Mapping Parallel Loops on Multicore Systems

Siham Tabik[1], Felipe Romero[1], Gladys Utrera[2], and Oscar Plata[1]

[1] Department of Computer Architecture
University of Malaga
29071 Malaga, Spain
[2] Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya (UPC)
Barcelona, Spain
{stabik,felipe,oplata}@uma.es, gutrera@ac.upc.edu

Abstract. The compute nodes in contemporary HPC systems contain one or more multicore processors. As a result, these nodes constitute a shared-memory multiprocessor, often combining CMP and SMT concurrency technologies. This configuration introduces different levels of sharing in the cache hierarchy, resulting in non-uniform data sharing overheads. In this paper we analyze the data-sharing patterns that exhibit a real multithreaded application when executing on a multicore system, with emphasis in the use of the shared last level cache (LLC) for the concurrent threads. As a consequence of this study, we explore the loop mapping problem in such systems with the aim of optimizing the shared use of the the LLC by all parallel threads. We propose a three-phase loop mapping strategy that deals with workload imbalances, minimizes cache sharing interferences, and maximizes intra-core and inter-core data reuse in the cache hierarchy. Preliminary results show some benefits of our approach. However, this is a work in progress and much more research is being done.

1 Introduction

The majority of current high-performance computing systems are clusters. The compute nodes in these systems are usually multicore, multi-socket commodity servers and blades. For instance, it is common for a typical node to have 2 or 4 processor chips and each one to have 2, 4, 6 or 8 cores. In addition, some chip manufacturers introduce simultaneous multithreading (SMT) capabilities into their processing units. As a result, each node is a shared-memory multiprocessor, a cache coherent (N)UMA system combining SMT and CMP concurrency technologies. This configuration introduces different levels of sharing in the memory hierarchy (cache hierarchy is partially shared among the cores), resulting in non-uniform data sharing overheads.

In this paper we analyze the data-sharing patterns that exhibit a real multithreaded application when executing on a multicore system, with emphasis in

the use of the shared last level cache (LLC) for the concurrent threads. As a consequence of this study, we explore the task scheduling problem in such systems with the aim of optimizing the shared use of the the LLC by all parallel threads.

In particular, our initial focus is on parallel loops with variable workload per iteration, and with data access patterns that cannot be analyzed at compile time. These loops are present in most realistic applications. Hard issues one faces when scheduling such parallel loops on multicore systems are: i) maximizing data reuse in partially-shared cache hierarchies; ii) minimizing interferences when accessing a shared cache from concurrent threads running on different cores; iii) optimizing reuse of data stored across the same cache hierarchy; iv) reducing workload imbalances over processor cores; v) minimizing synchronization overheads (for instance, when using dynamic scheduling).

Recent work deals with this problem but tackle only a few of the hard issues described above. For instance, one of the works most related with our own one (see related work section) considers only the data reuse in the LLC and compiler analyzable parallel loops. That means that the compiler is able to determine which data sections are referenced in each loop iteration. But, for instance, workload unbalance is not considered at all.

Our work deals with real applications, where we can identify a large time-consuming iterative computation (loop) that can be freely partitioned and distributed among a set of concurrent threads (fully parallel computation). This kind of computation organization is very frequent in different areas, for instance, when solving numerically many real world problems or when applying a repetitive task to a sequence of independent input data objects. In this class of applications, the body of the parallel iterative computation is usually very complex, using a large amount of data structured in many different ways. In addition, the body itself is organized as a sequence of different control structures (other loops, if-then-else sentences, calls to functions and/or methods defined elsewhere ...). All of this results in complex memory reference patterns that usually cannot be determined by the compiler (statically), as well as a variable workload per iteration.

