current Access Threshold. We

can use this value of AC in Eq. 13 to get the maximum value that $f(AC)$ can subtract from AT in Eq. 8.

$$f\left(\frac{AT}{2}\right) = \frac{AT}{2} \times AT \times c - \left(\frac{AT}{2}\right)^2 \times c \tag{20}$$

$$= \frac{AT^2}{2} \times c - \frac{AT^2}{4} \times c \tag{21}$$

$$= \frac{2}{4} \times AT^2 \times c - \frac{AT^2}{4} \times c \tag{22}$$

$$= \frac{c \times AT^2(2-1)}{4} \tag{23}$$

$$= \frac{c \times AT^2}{4} \tag{24}$$

Eq. 24 specifies the maximum value of $f(AC)$. We need now to find the possible values of c such that $f(AC) < AT$.

$$\frac{c \times AT^2}{4} < AT \tag{25}$$

$$c < \frac{4 \times AT}{AT^2} \tag{26}$$

$$c < \frac{4}{AT} \tag{27}$$

Since we want to reduce the impact of threads that access the page few times as much as possible, as well as keep the hardware implementation as simple as we can, we chose to use $c = 1/AT$. Therefore, the equation used to update the Access Threshold in **Case 2** is given by Eq. 32, found by substituting the value of c in Eq. 8 and Eq. 13 by $c = 1/AT$.

$$AT_{new} = AT - f(AC) \tag{28}$$

$$= AT - \left(AC \times AT \times c - AC^2 \times c\right) \tag{29}$$

$$= AT - \left(AC \times AT \times \frac{1}{AT} - AC^2 \times \frac{1}{AT}\right) \tag{30}$$

$$= AT - \frac{AC \times (AT - AC)}{AT} \tag{31}$$

$$AT_{new} = AT - \frac{AT - AC}{\left(\frac{AT}{AC}\right)}, \quad AC < AT \tag{32}$$

Eq. 32 guarantees that AT will never be decreased by more than 25% at each update. The behavior of the equations that control the updates of the Access Threshold are illustrated in Fig. 12 and Fig. 13. In the following equations, we prove that AT will never be decreased by more than 25% at each update. For that, the value of AT_{new}/AT must be greater than or equal to 0.75.

$$\frac{AT_{new}}{AT} = \frac{AT - \frac{AT-AC}{\left(\frac{AT}{AC}\right)}}{AT} \tag{33}$$

$$= 1 - \frac{AT - AC}{\left(\frac{AT^2}{AC}\right)} \tag{34}$$

$$= 1 - \frac{AC \times AT - AC^2}{AT^2} \tag{35}$$

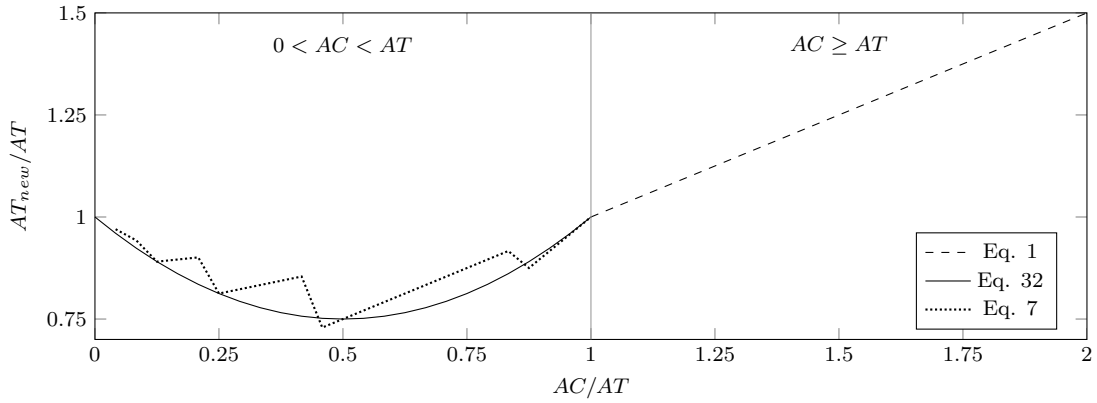
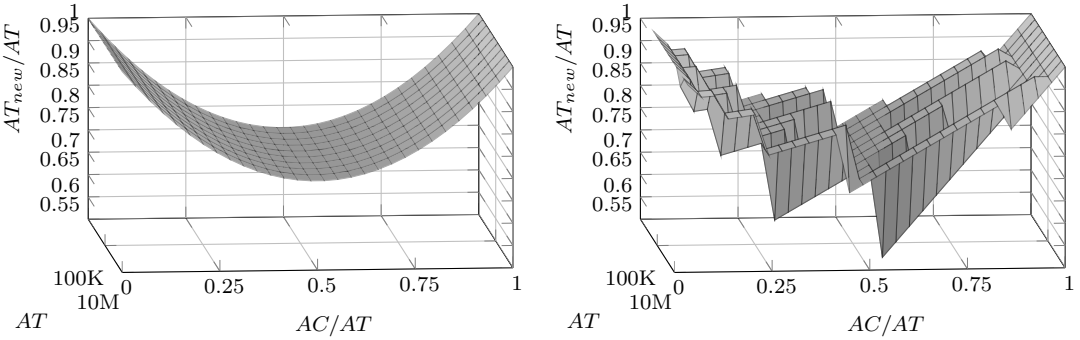


Figure 12: Value of AT_{new} relative to AT. For Eq. 7, we consider AT as $10M$.



(a) Equation 32

(b) Equation 7

Figure 13: Behavior of Eq. 32 and Eq. 7 when varying AT from $100K$ to $10M$.

.

At this point, we can make a change of variables, considering $AC = \alpha AT$. Since in **Case 2** $AC < AT$, α must be $0 \leq \alpha < 1$.

$$\frac{AT_{new}}{AT} = 1 - \frac{\alpha AT \times AT - (\alpha AT)^2}{AT^2} \tag{36}$$

$$= 1 - \frac{\alpha AT^2 - \alpha^2 AT^2}{AT^2} \tag{37}$$

$$= 1 - \left(\alpha - \alpha^2\right) \tag{38}$$

$$= \alpha^2 - \alpha + 1 \tag{39}$$

Therefore, we know that ${}^{AT_{new}}/_{AT}$ is a quadratic equation and is concave up, which means it has a global minimum value. To get the minimum value of AT_{new} relative to AT, we need first to differentiate Eq. 39.

$$\frac{d(\alpha^2 - \alpha + 1)}{d\alpha} = 2\alpha - 1 \tag{40}$$

Now, we need to find the value of α when its derivative is equal to 0.

$$2\alpha - 1 = 0 \tag{41}$$

$$\alpha = \frac{1}{2} \tag{42}$$

Then, we know that the minimum value of AT_{new} relative to AT happens when $\alpha = 1/2$. In other words, when the value of the Access Counter AC is half of the Access Threshold AT. To find the minimum value, we need to insert the value of $\alpha = 1/2$ in Eq. 39.

$$\frac{AT_{new}}{AT} = \alpha^2 - \alpha + 1 \tag{43}$$

$$= \left(\frac{1}{2}\right)^2 - \frac{1}{2} + 1 \tag{44}$$

$$= \frac{3}{4} = 0.75 \tag{45}$$

With this, we demonstrated that, for **Case 2**, the minimum value of AT_{new} is $0.75 \times AT$, a 25% decrease, and happens when AC is half of AT.