A Locality Approach to Task-scheduling in OpenMP for Tiled Multicore Architectures

No Author Given

No Institute Given

Abstract. Multicore and other parallel computer systems increasingly expose architectural aspects such as different memory access latencies depending on the physical memory address/location. In order to achieve high performance, programmers need to take these non-uniformities into consideration but this not only complicates the programming process but also will lead to code that is not performance portable between different architectures.
Task-centric programming models, such as OpenMP tasks, relieves the programmer from explicitly mapping computation on threads while still enabling effective resource management. We propose an approach to take memory locality into account in the task scheduler of an OpenMP run-time system. This approach make use of programmer annotation to identify memory regions used in a task and a run-time system scheduler making use of this information.
We have made an initial implementation of a locality-aware OpenMP task scheduler on the Tilera TilerPro64 architecture and provide some initial results showing its effectiveness in fulfilling the need to minimize non-uniform data and resource access latency.

1 Introduction

Symmetric multiprocessors are rapidly giving way to multicore systems with great deals of non-uniformity in memory accesses. High-end servers built on AMD Hypertransport or Intel Quickpath interconnects exhibit NUMA (non-uniform memory access) characteristics. As an example, a four-socket AMD server typically connects each processor with two others using Hypertransport. Each processor has one DRAM memory controller so there are at least three different latencies to DRAM memory. Accesses to the local node is the fastest. Accesses to memory belonging to he neighbouring nodes adds some 40 ns and yet another 40 ns is added for accesses to the node which is two Hypertransport links away.

Another example where memory locality matters is in the TilePro64 architecture from Tilera. This architecture exhibits NUMA characteristics as access time to memory slightly depends on how far away (counting in tiles) you are from the memory controller (one out of four) you want to access. Because of its distributed shared cache architecture, it also exhibits varying cache miss latencies as a cache line is always *homed* at some tile's L2 cache, which in this case acts as an L3 cache for other tiles. This characteristics is not always easy to deal with and most often one has to resort to configure the memory, if one has that option, as uniformly far away interleaving the home on the tiles per cache line or whatever granularity is possible.Or, alternatively, the architecture detects access patterns and allocates memory to where it is used the most as in the R-NUCA approach [7].

Task-centric programming models such as OpenMP tasks, Intel Cilk Plus, Intel TBB and Wool [1, 5, 6, 9], are rapidly gaining interest in the parallel software community. The main advantage of a pure task-centric model over thread- and process-centric models, and even data-parallel models are that they implement an efficient "bag-of-tasks" abstraction for all parallel activities, even for parallel for loops that otherwise can be performed efficiently with static scheduling provided that all iterations have the same computational complexity and memory access behaviour.

We argue that for certain applications, we can take knowledge of which data region each task uses and whether these regions are homed at a particular tile or not and schedule tasks to be executed on tiles where they have high likelihood of having its data homed. We have made a prototype implementation of such a scheduling policy in the experimental OpenMP run-time system Nanos++ [2] and present here an initial study on its effectiveness on the TilePro64 architecture from Tilera.As far as we know, no other task schedulers have been presented before that take locality aspects into account. We also propose an extension to the OpenMP directives to allow specification of which data regions a task uses in order to relieve the run-time system from inferring this indirectly.

Although our implementation is on the TilePro64 architecture, we believe that our ideas will hold for any architecture using a Distributed Shared Cache or exhibiting a distributed NUCA behaviour. Future work will include investigating the system with no explicit cache coherence similar to the Intel Single Chip Cloud.

We find that there indeed is performance to be saved by placing tasks at the right core if the memory access pattern is such that it allows it. For the evaluation architecture, however, it is surprisingly hard to predict whether accesses to locally homed data will have substantially lower latency as compared to remotely homed data.

2 Architectural locality

The spreading of processing units and caches across the chip area of existing and future multicore processors leads to an architecture with non-uniform communication latencies which depend on the physical location of on-chip resources. The TilePro64 processor is an example where strong notions of architectural locality exist. The TilePro64 conforms to a distributed shared cache architecture where a processing unit (core) and a slice of the shared last-level L2 cache are bundled into a structure known as a *tile* and tiles are distributed across the chip in a regular manner. With this architecture, a core can access its local shared L2 slice faster than other off-tile L2 slices. In effect, an off-tile shared cache slice becomes an additional L3 cache for that tile.