We are developing a cache model for multicore, multi-socket systems that takes into account the inter-core, partially-sharing property. Guided by this model and using the PMUs (performance monitoring units) that most modern processors integrate (run-time analysis), we are designing a loop mapping method that deals with workload imbalances, minimizes cache data sharing overheads (interferences), and maximizes intra-core and inter-core data reuse in the cache hierarchy. Our proposal follows a three-phase strategy. First, a loop partitioning phase, with the aim of obtaining fixed-size blocks of iterations (chunks), in such a way that once assigned to concurrent threads the involved data fits in the LLC. In principle, we will have as many threads as hardware contexts available in the system. This first phase is in charge of minimizing cache interferences among concurrent threads. Second, a loop allocation phase, with the aim of building balanced sets of chunks (super-chunk). These sets are created in two

steps. Initially, individual chunks are assigned to sets using a cyclic scheme. The workload of the obtained super-chunks (sets) may be unbalanced. In that case, a workload balancing algorithm is applied by moving chunks among the super-chunks. This phase is very important because workload balancing is usually a critical performance issue. Third, a loop scheduling phase, where a reordering of the chunks inside each set is accomplished in order to optimize the use of the partially-shared cache hierarchy. As the L2 and/or L3 data caches of contemporary multicore processors are large (12-24 MB L3 cache in the Intel Nehalem "Beckton" architecture, and 2x6 MB L3 cache in the AMD Opteron "Magny-Cours" architecture), this step is becoming an important performance issue even for large parallel tasks.

This is a work in progress and our plan is to show and discuss some of our findings and preliminary experimental results.

The rest of the paper is organized as follows. In Section 2 we review works that are related to this paper. In Section 3, we present a motivating application, that inspired our proposed loop mapping method, which is described in detail in section 4. In Section 5, we present and discuss experimental results based on a large benchmark application. Finally, in Section 6, we draw some conclusions.

2 Related Work

Research on how to take advantage of the partially-shared cache hierarchy present in the current multicore, multi-socket systems in order to improve the execution performance is very recent. However, there is a significant number of works that explore many of the most important issues [1,2,3,4,5,6,7,8,9,10,11,12]. In this section we will discuss only those papers more related to our own work.

Jaleel et al. [1] present a careful study on data-mining bioinformatics workloads. They show that this kind of workloads exhibit a very high data-sharing, about 50 to 95%, and that this data sharing is function of the total cache size available. Taking advantage of this, they report a reduction in memory bandwidth by factors of 3 to 625 when the LLC is shared versus private LLC. Ding and Kennedy [2] present a two-step approach to reduce memory bandwidth requirements within a workload by exploiting data locality. The first step involves fusing computations on the same data to increase the amount of temporal locality. The second step involves reorganizing the data layout to group data used by the same computation to increase spatial locality. They used array regrouping to improve program locality by interleaving the elements of two arrays if they are always accessed together. In [3] authors propose a strategy based on page-allocation at OS level to enforce equal cache sharing by controlling where data is allocated. In this way cache contention is alleviated when multiple cores compete for the usage of a single shared cache.

Recently, hardware performance counters (HPCs) have been increasingly used for many purposes. Kejariwal et al. [4] used L1D cache misses and retired instructions profiles to statically partition the iterations of triangular loops on

multicores. However they have not been considered neither data sharing between iterations nor cache sharing at the hardware level.

Nevertheless, a number of studies have examined the influence of cache sharing on multithreaded applications. For instance, in [5] it has shown that certain configuration of binding threads to cores benefits more from cache sharing. In the domain of parallelizable computations, such as parallel sort algorithms, Chen et al. [6] investigated fine-grained, application-specific scheduling techniques to promote constructive cache sharing on multicore processors. The technique involves decomposing a large, parallelizable computation into threads, and controlling the order and location of thread execution.

An automated compiler-based technique to achieve these goals was explored by Kandemir et al. [7]. This is the most similar work to ours. Their code restructuring scheme is a two-phase method, allocation and scheduling, but with pursued goals different from our approach. As they restrict the study to analyzable loops, that is, loops that the compiler can determine the data accessed in each iteration, both phases reestructures the loop such that different cores operates on shared data blocks at the same time. Allocation exploits inter-core sharing and scheduling exploits intra-core sharing. In our case, as the loop is large and, in general, irregular, we first allocate (variable-size) iteration super-chunks to balance the workload and then schedule (reorder) individual chunks in each super-chunk such that data sharing across partially-shared cache hierarchies is optimized.

