A Sharing-Aware Memory Management Unit for Online Mapping in Multi-Core Architectures

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

[1] Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Brazil
{ehmcruz,mdiener,navaux}@inf.ufrgs.br
[2] Department of Informatics and Statistics, Federal University of Santa Catarina (UFSC), Brazil laercio.pilla@ufsc.br

Abstract. In modern shared-memory architectures, it is important to map threads and data in a way that increases the locality of their memory accesses, thereby improving performance and energy efficiency. Threads that access shared data should be mapped close to each other in the memory hierarchy, while the data they access should be mapped to their NUMA node, which is called sharing-aware mapping. In this paper, we propose SAMMU, which adds sharing-awareness to the memory management unit in current architectures. SAMMU analyzes the memory access behavior in hardware and provides information to the operating system so it can perform an online mapping of threads and data. In the evaluation with a wide range of parallel applications, performance was improved by up to 35.7% (13.1% on average).

1 Introduction

As parallel applications need to access shared data, the memory hierarchy presents challenges for mapping threads to cores, and data to NUMA nodes [24]. 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 that the threads that access it are executing on [22]. In this way, the *locality* of the memory accesses is improved, which leads to an increase of performance and energy efficiency. This type of thread and data mapping is called *sharing-aware* mapping. For optimal performance improvements, data and thread mapping should be performed together [23]. For the thread mapping, knowledge about how data is shared between the threads is necessary. Data mapping additionally requires information about the memory pages that are accessed by each thread.

Sharing-aware thread and data mapping improve performance and energy efficiency of parallel applications by optimizing memory accesses [11]. Improvements happen for three main reasons. First, cache misses are reduced by decreasing the number of invalidations that happen when write operations are performed on shared data [19]. For read operations, the effective cache size is increased by reducing the replication of cache lines on multiple caches [6]. Second, the locality

of memory accesses is increased by mapping data to the NUMA node where it is most accessed. 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.

In this paper, we propose a *Sharing-Aware Memory Management Unit* (SAMMU), which uses the virtual memory implementation to detect the memory access pattern during the execution of a parallel application. SAMMU modifies the memory management unit to analyze the memory access behavior, which is used to perform online thread and data mapping. To the best of our knowledge, SAMMU is the first mechanism that detects the memory access pattern for thread and data mapping completely on the hardware level, considering many more memory accesses than related work to achieve a higher accuracy. It requires no changes to the application or its runtime system, and needs no previous information about application behavior.

2 Related Work

Traditional data mapping strategies, such as *first-touch* and *next-touch* [15], 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 [7] 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. However, it does not detect sharing patterns between threads.

Marathe et al. [18] 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 [3]. A similar technique is used in Marathe and Mueller [17] to perform data mapping dynamically. They enable the profiling mechanism just during the beginning of each application due to the high overhead, losing the opportunity to handle changes in rest of the execution. 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. [1] 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. Cruz et al. [9] detect the pattern by monitoring the invalidation messages of cache coherence protocols. Only thread mapping was performed, which does not improve the locality of memory accesses in NUMA architectures.

The kMAF affinity framework is proposed in [11]. It performs both thread and data mapping and gather information from page faults. Carrefour [10] is a similar mechanism that uses sampling to detect page usage. Due to its over-

head, the authors restrict the mechanism to 30,000 pages, which limits its use to applications with a low memory usage. These mechanisms generate mapping information based on a very small number of samples compared to SAMMU, as all memory accesses are handled by the MMU. Some techniques such as Forest-GOMP [4] require annotations in the source code and depend on specific parallelization libraries. Similarly, Ogasawara [20] proposes a data mapping method that is limited to object oriented languages.

The usage of the instructions per cycle (IPC) metric to guide thread mapping is evaluated in Autopin [14]. Autopin itself does not detect the sharing pattern, it only verifies the IPC of several mappings fed to it and executes the application with the thread mapping that presented the highest IPC. The BlackBox scheduler [21], 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.

3 SAMMU: A Sharing-Aware Memory Management Unit

Computer systems that support virtual memory use a memory management unit (MMU) to translate virtual to physical addresses. 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. A special cache memory, the Translation Lookaside Buffer (TLB), is used to speed up the address translation. 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.

