2010 22nd International Symposium on Computer Architecture and High Performance Computing

Feedback-driven Restructuring of Multi-threaded Applications for NUCA Cache Performance in CMPs

Sandro Bartolini
Dipartimento di Ingegneria dell'Informazione
Università degli studi di Siena, Siena, Italy
bartolini@dii.unisi.it

Pierfrancesco Foglia, Marco Solinas, Cosimo Antonio Prete
Dipartimento di Ingegneria dell'Informazione
Università di Pisa, Pisa, Italy
foglia,marco.solinas,prete@iet.unipi.it

***Abstract*—This paper addresses feedback-directed restructuring techniques tuned to Non Uniform Cache Architectures (NUCA) in CMPs running multi-threaded applications. Access time to NUCA caches depends on the location of the referred block, so the locality and cache mapping of the application influence the overall performance. We show techniques for altering the distribution of applications into the cache space as to achieve improved average memory access time. In CMPs running multi-threaded applications, the aggregated accesses (and locality) of the processors form the actual cache load and pose specific issues. We consider a number of Splash-2 and Parsec benchmarks on an 8 processor system and we show that a relatively simple remapping algorithm is able to improve the average Static-NUCA (SNUCA) cache access time by 5.5% and allows an SNUCA cache to surpass the performance of a more complex dynamic-NUCA (DNUCA) for most benchmarks.**

Then, we present a more sophisticated remapping algorithm, relying on cache geometry information and on the access distribution statistics from individual processors, that reduces the average cache access time by 10.2% and is very stable across all benchmarks.

***Keywords*-Feedback-directed optimizations, compiler optimizations, NUCA caches, CMPs, multi-threaded applications.**

I. Introduction

NUCA caches are developing as a promising design strategy to allow scaling-up the size of the on-chip cache, while coping with the emerging wire-delay effects of deep sub-micron technology [1]. The pressure on increasing last-level-cache size is kept high by the memory-wall problem and by the increasing aggregated demand of memory bandwidth from the numerous cores of new chip-multiprocessor architectures [2].

Non Uniform Cache Architectures (NUCA) propose a *tiled* last-level cache (LLC) design which are scalable with cache size, with the number of on-chip cores and with wire delay issues. As shown in figure 1, the cache is divided into multiple banks connected through an ad-hoc network-on-chip (NoC). NUCA caches exhibit a different hit access time depending on the positioning of the target block: banks closer to the processor are accessed faster than the ones farther away. Therefore, application performance on a NUCA cache is highly dependant on how its accesses are distributed on the cache banks (see figure 1c).

Multi-core architectures and multi-threaded programs increase the complexity of analyzing the application behavior and, therefore, the complexity of defining application-specific optimization strategies.

We have analyzed the behavior of some Parsec [3] and Splash-2 [4] parallel benchmarks aiming to define remapping approaches for improving the number of accesses that are served with low hit time, i.e., from banks close to the processors. However, this has to be done with extreme care because an excessive accumulation of accesses in the same cache region can increase misses, which are very time-costly to serve, and can easily overwhelm the hit time benefits.

In this paper we will explore two feedback-driven remapping strategies for improving the average NUCA+memory average access time and we evaluate their performance on multi-core architectures running multi-threaded benchmarks. In particular, section II introduces the main NUCA features and section III describes the proposed remapping strategies (*simple* and *advanced*). Results are discussed in section IV, while related works are addressed in section V. Finally section VI concludes the paper.

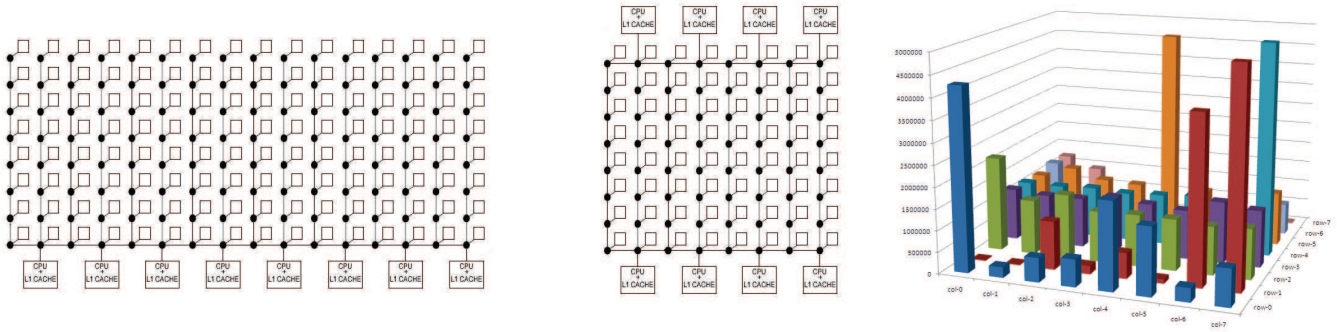
II. Nuca Caches

NUCA caches were proposed [5] for addressing wire-delay problems and to ease the size and organization scalability of large, typically, last-level caches.