3 Motivating Application

In this section we present an application (Wator [13]) that motivated the development of our method for mapping unbalanced and non statically analyzable parallel loops. We analyzed the behavior of the partially-shared cache hierarchy of a commercial multicore system for different iteration distributions of a parallel loop in Wator.

3.1 Wator application

Wator studies the evolution over time of predators and preys in an ocean. The ocean is represented by a rectangular cell matrix where each cell can either be empty or contain an individual, i.e., a predator or prey. In each time step predators and preys can move or replicate themselves, after a certain time period, to closer cells using a 8-point stencil pattern. Preys can move or replicate only to empty cells while predators can eliminate a neighbor prey or die for starvation.

The movement or replication of preys and predators occur in a random direction. Empty cells involve a single load operation while occupied cells usually require from one to nine loads, one or two stores to the previous loaded cell, and tens of integer operations. This makes the distribution of the amount of computation non-uniform across the iteration space. Due to the non-determinism of

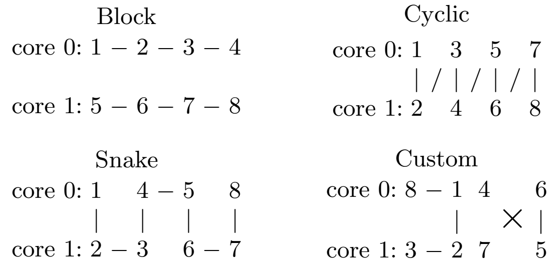

Fig. 1. Iteration distributions used in wator

this application, the order in which iterations are executed does not affect the correctness of the result.

The code below shows the basic structure of Wator. The outer loop is the time-step iterative process, which is sequential. The next two loops are fully parallel and run over the predator and prey domain (ocean). The outer parallel loop ($i2$) is selected for the mapping process, while the inner one ($i3$) will be considered as sequential.

```
do i1 = 1, N1
    doall i2=1,N2
        doall i3=1,N3
            if (condition) f(i1, i2)
        end do
    end do
end do
```

3.2 Wator cache sharing evaluation

To evaluate the behavior of the partially-shared cache hierarchy we have selected the iteration distributions shown in Figure 1, considering two concurrent threads (running in two cores). The parallel loop is sized to 8 iterations ($N2 = 8$). Execution time runs horizontally. Horizontal lines represent that consecutive iterations share data stored in the cache hierarchy of the same core (intra-core sharing). Vertical lines represent that concurrent iterations (running on different cores) share data (concurrent inter-core sharing). Finally, sloping lines represent that non-concurrent iterations share data stored in the cache hierarchy of a different core (non-concurrent inter-core sharing).

Figure 2 shows the results obtained when executing the wator application in parallel (two threads) using the iteration distributions described above. In the horizontal axis, consecutive numbers k and $k+1$ represent two concurrent iterations (depending on the distribution) executed in cores 0 and 1, respectively. All experiments were conducted on a Intel Xeon "Clovertown" based system, where a L2 cache (4 MB) is shared between two cores (see Table 1). We compiled the code using the Intel "icc" compiler with "-O0" optimization level and profiled the resulting executable using Intel Vtune. In particular, Figure 2 shows L2 cache misses (event "L2_LINES_IN.SELF.ANY") for three different sizes of the problem.