The operation of the MMU is extended 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 saturating *access counters* (AC), one per TLB entry, in a table that we call *TLB access table*, which is stored in the MMU.
2. On every TLB eviction or when an access counter saturates, SAMMU analyzes statistics about the page and stores them in the main memory in two separate structures. The first structure is the *sharing matrix (SM)*, which estimates the amount of sharing between the threads. The second structure is the

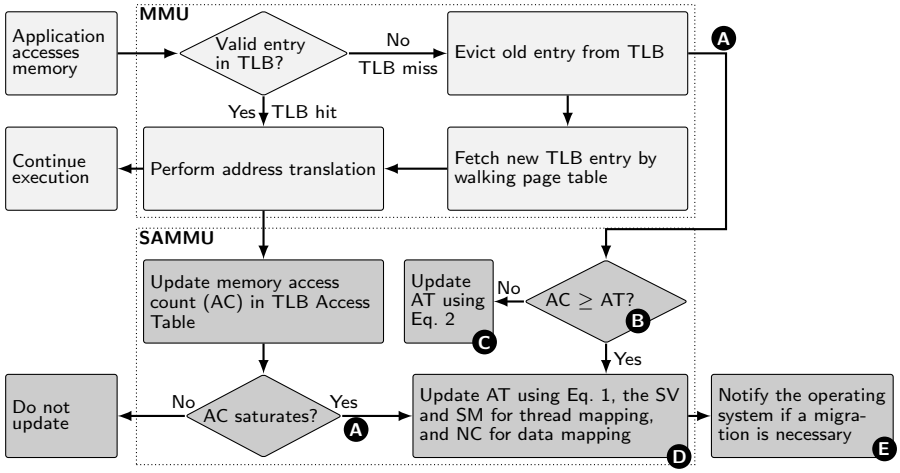


Fig. 1: Overview of the MMU and SAMMU.

page history table, which contains one entry per physical page in the system with information about the threads and NUMA nodes that accessed them, and is indexed by the physical page address. Each entry of the page history table has three fields: (1) *access threshold* (AT), which defines the minimum number of memory accesses required to modify the statistics; (2) *sharers vector* (SV), which contains the ID of the last threads that accessed the page; (3) *NUMA counters* (NC), which estimate the number of accesses from each NUMA node.

3.1 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. We count the number of accesses to the TLB entry of a page by adding a saturating *access counter* (AC) to the TLB access table. When AC saturates or a TLB entry gets evicted, SAMMU collects the information and updates the page history table entry of the page. To filter out threads that perform only few accesses to a page, we use an *access threshold* (AT) in the page history table. AT specifies the minimum number of memory accesses required to update the mapping-related information for a page. A number of memory accesses smaller than AT means that a thread does not use a page enough to influence its mapping. SAMMU updates the mapping-related statistics of a page only when its AC 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 (Fig. 1-B,D).

A detailed example of the operation of SAMMU can be found in Fig. 1. The initial value of AT is 0. Access thresholds are kept per page because the number of memory accesses can vary from page to page. SAMMU automatically adjusts the access threshold of a given page, separating this procedure into two cases (Fig. 1-B):

Case 1 (AC saturates or $AC \geq AT$): When AC saturates or, during a TLB eviction, AC is greater than or equal to its access threshold (AT) (Fig. 1-Ⓑ,Ⓓ), we need to increase AT to reduce the influence of threads that perform few accesses to the page. Therefore, AT is updated with the average value of AC and AT, as illustrated in Eq. 1. Also, the mapping statistics are updated, as explained in Sections 3.2 and 3.3. 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.