In this paper we will mainly focus on static-NUCA caches (SNUCA) [5], in which a memory block is univocally associated to a specific bank according to its memory address. Part of the address bits are used to route the search in the NoC towards the bank that might host the requested information.

We will compare also against a dynamic version of the NUCA design (DNUCA) [5], in which each memory block is associated with a *bankset* (e.g., a vertical bank column in figure 1) according to its address. The bankset works as a set in associative caches and, upon a hit, a

1550-6533/10 $26.00 © 2010 IEEE
DOI 10.1109/SBAC-PAD.2010.20



(a) 8p (b) 4p4 (c) bank access distribution (*streamcluster*)

Figure 1: Examples of NUCA configurations. (a) An 8x16 banks cache with eight processor on the same side (*8p*), and (b) an 8x8 bank cache with four processors on two opposite sides (*4p4*). (c) Example of bank access distribution of *streamcluster* application on an *8p* SNUCA 8x8 bank array.

blocks can migrate to the adjacent bank closer to the requesting processor. In this way, the next access to it will have a smaller hit time. The bank access policy in DNUCA caches is sequential into the bankset because the requested block can be everywhere in it.

The flexible and scalable nature of NUCAs allows a number of processor-to-cache connections to be easily implementable. In this paper we consider two schemes featuring eight processors, and whose behavior has been investigated in [6]: one with all processors connected to one of its sides as shown in figure 1a (*8p*), and another with four processors at the two opposite sides of the bank array as in figure 1b (*4p4*).

The access time to a NUCA bank depends on the number of cache banks to be traversed (NoC hops) for reaching the target one. We aim at investigating possible remapping techniques for exploiting the combination of application locality and non-uniform cache access time for improving the average NUCA access time.

III. Program restructuring strategies

We present two profile-guided strategies for improving and exploiting locality into SNUCA caches, leveraging on the optimizer ability of analyzing profile information and identifying favorable mappings of the parallel application behavior on the non-uniform cache space. These optimizations operate at compile/link time and insert the remapping information (tables) into specific sections of the executable. Such tables associate the address of performance-critical virtual pages to the desired cache positioning(s). The Operating System will then use them to position the pages into the physical memory, when needed.

Our profiling, cache mapping and management strategies operate at 4-kB granularity. This allows a standard Operating System and virtual-memory management service to manage the cache remapping with minimal modifications. This work aims at highlighting the applicability and the achievable performance improvement of software optimization strategies, mostly relying on compile/link-time activity, and not needing modifications in the SNUCA hardware.

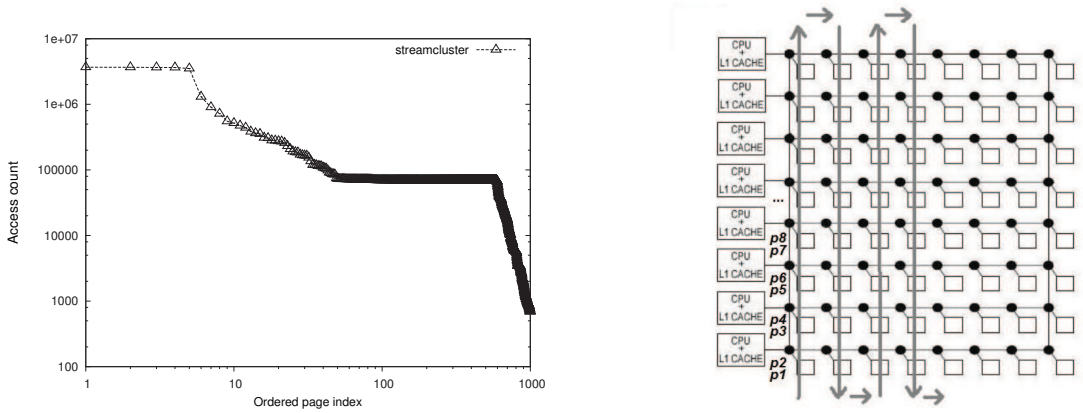
A. Profiling

We profile a sample run of each application and collect the page usage statistics, taking into consideration both the total number of page accesses, broken down into access type (read, write and code fetch), and per-processor quotes. Our functional simulator allows gathering these coarse-grain stats quite quickly and, working on LLC level, with limited approximation in comparison to a cycle-accurate approach. For example, figure 1c shows the access distribution of *streamcluster* application onto the banks of a 8x8 *8p* SNUCA cache, with the processors on the front side of the 3D graph.