On the TilePro64, a block of main memory is mapped to a specific last-level cache slice called the *home*. Loads issued by cores are met by first bringing in the block of main memory into the home and then sending the home allocated block to the local L2 slice. Cache misses to blocks of main memory homed locally can therefore be loaded faster than those blocks that are homed remotely. Also, since the TilePro64 employs a write-through cache policy, writes are forwarded to the home tile and invalidations sent to local copies in any other L2 cache. Therefore, cache re-use of data written to is much faster on the tile that homes a particular block. The TilePro64 architecture has in total 64 tiles with 8 kB private L1 instruction and data caches and a 64 kB large slice of the L2 cache which acts as a normal L2 cache and as an L3 cache for all tiles accessing data homed at this tile.

The mapping of main memory blocks to homes is greatly configurable on TilePro64 by means of hashing functions. For example, *hash-for-home* is a vendor hashing function provided by Tilera which spreads all main memory blocks contained within in a OS virtual memory page to a configurable set of homes interleaved on a cache block basis. The default behaviour of hash-for-home, which is the one used in this paper, is that cache blocks are interleaved with homes across all tiles on the chip. It is also possible to configure the home tile of an entire memory page of 64 kB.

Configurable homing of memory blocks on the Tilepro64 is the architectural locality feature which we aim to exploit in this paper. We first begin by constructing a static latency graph of the TilePro64 architecture. The graph is constructed by running micro-benchmarks which measure the latency of communication between shared L2 slices. Using the graph, scheduling and data placement over the chip can be performed with the goal of minimizing the load latencies experienced by OpenMP tasks. The graph is updated by running the latency benchmarks occasionally to get a feel for dynamic communication latencies when the chip is loaded with work.

In order to exploit task memory access patterns effectively on distributed shared cache architectures, allocation of data structures has to be done carefully with communication latency in mind. We illustrate this idea by quantifying the impact of homing decisions while allocating data on the TilePro64. We consider a simple OpenMP application called *home-test* which creates tasks that only makes a number of memory references on specific exclusive regions of memory. These specific regions are either all allocated homed on a single L2 slice, or spread across all available L2 slices using TilePro64 hash-for-home, or every region is homed on a specific L2 slice. We then schedule the tasks to execute one at each core and collect some statistics using hardware counters to illustrate the performance effect of homing decisions. Task execution time is measured using cycle counters and we then also count the number of L1 misses that go to data homed either locally or remotely.

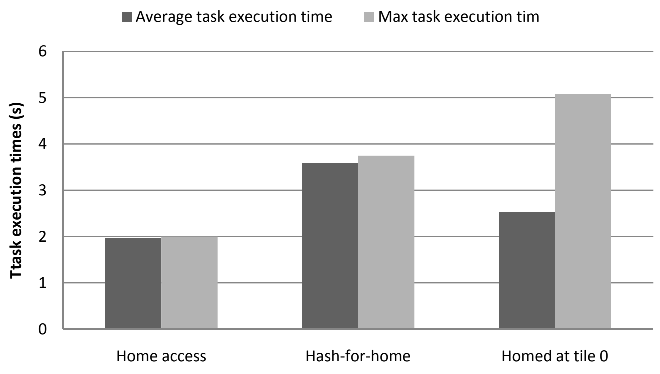

Fig. 1: Average and maximum task execution times.

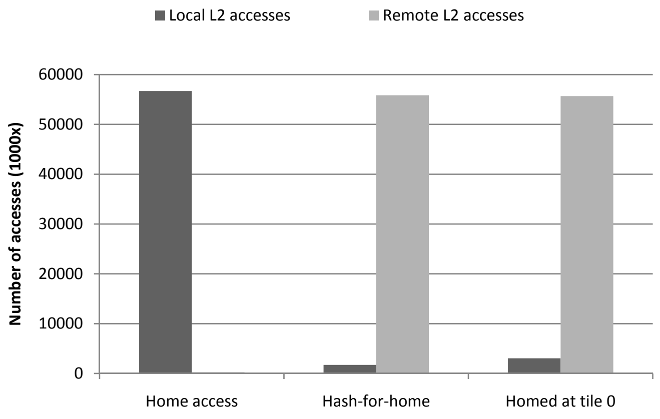

Fig. 2: Average number of accesses to L2 where the home is local or remote.