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

Case 2 ($AC < AT$): In the second case, when the number of memory accesses registered by AC during a TLB eviction is lower than the access threshold (Fig. 1-Ⓑ,Ⓒ), we update AT in such a way that NUMA nodes with a small number of accesses to the page have a lower influence on the threshold. For that, we use Eq. 2, which guarantees that AT will never be decreased by more than 25% at each update. In this case, mapping statistics are not updated.

$$AT_{new} \leftarrow AT - \frac{AT - AC}{AT/AC}, \quad AC < AT \tag{2}$$

3.2 Detecting the Sharing Pattern for Thread Mapping

To detect the sharing pattern, SAMMU identifies the last threads that accessed a memory page. To obtain that information, SAMMU adds a small *sharers vector* (SV) to each page history table entry. Each SV stores the IDs of the last threads to access its page. This has the advantage of maintaining temporal locality when detecting which threads share each page. Old entries will be overwritten and not considered as sharers. 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. In the TLB access table, SAMMU stores the ID of the thread that accessed each TLB entry. Control registers containing the memory address and dimensions of the sharing matrix, and the ID of the thread being executed must be added to the architecture and updated by the operating system.

When SAMMU is triggered for a certain page by thread T (Fig. 1-Ⓐ), 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. 1-Ⓑ), SAMMU then increments the sharing matrix in row T, for all the columns that correspond to an entry in the SV (Fig. 1-Ⓓ): $SM[T][SV[i]] \leftarrow SM[T][SV[i]] + 1$.

Each line of SM is accessed by its corresponding thread only, minimizing the impact of 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.

3.3 Detecting the Page Usage Pattern for Data Mapping

To identify where a memory page should be mapped, SAMMU requires the addition of a vector to each page history table entry. The vector, which we call

Table 1: Configuration of the experiments.

System	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, $V_{add} = 2$, $NT = 10$
Pin	L1 TLB	64 entries, 4-way, shared between 2 SMT-cores
	L2 TLB	512 entries, 4-way, shared between 2 SMT-cores
Xeon	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

NUMA counters (NC), has N elements for a system with N NUMA nodes. NC employs saturating counters to count a relative number of accesses from different NUMA nodes. The initial value of each NC is 0.

When a TLB entry from a core in NUMA node n is selected for eviction or its AC reaches its maximum value (Fig. 1-Ⓐ), 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. 1-Ⓑ), SAMMU increments the NUMA counter of node n, and decrements all other NUMA counters (Fig. 1-Ⓓ). Since the NUMA counters are saturated, they do not overflow nor underflow.

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 the NUMA counters of n and m is greater than or equal to a global value *NUMA threshold* (NT) (Fig. 1-Ⓔ): $NC[n] - NC[m] \geq NT$. If that is the case, SAMMU notifies the operating system of the page and its destination node n. The NUMA threshold may be configured by a control register. The operating system then chooses how it will handle the migration of the page. The higher the NUMA threshold NT, the lower the number of page migrations.

4 Experiments and Results

In this section, we present the experiments we performed with SAMMU. We describe the methodology and then evaluate the performance and overhead.

4.1 Methodology

The parameters of our experiments are summarized in Table 1. The experiments were performed using a real machine. The machine consists of 4 NUMA nodes with one 8-core, 2-SMT Intel Xeon X7550 processor per node, with a total of 64 virtual cores. It is running version 3.8 of the Linux kernel. Information about the hardware topology is gathered using Hwloc [5]. To generate the thread mappings, we used the EagerMap [8] mapping algorithm, which receives the sharing matrix and a graph representing the memory hierarchy (from Hwloc) as input, and it outputs which core will execute each thread.

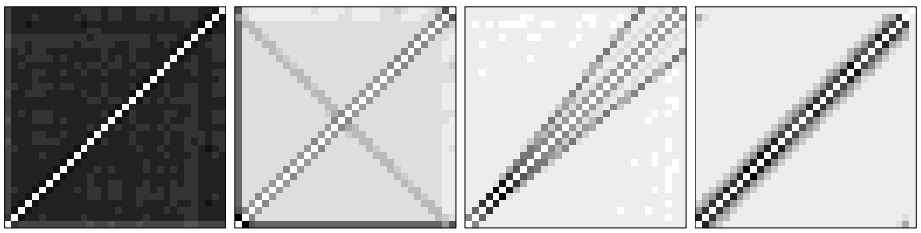
As workloads, we used the OpenMP implementation of the NAS parallel benchmarks (NPB) [13], v3.3.1. All experiments were executed 30 times. We show average values as well as a 95% confidence interval calculated with Student's t-distribution. Results are normalized to the operating system original mapping. We configured the benchmarks to run with one thread per virtual core. Input sizes were chosen to provide similar total execution times and feasible simulation time. Benchmarks BT, LU, SP and UA were executed using input size *A*. Benchmarks CG, EP, FT, IS and MG were executed using input size *B*.