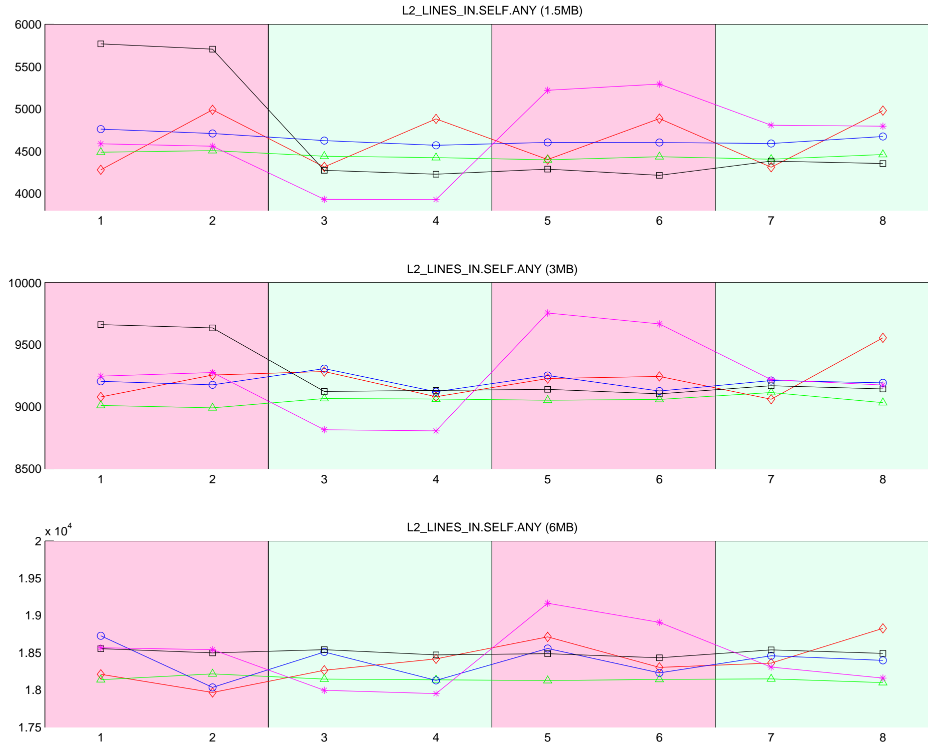

Fig. 2. Shared cache (L2) misses for different iteration distributions in Wator using two threads: block (squares), cyclic (circles), snake (diamonds) and custom (asterisks). Triangles correspond to the sequential execution

We profiled 8 iterations of the selected parallel loop. To preserve the same data access pattern in the last iteration, we connected the first iteration (1) with the last iteration (8) (toroidal ocean). Notice that iteration i shares data with iterations $i-1$ and $i+1$ (modular calculus). We also analyzed the effect of the problem size using three different sizes for the cell matrix: 750K, 1.5M and 3M cells. As each cell occupies 2 bytes we have 1.5 MB, 3 MB and 6 MB total space occupation for the three cell matrices. This values were selected in order to test different fittings into the L2 cache (4 MB).

The four curves in each of the three plots in Figure 2 correspond to block (squares), cyclic (circles), snake (diamonds) and custom (crosses) distributions. In addition, for reference purposes, we added a fifth curve (triangles) that correspond to the serial execution of the 8 iterations.

As it can be observed, the cyclic distribution produces the most stable values for L2 misses per iteration, specially for the smallest problem size (complete fitting in the L2 cache). We observe (see Figure 1) that the cyclic distribution have four vertical lines and 3 sloping lines, showing that it takes advantage, in a

even way, of both, intra- and inter-core sharing. However, a block distribution, that favors most the intra-core sharing (horizontal lines), produces less L2 cache misses. We have also tested two more distributions, snake and custom, in order to evaluate other combinations of intra- and inter-core sharing. The irregularity of these sharing patterns is reflected on the L2 cache misses curve shape. In some cases, even improving the block distribution results.

Let us consider one specific case, corresponding to the last two concurrent iterations of the cyclic and custom distributions, and for the smallest domain size (1.5 MB). This is shown at the rightmost box in the top plot in Figure 2 (horizontal numbers 7 and 8). It can be observed that the custom distribution has a number of L2 misses slightly larger than the cyclic one. On the other hand, Figure 1 shows that both iterations (7-8 in cyclic and 6-5 in custom) have a similar data sharing pattern, except that the custom distribution has more non-concurrent inter-core sharing. This effect may introduce extra cache misses due to contention. This kind of behaviour was taken into account in the design of our proposed loop mapping method.