As last-level-caches typically work on physical addresses, remapping is implemented inducing a profile-guided association of application pages to the physical memory space. The assumption of relying on the virtual memory management in this way is not constraining as a number of Operating Systems already implement something similar (e.g.,page coloring [7]).

B. Simple remapping scheme

The *simple* remapping scheme is thought for a SNUCA cache with all processors at one side of the bank array. It considers only the overall access count to pages, neglecting the distribution of accesses across processors and the access type (read,write, fetch). This is a big simplification because, pages having different



(a) Page access distribution

(b) mapping order to banks

Figure 2: (a) Access distribution on the first 1000 most accessed pages ($p1 \ldots p1000$), for *streamcluster* benchmark, and (b) exemplification of the *simple* bank-mapping order on the first four bank-columns (grey arrows) and, in particular, of the first eight pages ($p1 \ldots p8$) in case of a 8x8 SNUCA cache.

degree of sharing between threads (and processors) and having different main access typology (load, store, fetch) can have quite different mapping requirements. However, the proposed *simple* scheme keeps the complexity as low as possible and constitutes the reference baseline for our page-level compiler-directed optimizations.

The algorithm starts assigning pages to the cache banks close to the processors (i.e., a bank *column*) in decreasing access count order. As single processor contribution to page accesses is not considered, cache banks within a column are considered equivalent. Figure 2a shows the page access count distribution of the first 1000 most referenced pages of *streamcluster* ($p_1 \ldots p_{1000}$), while figure 2b exemplifies the order of assignments of pages to the first four bank-columns of the NUCA cache (grey arrows) and, in particular, of the first eight pages ($p1 \ldots p8$).

When the page assignment reaches the end of a bank column (vertical swipe), the algorithm starts assigning to the successive column and so on till the last column. Every time the end of the last (first) column is reached, the mapping algorithm changes horizontal direction and goes back assigning to the penultimate (second) column.

This assignment heuristic tends to concentrate most of the accesses on the banks close to the processors, aiming at reducing the average hit time.

In conventional NUCA caches, cache line interleaving is typically adopted to limit NoC link conflicts. Our policy maps pages assuming bank interleaving and leaves the hardware implement cache line interleaving within each bank column. This limits conflicts and still maintains 'hot' references close to the processors.

Note that pages mapped onto the same bank in successive swipes can potentially conflict. This interference (i.e., conflict misses) can be very dangerous to the overall performance because last-level-cache misses are very expensive to serve (hundreds of cycles). Therefore, if misses are not kept under control during remapping, the potential benefits of decreased hit time can easily be overtaken by even a small increase of misses, resulting in higher average access time to the SNUCA.

However, the proposed *simple* ordering tends to avoid conflicts between pages with similar access count, because they are mapped one near to the other. Moreover, the most executed pages can potentially conflict only with pages that are about two *cache pages* far, i.e., after one cache swipe in one direction and another in the opposite, in the mapping order. This reduces the probability that the possible associated conflicts generate many misses.

Operatively, the following equation explains how to build a remapped address that maps a page onto a desired cache bank and positioning within the bank, given the NUCA cache and page features. Let's consider a cache with size C_s, organized in n_x by n_y banks, with each bank having associativity w. Given this, let's define the cache page size $Cp_s = C/w$, the bank size $B_s = C_s/(n_x \cdot n_y)$, the bank page size $Bp_s = B_s/w$ and the virtual memory (vm) page size p_s. A page can be placed in $N_{ppb} = Bp_s/p_s$ different positions within the bank.

Now, the page addresses that map onto the bank (i, j) and in the k^{th} position of the bank ($0 \leq k < N_{ppb}$) are defined by the following equation:

$$addr(c) = c \cdot Cp_s + (((i \cdot n_y) + j) \cdot Bp_s + k) \cdot p_s \quad (1)$$

where, the horizontal and vertical coordinates of the

bank correspond to the highest $log_2(n_x)$ and subsequent $log_2(n_y)$ index bits of the address, respectively. Note that such an address exists for every c that makes $addr$ within the available physical address space.