Since SAMMU is an extension to the current MMU hardware, we simulate its behavior with the Pin [2] 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 in real machines, the mapping information generated in Pin is fed into the mapping mechanism in runtime. This is possible because the access pattern of the applications we evaluated and their memory addresses remain the same across different executions, since their memory is statically allocated. Besides performance, we measured L3 cache misses per thousand instructions (MPKI) and QPI interchip interconnection traffic using the Intel PCM tool [12].

4.2 Performance Results

The sharing patterns of a subset of our workloads are illustrated in Fig. 2. The results of execution time can be found in Fig. 3, L3 cache misses per thousand instructions (MPKI) in Fig. 4a, and interchip traffic in Fig. 4b. Lower values are better. In these figures, we also show the average improvements, calculated using the geometric mean function. In this section, we focus on the SAMMU results. The next section presents a comparison to other mapping techniques that are shown in the figures.

In applications whose pages are shared within a small subgroup of threads, mapping presents a high potential for performance improvement. For instance, in SP, most sharing happens between neighboring threads, which is very common in parallel applications that use domain decomposition. In LU and MG, the sharing



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

Fig. 2: 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.

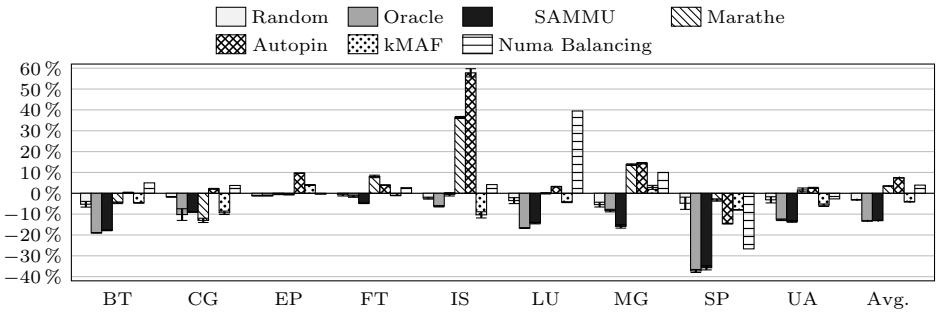
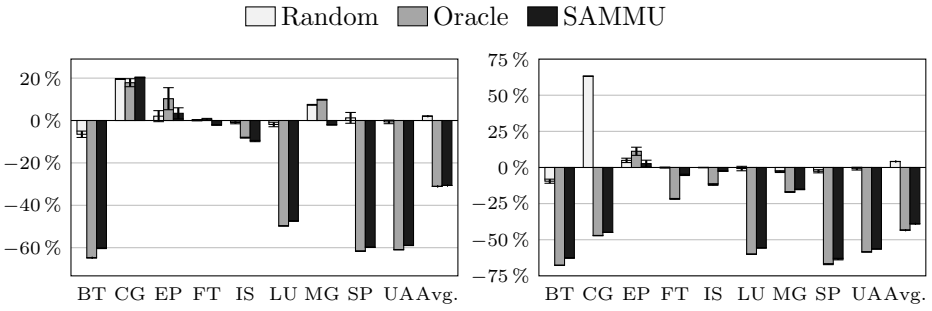


Fig. 3: Execution time normalized to the operating system.



(a) L3 cache MPKI.

(b) Interchip interconnection traffic.

Fig. 4: Performance results, normalized to the operating system.

between more distant threads is more evident than in the other applications. The threads of these applications are able to benefit from the shared cache memories and faster interconnection when mapped nearby in the memory hierarchy, as well as accessing shared pages from their local NUMA node. In general, the effect is a reduction of cache misses and interchip traffic, observed in LU and SP. SP presented the highest improvements, with an execution time reduction of 35.7%.