4 Shared Cache Aware Mapping

From experiments like those discussed in the previous section, we have developed a parallel loop mapping specially suitable for loops with variable workload and with complex data access patterns that cannot be determined at compile time. The main objectives of our method is to balance the workload and to extract maximum advantage from the data sharing properties exhibited by the modern multicore architectures.

Due to the dynamic nature of the considered applications, we resort to profiling techniques to obtain information about the application behaviour in order to guide the mapping process. In particular, our method needs the following information:

- An estimation of the workload per iteration. This parameter could be computed statically, using an analytical or heuristical method implemented in the compiler. However, for many real applications, results from a profiling technique are usually more reliable. For instance, we can resort to event counters like the number of processor cycles or the number of retired instructions.
- An estimation of the data size referenced in each iteration. Similarly, this parameter could be statically guessed, by analyzing the code, or at runtime via profiling. For instance, we can count the demand LLC misses (that is, prefetch misses are omitted) using the apropriate event counter by executing the loop sequentially and flushing the cache between iterations.

Based on the above information, our mapping approach proceeds in the following three phases:

- **Loop Partitioning Phase**: The objective is to calculate the basic chunk size, which depends on the data size accessed per iteration, the size of the LLC, and the number of concurrent threads. The idea is to select a chunk size so as all data referenced by the concurrent threads when executing their corresponding chunk fits into the LLC. As chunks can be executed concurrently in any ordering, the above condition is fulfilled if the size of the data accessed by the chunk is samaller than size of LLC divided by the number of threads (NT), that is, $data_size(chunk) = LLCsize/NT$. In the cases where one iteration overflows the shared cache (LLC), we fix the chunk size to 1. This phase reduces the interferences in LLC due to data accesses from concurrent threads.

- **Loop Allocation Phase**: Once the basic chunk size is calculated, the loop iteration space is partitioned into fixed-size chunks of that size. Then, these iteration chunks are assigned cyclically to NT sets (super-chunks). These sets contain the initial allocation of iterations to threads. After that, using the estimated workload per iteration, we determine the total workload for each super-chunk. If the super-chunks are unbalanced, a balancing process applies by moving some chunks from the heaviest sets to the lightest sets. The objective in this phase is to balance the workload of all super-chunks, that is, to minimize the load unbalance factor, calculated as follows,

$$\mid 1 - (\sum_{i=1}^{NSC} W_i)/(W_{max} \cdot NSC) \mid,$$

  where W_i is the workload for super-chunk i, NSC is the number of super-chunks, and W_{max} is the maximum workload over all super-chunks.

- **Loop Scheduling Phase**: This phase introduces a last process in the balanced super-chunks to be assigned to the threads determined in the previous phase. This process consists in reordering the chunks included in the super-chunks in order to optimize the use of the partially-shared cache hierarchy. There are different approaches to address this problem. One of the strategies we are evaluating consists in reordering the chunks inside each super-chunk in terms of the total number of LLC misses. Assuming that super-chunks are indexed from 0 to $NSC-1$, all even-indexed super-chunks are ordered in increasing number of LLC misses, while the rest (odd-indexed super-chunks) are ordered in decreasing number of LLC misses. Finally, these processed super-chunks are assigned to the threads in indexing order (super-chunk with index i is assigned to thread with index i). The proposed reordering strategy reduces both LLC interferences between co-running threads and the contention in LLC as hot-spot misses are uncoupled. This specific strategy is not designed to exploit data reuse. However, we are working on new methods to tackle this issue.