Figure 1 shows the resulting average task and maximum execution time. The absolute values are not interesting but rather the relative difference between the three allocations. To home data where it is accessed here shows a 43% improvement over the has-for-home policy which distributes the data across all tiles. Hash-for-home represents a uniformly "bad" policy but it has the great advantage that the aggregate L2 caches of all tiles effectively work as a large L3 cache. The single-tile allocation is on average a little better than hash-for-home, since at least one core has local access, but the effect on the maximum task execution time is devastating.

Figure 2 show the causes behind the differences in execution times. It shows the average number of data accesses done by each task to data homed either locally or remotely. The home access scheme has virtually no remote data references which explains the low average and maximum task execution time.

This simple experiment illustrates the importance of taking architectural locality aspects into account when mapping data and/or computations onto a tiled architecture with non-uniform communication and memory access costs. To use this inference effectively, application data structures need to be homed with task access patterns in mind and the corresponding homing information must be conveyed to the runtime. This is easier said than done as it involves substantial compiler, interface design and memory management work. As our research is in its initial proof-of-concept phase, we assume that the programmer can explicitly expose task memory access patterns to the run-time system and allocate application data on specific homes tiles. This is followed by annotating the task definition with a list of home cache id:s called the home-list. The home-list is further sub-divided according to home caches are that written, read or both read and written by the task.

3 Task-centric programming models and memory behaviour

We are concerned here with OpenMP tasks which were introduced to handle applications with irregular parallelism, but also suitable for data-parallel applications for multiprogrammed systems where the parallelism may change during execution time. The nature of the parallel execution graph of such applications can only be unfolded at runtime. Examples of irregularly parallel applications include those perform sorting, searching, sparse linear algebra calculations and volume rendering.

OpenMP tasks are defined as follows:

```
#pragma omp task
{
// This compound statement is scheduled
// to be executed on a core by the
// run-time system
}
```

When a task is encountered during execution, the run-time system may choose to execute it immediately or defer it for later execution. A typical implementation of a task-centric model employs a number of *worker threads* onto which the tasks are scheduled. In OpenMP these worker threads are explicitly created using the `parallel` directive. Different policies for scheduling task execution onto have been studied and implemented and many of them use global task queues or per-core local task-queues with *work-stealing* in order to provide for a load-balancing mechanism.

For an irregular application written using OpenMP tasks, its is not known a priori which tasks will execute where. Although there is irregularity in task creations, a regularity can exist in the memory access behaviour of tasks. If such a regularity is found, tasks can be suitably classified and scheduled on cores such that their memory accesses execute on locally homed data . For the moment we restrict ourselves to an application where we know there is a regular access behaviour, the sparse LU factorization problem from the Barcelona OpenMP Task Suite (BOTS) [4].

Regularity within task memory access patterns can be found during compile time by static analysis. Suitable language extensions can also be provided to the programmer to indicate the regularity himself. Following either mechanism, the compiler can be used to divide and allocate application data structures such that tasks with regular memory access patterns experience minimal access latency to data.

4 Taking locality into consideration in OpenMP

In order to take architectural locality into account in an OpenMP program we need to: (i) be able to communicate to the run-time system which data regions a task is expected to make most accesses to, and (ii) a scheduling algorithm that based on the information from (i) and on knowledge about memory region homing can improve on the access behaviour from a locality perspective. Furthermore there is a choice on who makes the mapping of memory regions to different homing policies. For now, we just assume that the latter is done by the programmer in some ad-hoc manner allocating memory regions to be accessed by tasks homed to tiles in a round-robin fashion with a granularity of 64 kB. When you allocate a chunk of memory to be homed in a particular tile, you always get an integral number of memory pages of size 64 kB.

We propose some extensions to the OpenMP task concept in order to express memory region usage and also discuss a simple scheduling methodology that makes use of this.

4.1 OpenMP directive extensions and compiler support

For this initial study we are for the moment only concerned with specifying important regions of memory used by a task. We have identified the need to specify either a single region of memory used or a sequence of regions.

To allow the expression regularity in the access pattern of tasks, we begin with a simple extension which indicates specific sections of the application data structure which will be read or written by a task. The C syntax of the extension, called range, is shown below:

```
#pragma omp task range(pointer, size, "r|w|rw")
```

The range extension is a clause to `task` construct which specifies the start address and size of shared memory that will read (r) or written (w) or both (rw) by the task. The compiler interprets the specified ranges and passes this information on to the run-time system. For the moment, we assume that the range is homed at some tile and that this information is known to the run-time systems.