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, the reduction of interchip traffic is higher than the reduction of cache misses. The reason is that the better fitting thread mapping results in a placement of threads that share data on the same NUMA node, thus reducing interchip traffic. 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 suitable for thread mapping. One example of this type of application is CG. The sharing pattern of CG is illustrated in Fig. 2a, where we can observe that each pair of threads has a similar amount of sharing. Therefore, no thread mapping is able to improve the

usage of cache memories. This is the reason that SAMMU does not decrease the number of cache misses in CG. However, due to the data mapping, SAMMU improved the memory access locality in CG such that the amount of interchip traffic was decreased by 44.9%, leading to a performance improvement of 9.0%.

In some applications, no performance improvements are expected, either by thread or data mapping. For instance, EP is a CPU-bound application [13] with almost no data sharing among its threads. Due to this, there is no thread mapping that is able to optimize the memory accesses. Regarding data mapping, since it is a CPU bound application, the memory accesses have very little influence in the performance of EP.

The number of cache misses and the traffic in the interconnections were reduced by SAMMU significantly. L3 MPKI was reduced by an average of 30.6%. Interchip traffic was reduced by an average of 39.0%. The execution time was reduced by an average of 13.1%. This smaller reduction 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 several threads.

4.3 Comparison to Related Work

We compare SAMMU to the following techniques: Random and Oracle mappings, the Marathe [17] data mapping mechanism, Autopin [14], the kMAF affinity framework [11] and NUMA Balancing [7]. 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 [3]. Autopin was executed with 5 mappings: the Oracle mapping and 4 random mappings. We implemented Marathe using a long latency load profile [17] in Pin and fed the information during the execution of the application.

Execution time results of the related work are also shown in Fig. 3. In CG, Marathe presented slightly better results than SAMMU. This happens because, as previously explained, CG is only affected by data mapping, such that SAMMU introduces thread migrations during execution that increase the overhead. Unnecessary thread 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 kMAF are lower than SAMMU for most of the benchmarks. Due to its sampling mechanism, kMAF needs more time to detect the memory access behavior, losing opportunities for improvements. The only application in which NUMA Balancing performed well was SP.

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 mechanisms that have access to more accurate information about the memory accesses can provide better performance improvements. SAMMU presented results similar to the Oracle mapping, demonstrating its effectiveness. In most cases, it performed significantly better than the random mapping. This shows that the gains 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.

4.4 Overhead of SAMMU

SAMMU causes an overhead on the execution of the parallel application on the hardware and software levels. In the hardware level, the additional hardware of SAMMU is not in the critical path, since 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 its structures stored in the main memory. To calculate this overhead, we measured the average memory access latency in the Simics full system simulator [16], and multiplied it by the number of additional memory accesses introduced by SAMMU. On the software level, the operating system introduces overhead when calculating the thread mapping, and when migrating threads and pages.

The performance overhead caused by the hardware was 0.41%, due to the introduction of 1.43% additional memory transactions, on average. The overhead in the software level was 0.29%, on average. These results show that SAMMU has only a small performance overhead. Regarding storage overhead, each entry of the page history table would require 8 Bytes, with a total space overhead of 0.2% relative to the total main memory. The sharing matrix would require 256 KByte, each of its elements with 4 Bytes. We estimate the additional hardware required by SAMMU by counting the amount of transistors required in the implementation. SAMMU would require 143,000 transistors per core, which results in an increase in transistors of less than 0.05% in a modern processor.

5 Conclusions and Future Work

In this paper, we presented SAMMU, an extension of the memory management unit to improve locality of memory accesses. SAMMU analyzes the memory accesses of multithreaded applications during execution, such that the operating

system can perform a sharing-aware online mapping of threads to cores and data to NUMA nodes. In contrast to previous proposals, it detects the memory access pattern completely in hardware, considering most memory accesses and achieving a higher accuracy. It is independent of the application and its runtime system, and requires no source code modification or previous information about the behavior of the application.