A straightforward implementation of such mapping scheme, which automatically verifies equation 1, assigns consecutive physical page addresses (i.e., adding or subtracting p_s) while swiping bank columns, and advances and regresses of the size of a bank column when going from a column to the next one.

C. Advanced remapping scheme

This scheme aims at refining the *simple* one taking into account additional information on application behavior. First, individual processor access distributions to pages are used as to privilege page mappings to banks closer to the processors that generate the majority of page accesses. Second, the mapping algorithm is augmented of the guidance of a miss estimator that evaluates the potential conflicts of a page against the already mapped ones.

As in the *simple* algorithm, memory pages are again evaluated in decreasing access count order. When mapping a page, the processor access distribution is used to calculate the cache bank that would result in the minimal overall page access cost from all the processors. This bank constitutes the *base bank* of the search done by the algorithm. In this process, NUCA cache geometry is taken into account and the access cost from each processor is evaluated according to its relative position respect to the target bank. The following equation allows estimating the access cost from a processor pi, positioned at bank coordinate (pi_x, pi_y), to a page p mapped on bank bj, having coordinate (bj_x, bj_y):

$$pgAC = Ni_p \cdot [(|pi_x - bj_x| \cdot C_x + |pi_y - bj_y| \cdot C_y) + bkAC] \quad (2)$$

where, C_x, C_y and $bkAC$ are the cost for traversing one horizontal and vertical NoC link and for accessing the target bank, respectively. Ni_p is the number of accesses to the page from the processor.

Instead of laying out pages into the cache space in a simple sequential fashion, the *advanced* algorithm evaluates a number of possible mappings starting from the *base bank*. The search sequence first evaluates the banks immediately adjacent to the base one and then moves to farther banks. Farther it goes, higher the processors-aggregated hit access cost can be, but this can help to limit potential conflicts.

Mapping many pages onto the same bank can induce numerous conflict misses that, in turn, risk to overwhelm the benefits of lower hit times. This is a very critical tradeoff because a miss in the NUCA last level cache needs accessing off-chip main memory, which requires a long time.

CPUs	8 Ultra Sparc II
Clock	5 GHz (16 FO4 @ 65 nm)
L1 cache	Private 16-KByte I + 16-KByte D, 2 ways, 2 cycles
L2 cache	NUCA 16-MByte, 64 banks (256-KByte banks, 4 ways, 9 cycles)
Block size	64 bytes
NoC	Partial 2D Mesh Network; switch lat.: 1 cycle; link lat.: 1 cycle
Memory	300 cycle lat.

Table I: Features of the reference architecture.

In details, the algorithm takes into consideration the access distribution to the page, using a cache block granularity and heuristically estimates that two pages, if mapped onto the same cache address, can potentially generate a number of misses that is given by the following equation:

$$miss_{est} = \sum_{i \ in \ page \ blocks} min(ap1_i, ap2_i) \quad (3)$$

where apj_i is the access count to block i of page j. Equation 3, for simplicity, assumes uniform time distribution of page accesses and states that conflicting blocks of two pages can generate a number of misses equal to the minimum number of accesses between the two, because the other will hit the remaining times. This is quite precise for direct-mapped banks and would need refinement for associative banks. For the latter, it gives a conservative over-estimate of the miss number and is still able to guide the mapping decisions.

The search for a suitable bank, and positioning within the bank, evaluates the compound cost of potential miss penalty *plus* aggregated hit time, and stops when evaluating a cache positioning that is not yet assigned to any page, if present. Otherwise, exhaustively evaluates all cache positions, which are a fairly reasonable number. For example, a 16 MByte SNUCA cache, organized into 8x8 4-way associative banks with 64-Byte block size, has a bank page size of 64-kByte and thus 16 positions in case of 4-kByte pages. The possible cache mappings of a page in such a cache are 1024, 16 per bank.

This turns into a few seconds run time for our algorithm in case of the fairly complex considered applications. Therefore, the computational complexity of the algorithm is not an issue.

Given the cache mapping address, the physical access is calculated as in the $simple$ algorithm.