Table 1. Characteristics of the used multicore systems

System	#cores/socket	#threads/core	L2 (data)	L3 (data)
Clovertown (Core arch.)	4	1	2×4 MB shared by 2 cores	–
Montecito (Itanium arch.)	2	2	2×256 KB private	2×12 MB private

5 Experimental Case Study

As a work in progress, we will show an initial experimental evaluation of a benchmark application taken from PARSEC [14], a recently released suite designed for CMP research. This evaluation was conducted in two commercial systems, based on two different multicore processors, as shown in Table 1. Note that the Montecito-based system has no shared cache among cores. However, we used this processor to quantify the intra-core data sharing between iterations of the parallel loop, with no interference effects from the inter-core data sharing.

In all the experiments, we used the Pthreads library to write the parallel loop and the *pthreads_setaffinity()* function to bind threads to cores. The executable was obtained using the Intel "icc" compiler with maximum optimization level ("-O3").

Ferret is the application benchmark we selected for our initial evaluation of our proposed mapping strategy. Ferret is an algorithm for image similarity search that finds the images most similar to a query image by analyzing their contents. The parallel implementation of Ferret available in PARSEC uses the pipeline model. However, in order to evaluate our loop mapping approach we reformulated this pipeline implementation into a data-parallel one, i.e., a parallel do. Each iteration of the parallel loop reads an image, analyzes it against a database of images, and outputs the list of names of the top 50 similar images. The parallel loop iterates 3,500 times to analyze 3,500 images against a database of 59,695 images. The analysis applied to each image in each iteration consists in segmentation, feature extraction, distance computation, and finally ranking. The size of the data set involved in each iteration is on the order of 80 MB (which is about 10-20x larger than typical shared L2 or L3 caches).

For this application, Ferret, the partitioning phase of our mapping method results in chunk size of 1, due to the size of the data set accessed in each iteration. This way, in the subsequent allocation phase, individual iterations are assigned to super-chunks, having as many of them as the number of considered threads. The amount of computational load per iteration is non-uniformly distributed along the loop iterations, as shown in Figure 3. Similarly, the shared cache misses (LLC) also occur in a non-uniform manner. This situation results in workload unbalanced super-chunks. In fact, Figure 4 shows the workload unbalance exhibited by the Ferret code for different values of the super-chunk size,

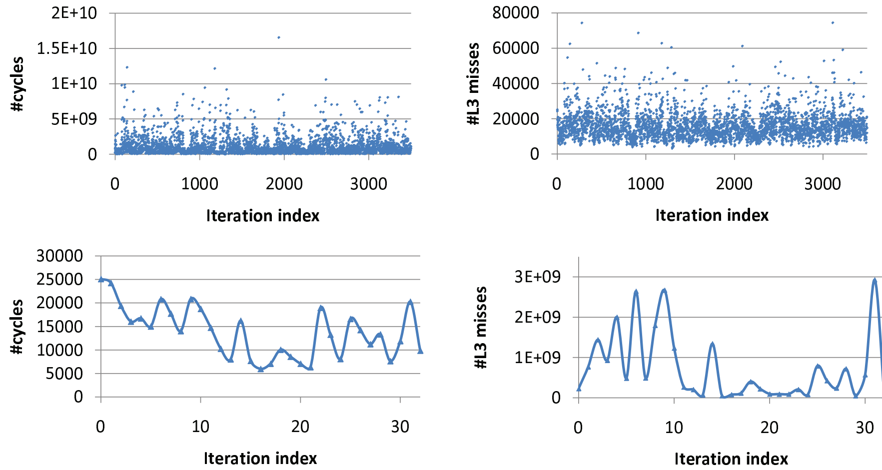

Fig. 3. Number of cycles (top left) and L3 misses (top right) per iteration across 3,500 loop iterations in the serial Ferret code executed on the Montecito-based system (bottom plots are zoomed in areas of the top plots for the first 32 iterations)