Experiments with the NAS OpenMP benchmarks showed performance improvements of up to 35.7% (13.1% on average). L3 cache MPKI and interchip interconnection traffic were reduced by an average of 30.6% and 39.0%, respectively. Compared to previous work, SAMMU presented the best performance improvements for most applications.

For the future, we will evaluate SAMMU using parallel applications with several processes that do not necessarily share the same virtual address space, as well as running multiple applications simultaneously.

Acknowledgment

This research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement n.º 689772. This work was also supported by the STIC-AmSud/CAPES scientific cooperation program under the EnergySFE research project grant 99999.007556/2015-02. Additional funding was provided by CNPq and Capes.

References

1. Azimi, R., Tam, D.K., Soares, L., Stumm, M.: Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. ACM SIGOPS Operating Systems Review 43(2), 56–65 (Apr 2009)
2. Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.K., Lyons, G., Patil, H., Tal, A.: Analyzing Parallel Programs with Pin. IEEE Computer 43(3) (2010)
3. Barrow-Williams, N., Fensch, C., Moore, S.: A Communication Characterisation of Splash-2 and Parsec. In: IEEE International Symposium on Workload Characterization (IISWC) (2009)
4. Broquedis, F., Aumage, O., Goglin, B., Thibault, S., Wacrenier, P.A., Namyst, R.: Structuring the execution of OpenMP applications for multicore architectures. In: IEEE International Parallel & Distributed Processing Symposium (IPDPS) (2010)
5. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In: Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 180–186 (2010)
6. Chishti, Z., Powell, M.D., Vijaykumar, T.N.: Optimizing Replication, Communication, and Capacity Allocation in CMPs. ACM SIGARCH Computer Architecture News 33(2) (2005)
7. Corbet, J.: Toward better NUMA scheduling (2012), `http://lwn.net/Articles/486858/`

8. Cruz, E.H.M., Diener, M., Pilla, L.L., Navaux, P.O.A.: An Efficient Algorithm for Communication-Based Task Mapping. In: International Conference on Parallel, Distributed, and Network-Based Processing (PDP). pp. 207–214 (2015)
9. Cruz, E.H.M., Diener, M., Alves, M.A.Z., Navaux, P.O.A.: Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. Journal of Parallel and Distributed Computing 74(3), 2215–2228 (Mar 2014)
10. Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., Roth, M.: Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2013)
11. Diener, M., Cruz, E.H.M., Navaux, P.O.A., Busse, A., Heiß, H.U.: kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In: Interntional Conference on Parallel Architectures and Compilation Techniques (PACT) (2014)
12. Intel: Intel Performance Counter Monitor - A better way to measure CPU utilization (2012), `http://www.intel.com/software/pcm`
13. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS Parallel Benchmarks and Its Performance (1999)
14. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. High Performance Embedded Architectures and Compilers 3(4) (2008)
15. Löf, H., Holmgren, S.: affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In: International Conference on Supercomputing (2005)
16. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. IEEE Computer 35(2) (2002)
17. Marathe, J., Mueller, F.: Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (2006)
18. Marathe, J., Thakkar, V., Mueller, F.: Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces. Journal of Parallel and Distributed Computing 70(12) (2010)
19. Martin, M.M.K., Hill, M.D., Sorin, D.J.: Why On-Chip Cache Coherence is Here to Stay. Communications of the ACM 55(7), 78 (Jul 2012)
20. Ogasawara, T.: NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. ACM SIGPLAN Notices 44(10), 377–389 (Oct 2009)
21. Radojković, P., Cakarević, V., Verdú, J., Pajuelo, A., Cazorla, F.J., Nemirovsky, M., Valero, M.: Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors. IEEE Transactions on Parallel and Distributed Systems (TPDS) 24(12), 2513–2525 (2013)
22. Ribeiro, C.P., Mehaut, J.F., Carissimi, A., Castro, M., Fernandes, L.G.: Memory Affinity for Hierarchical Shared Memory Multiprocessors. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (2009)
23. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and Thread Affinity in OpenMP Programs. In: Workshop on Memory Access on Future Processors: A Solved Problem? (MAW) (2008)
24. Wang, W., Dey, T., Mars, J., Tang, L., Davidson, J.W., Soffa, M.L.: 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)