IV. Results

In this section we evaluate the performance of the proposed profile-guided remapping optimizations. First,

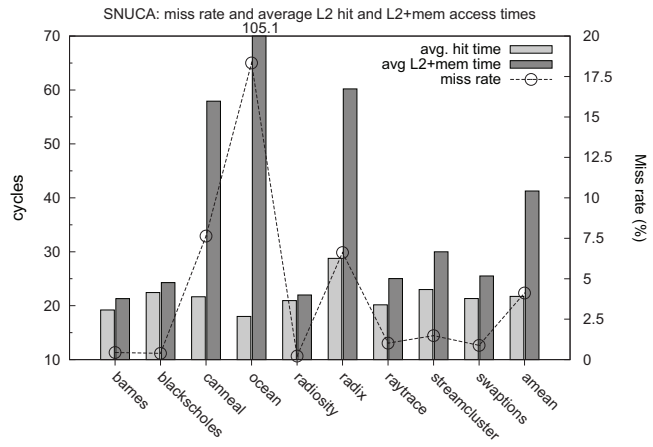


Figure 3: Miss rate (*line*) and SNUCA hit time and overall access time latencies (*bars*) of the considered SNUCA cache.

we introduce the experimental methodology and then we analyze and compare the performance of the *simple* and *advanced* schemes, also against the DNUCA hardware-only counterpart. We finish evaluating the proposed schemes on both *8p* and *4p4* processor-cache interconnections.

A. Methodology

Profile information and performance evaluation of the considered benchmarks are derived from the SimICS-based [8] GEMS [9] full system simulator.

Table 1 summarizes the architectural features of the reference 8-processor system. NUCA parameters and organization are the most performing for the considered technology.

The considered Splash-2 and Parsec benchmarks were run to completion and resulted in 15.3 (*blackscholes*) to 1871.8 (*radiosity*) millions overall L2 references, with an average of 325 millions.

B. Benchmark analysis

In this section we introduce the baseline performance of the SNUCA cache and the metrics we will use throughout the result section. Figure 3 shows the miss rate and the average L2 hit access time and L2+memory access time. The latter represents the overall L2 access time seen by L1 caches, while the hit access time is used as an indicator of the effectiveness of the considered remapping schemes and migration strategy (DNUCA).

Figure 3-*line* shows that most of the benchmark exhibit a limited miss rate, below 1-2%, in the last level cache (e.g., *barnes*, *blackscholes*, *radiosity*), while others (e.g., *canneal*, *radix* and specifically *ocean*) are around 6-8% and up to 18% in case of *ocean*.

Figure 3-*bars* highlights that the average L2+memory access time is quite correlated to the application miss rate. In fact, *ocean* exhibits 105 cycles of average L2 access cost as a consequence of the 18% miss rate. *radix* and *canneal* are somewhat below 60 cycles and score the second highest miss rates. Small deviations can be observed: *canneal* has a slightly higher miss rate than *radix* but exhibits a smaller overall L2 access time. This is mainly due to the combined effect average hit and miss access times, and miss rates. The higher hit-time of *radix* versus *canneal* (28.8 and 21.6 cycles, respectively) weights more than the effect of smaller miss rate (6.62% vs. 7.64%). Then, NoC link and bank contention can cause additional differences.

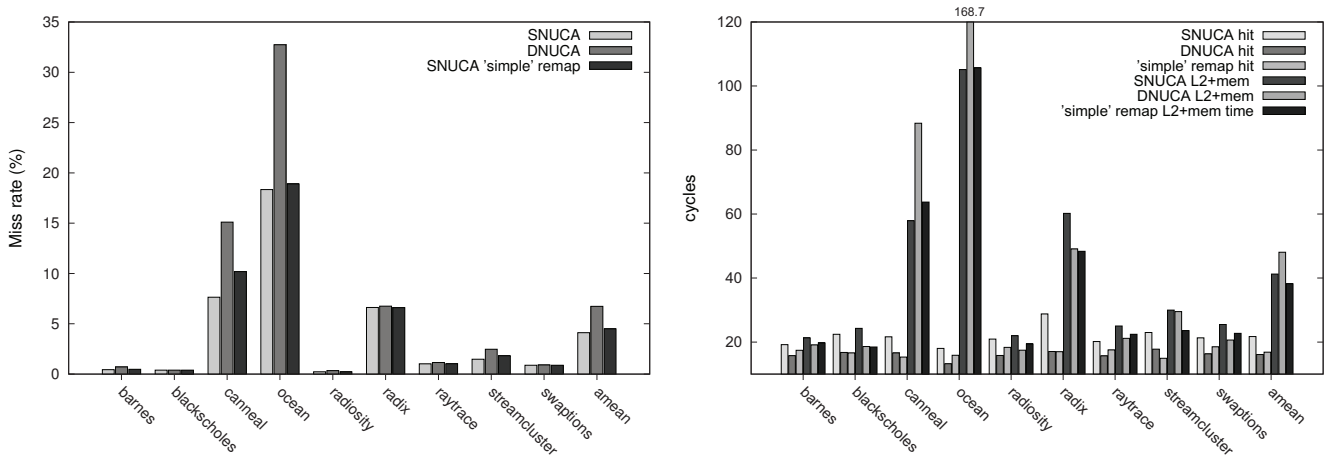
The same figure shows also the average *hit access time*, which falls between 18 (*ocean*) and 28 cycles (*radix*) across all benchmarks. The proposed remapping schemes aim at reducing the average L2+memory access time through profile-guided page mapping.