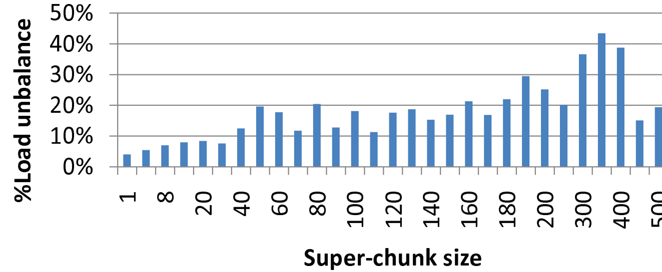

Fig. 4. Workload unbalance for different super-chunk sizes, considering 4 threads

when the parallel code is executed on 4 threads. In this particular case, the initial size of the super-blocks is $3,500/4 = 875$ iterations.

We further analyzed the intra-core data reuse between successive iterations in both serial (see Figure 5 (top)), and parallel execution using two threads (Figure 5 (bottom)), on the Montecito-based system. We used this system to avoid considering interferences from inter-core data sharing among iterations. The curves labelled as "without sharing" were obtained applying a flush of the L3 cache before each single iteration. On the other hand, in the other curves (labelled as "with sharing") this flush operation was omitted.

As it can be observed, the intra-core data sharing (distance between both curves in the top plot) is constant in the serial execution due to the fact that all iterations analyze their private image against the same database images.

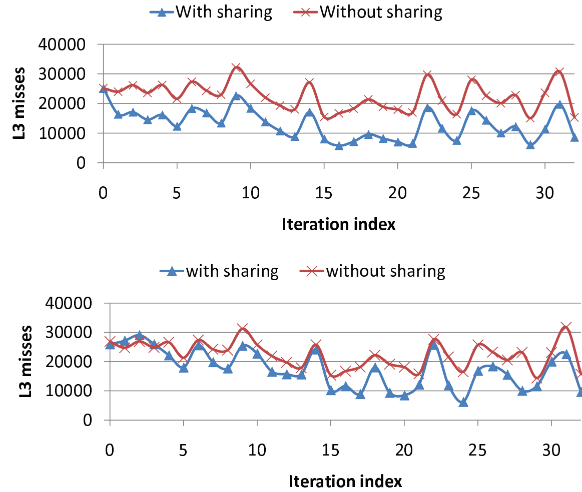

Fig. 5. On the top, L3 misses with and without intra-core data sharing between successive iterations in a serial execution. On the bottom, L3 misses with and without intra-core data sharing between successive iterations in a parallel execution using two threads (a Montecito-based system was used)

Regarding the parallel execution, it can be seen that the intra-core sharing is variable and much lower than in the serial case. This can be explained by the fact that iterations were distributed in a cyclic way among two cores, and the reused data reduces significantly when iterations are not consecutive. In addition, the private cache hierarchy of the core is not exploited completely from one iteration to the next one.

Finally, we evaluated the data-parallel Ferret on two Clovertown cores (that share a L2 LLC) applying the method we proposed in Section 4. Comparing the number of L2 misses and the number of total cycles using our approach versus using a cyclic distribution (which is the optimal distribution) of the iterations between cores we obtained an improvement of about 3% in both L2 misses and the number of cycles. This improvement can be explained by the slight decrease in interferences between cores that share the same cache level. Unfortunately, the resulting improvement is a bit disappointing due to, for this specific application, the amount of shared data is huge, overflowing largely the LLC capacity. In addition, most of the calculations are floating-point operations, so as, globally, the parallel execution introduces a very high memory traffic saturating the memory busses of the Clovertown architecture.

We are currently working on new applications with smaller data sets, comparable to the capacity of the available LLCs, in order to reduce the negative impact of collateral effects.

6 Conclusions

This paper presented and discussed preliminary results about a loop mapping method designed for multicore, multi-socket systems that takes into account the inter-core, partially-sharing property. Specifically, the method deals with workload imbalances, minimizes cache data sharing overheads (interferences), and maximizes intra-core and inter-core data reuse in the cache hierarchy. Our approach follows a three-phase strategy: First, a loop partitioning phase that calculates the chunk size so as the involved data fits into the LLC; second, a loop allocation phase that builds balanced sets of super-chunks; and third, a loop scheduling phase that reorders the chunks inside each set in order to optimize the use of the partially-shared cache hierarchy. Preliminary experimental results with a real application show improvements in both LLC miss rate and execution time. However, for the specific program we tested, this improvement was small due to that the amount of shared data among threads was much larger than the LLC size. As work in progress we are extending the evaluation of our approach to a larger range of applications with shared data comparable with the LLC size.

References

1. A. Jaleel, M. Mattina and B. Jacob, "Last level Cache (LLC) Performance of Data Mining Workloads on a CMP - A Case Study of Parallel Bioinformatics Workloads", *12th Int'l. Symp. on High Performance Computer Architecture (HPCA'06)*, Austin, TX, Feb. 2006, pp. 88–98.
2. C. Ding and K. Kennedy, "Improving Effective Bandwidth Through Compiler Enhancement of Global Cache Reuse", *15th Int'l. Parallel and Distributed Processing Symp. (IPDPS'01)*, San Francisco, CA, Apr. 2001.
3. D. Tam, R. Azimi, L. Soares and M. Stumm, "Managing Shared L2 Caches on Multicore Systems in Software", *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2007)*, (in conjunction with ISCA'07), San Diego, CA, Jun. 2007.
4. A. Kejariwal, A. Nicolau, A.V. Veidenbaum, U. Banerjee and C.D. Polychronopoulos, "Efficient Scheduling of Nested Parallel Loops on Multi-Core Systems", *38th Int'l. Conf. on Parallel Processing (ICPP'09)*, Vienna, Austria, Sep. 2009, pp. 74–83.
5. E.Z. Zhang, Y. Jiang and X. Shen, "Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs?", *15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'10)*, Bangalore, India, Jan. 2010, pp. 203–212.
6. S. Chen, P.B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G.E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T.C. Mowry and C. Wilkerson, "Scheduling Threads for Constructive Cache Sharing on CMPs", *19th Ann. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'07)*, San Diego, CA, Jun. 2007, pp. 105–115.
7. M. Kandemir, S.P. Muralidhara, S.H.K. Narayanan, Y. Zhang and O. Ozturk, "Optimizing Shared Cache Behavior of Chip Multiprocessors", *42nd International Symposium on Microarchitecture (MICRO'09)*, New York, NY, Dec. 2009, pp. 505–516.

8. D. Chandra, F. Guo, S. Kim and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture", *11th Int'l. Symp. on Huigh-Performance Computer Architecture (HPCA'05)*, San Francisco, CA, Feb. 2005, pp. 340–351.
9. S. Srikantaiah, M. Kandemir and M.J. Irwin, "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors", *13th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, Seattle, WA, Mar. 2008, pp. 135–144.
10. J.E. Savage and M. Zubair, "A Unified Model for Multicore Architectures", *First Int'l. Forum on Next-Generation Multicore/Manycore Technologies (IFMT'08)*, Cairo, Egypt, Nov. 2008.
11. D.L. Schuff, B.S. Parsons and V.S. Pai, "Multicore-Aware Reuse Distance Analysis", *9th Int'l. Workshop on Performance Modeling, Evaluation, and Optimization of Ubiquitous Computing and Networked Systems (PMEO-UCNS'2010)*, (in conjunction with IPDPS'2010), Atlanta, GA, Apr. 2010.
12. Y. Jiang, E.Z. Zhang, K. Tian and X. Shen, "Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?", *Int'l. Conf. on Compiler Construction (CC'2010)*, Paphos, Cyprus, Mar. 2010, pp. 264–282.
13. A.K. Dewdney, "Sharks and Fish Wage an Ecological War on the Toroidal Planet Wa-Tor", *Scientific American*, Dec. 1984.
14. C. Bienia, S. Kumar, J. Pal Singh and Kai Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", *17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'08)*, Toronto, Canada, Oct. 2008.