C. Simple remapping results

The achieved L2 miss rate and average hit and L2+memory access times to the considered SNUCA cache in case of original, *simple* remapped applications, as well as, for a DNUCA cache are given in Figure 4. In particular, figure 4a shows that DNUCA caches incur in a significant miss rate increase over the SNUCA baseline, +58% from 4.3% to 6.8%, on average. However, some benchmarks (e.g.,*blackscholes*, *radix*, *swaptions*) exhibit almost constant miss rate, while others show a dramatic miss rate increase: +93% for *canneal*, +78% for *ocean*, both reaching high absolute miss rates (15% and 32%, respectively). *barnes* and *streamcluster* too suffer from a noticeable increase even if to far smaller absolute values.

The same figure shows also that our *simple* remapping induces only a +7.8% average miss rate increase. Significant increases, even if smaller than DNUCA ones, occur only for *canneal* and *streamcluster*: +32% and 22%, respectively.

Figure 4b shows the hit and overall L2 latency performance. DNUCA caches is able to significantly reduce the hit access time (-25.8% from 21.7 to 16.1 cycles, on average) but this doesn't always translate into a reduction of the overall L2+memory average access time, which instead increases by +17% due to miss rate increase. *ocean* and *canneal* exemplify this correlation. Conversely, other benchmarks like *radix* show a decrease in both hit time and L2 overall access time. In this case, the smaller baseline miss rate and the strong reduction of the hit time (-41% from 28.7 down to 17 cycles) favor the result. However, DNUCA cache reduces the overall L2 access time of six benchmarks out of nine.



(a) L2 miss rate

(b) Latencies

Figure 4: Miss rate (a) and NUCA hit time and overall access time latencies (b) in case of SNUCA, SNUCA with remapped applications and DNUCA caches.

Our *simple* remapping reaches an average reduction of the hit time in line with the DNUCA, even if a bit smaller (-23.7% vs -25.8%). This is not surprising because DNUCA operates at a finer granularity than remapping, cache block (64-byte) instead of memory page (4-kByte) and thus it is more flexible in accommodating the most accessed cache blocks close to the processors. In some cases (*canneal* and *streamcluster*) reaches even slightly better results.

However, the significant metric for performance is the overall L2 access time and the *simple* scheme allows reducing it by 5.5% and 18% over SNUCA and DNUCA, respectively. In some benchmarks, like *blackscholes* and *radix* the reduction is higher, and in line with DNUCA, while *streamcluster* benefits from a big reduction from *simple* but not from DNUCA. Other few benchmarks show a limited slowdown over the DNUCA cache.

Our results are obtained using a profile execution fraction (PEF) of 0.1 (i.e., 10%), while using the whole execution for evaluation. Figure 5 shows the sensitivity of results to PEF, from 0.01 up to 0.3 and highlights that even a small fraction of the execution is able to capture the core and performance-critical portions of the application run. In some cases (e.g., *ocean* and *canneal*), a bigger PEF can even worsen results because, during the complete execution, not all the relatively highly referenced pages in PEF are actually in cache at the same time. This suggests that different inputs could benefit from single-input optimizations [10], especially if the profiling demonstrates to be able to favor the common and performance-critical parts of the application. We will address this issue in details in future works.

Summarizing, we have shown that the proposed *simple* algorithm on a SNUCA cache is able to improve the overall L2 access time of multi-threaded applications and, in particular, to improve over DNUCA results.

D. Advanced remapping results

The *advanced* remapping scheme (figure 6) obtains a higher reduction of the hit time than the *simple* scheme for every benchmark but *canneal* (which increases +3% from 15.3 to 15.8 cycles even if it is still below the unoptimized result). *canneal* is the only benchmark that degrades the overall L2 average access time (+4% over *simple* from 62.5 to 65.25 cycles). This is due to the relatively big working set, compared to the cache size, and to the specific access pattern that pose pressure onto the L2 cache and makes difficult to control misses. All other benchmarks exhibit a very stable reduction both of the hit time and of the overall L2 access time. The achieved hit times are the smallest across the considered configurations and translate into an overall 10.2% L2 access time reduction over unoptimized applications, almost twice as the *simple* scheme, witnessing the benefits of the more articulated remapping strategy.

E. 4p4 NUCA configuration

As shown in [6], the mapping of cache blocks to banks and the topology of the processor-cache interconnection may affect performance. To evaluate the effectiveness of the proposed techniques by varying the NUCA topology, we consider a mapping for *8p* configuration (figure 1a) onto a *4p4* configuration (figure 1b). Figure 7 highlights that the performance improvement is smaller than on an *8p* configuration. Anyway, we

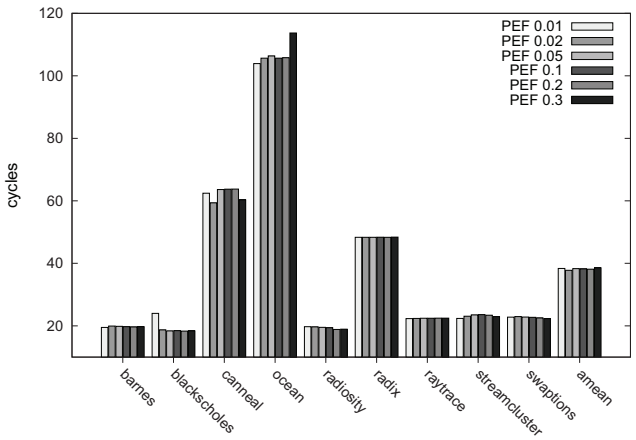


Figure 5: Sensitivity of *simple* scheme performance when varying the size of profiling run. PEF is the fraction of overall execution used for profiling.

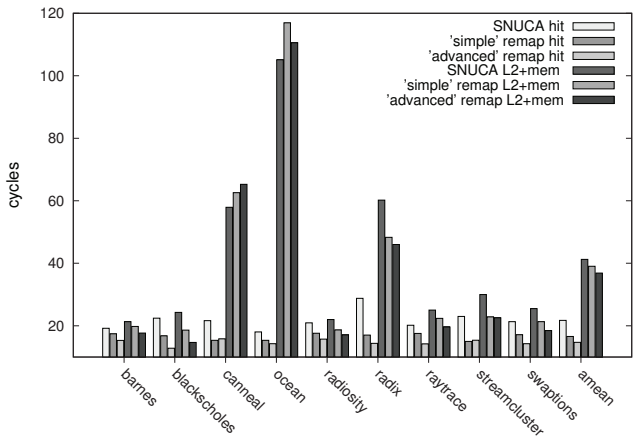


Figure 6: Comparison of the latencies of *simple* and *advanced* remapping schemes.

still achieve some improvement and this is due to the unbalance of the accesses from the processors and due to the specific locality features of the applications, as observed in [6].

However, figure 7 highlights another important point. The performance difference between both original and remapped applications on a SNUCA, compared to DNUCA results is higher in case of the *4p4* configuration. This is because DNUCA caches tend to have shared blocks that are pulled to the left and to the right by the processors at the two cache extremes, resulting in higher hit times than in *8p* configurations.

On this direction, we plan to investigate a compiler-directed replicated mapping scheme that could exploit the benefits of an hardware replication scheme but being able to control, through profiling, which portions of the application should be profitably replicated and which ones should not be replicated to control cache capacity effects.

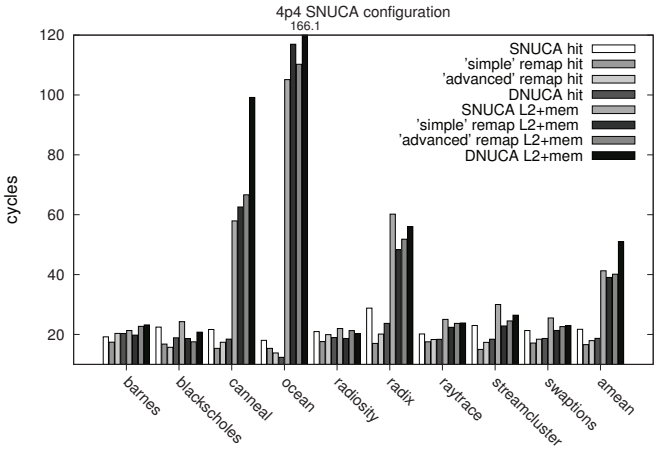


Figure 7: Comparison of the latencies of *simple* and *advanced* remapping schemes on a *4p4* SNUCA configuration, and of a DNUCA cache.

V. Related works

In recent years, a lot of work has been done for managing shared caches in CMPs, at different granularity (e.g.,cache block, virtual memory pages). As in this paper we address page-level technique, we will focus on the main works operating at page granularity and considering a shared cache.

The work by Cho et al. [11] proposes to employ a page coloring algorithm to guide page placement into a SNUCA cache, in case of multiprogrammed mono-process workloads and don't investigate multi-threaded applications. The idea is to let a program take advantage of some cache space from the near cores when it needs more than its available cache. In the present work we specifically focus on multi-threaded parallel applications. Lin et al. [12] explore the application of the Cho et al.proposal to the Linux kernel and highlight to incur in relatively high overheads due to page copying in main memory.

We don't allow migration in the present work, while a lot of work has been done on migration. For example, Chaudhuri [13] evaluates a technique for hardware page-grain migration in NUCA caches on multi-programmed sequential workloads.

Ding et al. [14] apply page coloring to balance the use of colors aiming at conflict reduction in case of mono-threaded applications and not NUCA caches.

Awasthi et al. [15] propose to employ a shadow address space [16] to decouple cache addressing from

virtual and physical addressing. This allows a flexible managing page re-mapping for cache allocation to threads and for placement of shared data structures. They address multi-threaded workloads. Our work aims at investigating the performance achievable through remapping in a standard SNUCA cache, without introducing new hardware structures in it.

In [17], Hardavellas et al.propose a NUCA cache architecture that reacts, R-NUCA, to the different classes of accesses and manages their placement appropriately in the cache. The Operating System cooperation enables migration and replication. The work presented here mainly differs in exploring the remapping opportunities at compile/linking time, while in Hardavellas et al.most of the placement/migration decisions are taken at runtime through the O.S. In addition, we leave replication for future work.

VI. Conclusions

In this paper we have presented two feedback-directed page-level remapping techniques for last-level (LL) SNUCA cache architectures. We have shown for a number of Splash-2 and Parsec multi-threaded applications that a relatively simple remapping algorithm is able to improve the average LL cache+mem access time by 5.5% and surpass DNUCA performance for most benchmarks. Then, we showed that a more sophisticated algorithm reduces the average LL cache+mem access time by 10.2%.

Our results on *4p4* NUCA configuration indicate that further improvement of the proposed strategies could be achieved exploring also profile-guided replication.

Acknowledgment

The authors would like to thank the colleague Manuel Comparetti for the insightful discussions on NUCA caches, and for the tests performed on the simulator[1].

References

[1] M. J. Flynn, P. Hung, and K. W. Rudd, "Deep-submicron microprocessor design issues," *IEEE Micro*, vol. 19, no. 4, pp. 11–22, 1999.

[2] L. A. Polka, H. Kalyanam, G. Hu, and S. Krishnamoorthy, "Package technology to address the memory bandwidth challenge for tera-scale computing," *Intel Techn. Journ.*, vol. 11, no. 3, pp. 197 – 205, Aug. 2007.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, October 2008.

[1]This work was partially funded by EU FP6 SARC project (contract no. 27648), IT FIRB PHOTONICA project (RBFR08LE6V) and IT PRIN 2008 (200855LRP2) project.

[4] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerationss," in *Proc. of the 22nd Int'l Symposium on Computer Architecture*, June 1995.

[5] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002, pp. 211–222.

[6] P. Foglia, F. Panicucci, C. A. Prete, and M. Solinas, "Analysis of performance dependencies in nuca-based cmp systems," in *SBAC-PAD '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 49–56.

[7] "The freebsd documentation project. architecture handbook." www.freebsd.org/doc/en/books/handbook/, 2010.

[8] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[9] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[10] S. Bartolini and C. A. Prete, "A proposal for input-sensitivity analysis of profile-driven optimizations on embedded applications," *SIGARCH Comput. Archit. News*, vol. 32, no. 3, pp. 70–77, 2004.

[11] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *MICRO 39*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 455–468.

[12] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *HPCA*, 2008, pp. 367–378.

[13] M. Chaudhuri, "Pagenuca: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *HPCA*, 2009, pp. 227–238.

[14] X. Ding, D. S. Nikolopoulos, S. Jiang, and X. Zhang, "Mesa: reducing cache conflicts by integrating static and run-time methods," in *ISPASS*, 2006, pp. 189–198.

[15] M. Awasthi, K. Sudan, R. Balasubramonian, and J. B. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA*, 2009, pp. 250–261.

[16] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," *HPCA*, vol. 0, p. 70, 1999.

[17] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ISCA*. New York, NY, USA: ACM, 2009, pp. 184–195.