In order to specify the situation, which we believe is the most common, when a task touches more than one memory region, we propose a syntax where each region is registered and named with the run-time system and then a task can specify a range of region names it uses.

```
#pragma omp region(name, ptr, size, "r|w|rw")
#pragma omp task ranges(name1, name2, ...)
```

4.2 Home-based scheduling policy

We use the experimental Nanos++ runtime system to implement a currently very simple task scheduling policy called the Home Scheduling (HS) policy which consults the home-list before scheduling a task [2]. Within the Nanos++ runtime system, threads are bound to processors for their entire lifetime. Therefore, every home cache has a corresponding home thread.

The HS policy has distributed task queues, one for each core and for every new task, we pick a core based on the home of one of the regions that this task has specified to use and schedules the task for execution on the associated home thread. The work sharing algorithm of the HS policy can be configured to disregard certain home caches based on whether they are read, written or read-written. To balance the load, the HS policy always picks the least loaded home thread. To balance the load further, the HS policy permits unrestricted work-stealing within a vicinity of cores. The HS policy uses the latency information gathered by the architecture graph and constructs fixed sized vicinities by grouping home caches which have a low latency of communication among each other. Different vicinity configurations are shown in Figure 3.

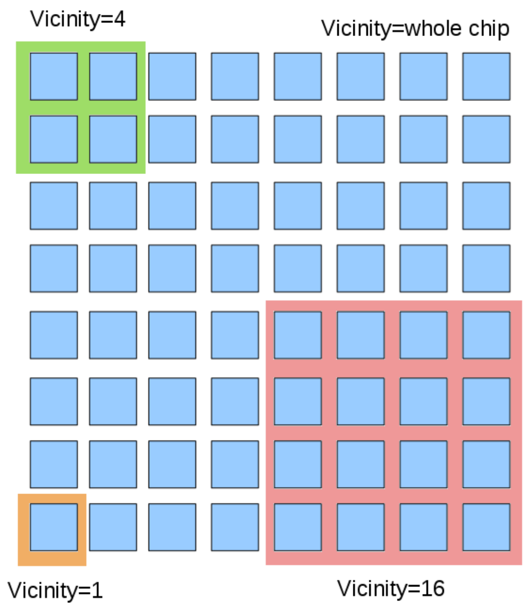

Fig. 3: Vicinity configurations of the HS policy.

A vicinity setting of one leads to a system with no work-stealing. Since tasks are placed on cores based on memory region specifications we are bound to find several tasks sharing regions and therefore leading to load imbalance. A vicinity setting of four allows for work-stealing among the nearest neighbours. This is sensible in systems where the node-to-node latency is a significant factor in the cache miss latency.

There are two obvious improvements to the HS scheme as outlined here, one with respect to work-sharing and one with respect to work-stealing. We could make an analysis on previous work-sharing decisions and avoid placing tasks on cores which already has been assigned a task and instead pick a core based on some other region specification. Currently we only look at the overall load on the possible cores (based on the region specification).

The second improvement is to steal only tasks that already has a memory region specification to the local core who needs to steal work. Currently we steal randomly within the vicinity.

5 Experiments

We consider the SparesLU linear algebra benchmark from the BOTS suite for testing the HS policy. The SparseLU benchmark performs the LU factorization of a sparsely allocated matrix which consists of sub-matrices. The tasks within the benchmark exhibit regular memory access patterns and work on a maximum of three different sub-matrices. With this in mind, we allocate the sub-matrices of the sparse-matrix using two homing schemes. In the first scheme, called the homed scheme, each sub-matrix is homed completely on a specific home cache chosen in a round-robin manner. In the second homing scheme, called the hashed scheme, all sub-matrices are allocated using the hash-for-home feature of the TilePro64. All tasks within the benchmark are marked as untied.

We use the Cilk-like (Nanos_Cilk) scheduling policies of Nanos++ to compare the execution performance of HS as this is also scheduling mechanism with distributed task queues and generally considered to be well performing. The Nanos_Cilk policy is a work-first scheduling policy which means that a task under Nanos_Cilk immediately executes the encountered (child) task and the encountering (parent) task is queued within the executing thread's local queue. Nanos_Cilk threads can steal tasks from any thread's queue but first considers the parent's queue.

We have also studied the Nanos_BF policy which places all encountered tasks in a global queue from which all threads pick tasks for execution. Nanos_BF threads do not have local queues. Normally this is a poor design choice, but for the Sparse LU which does not use a divide-and-conquer algorithm, the BF scheduler provides excellent load-balancing and good performance.

5.1 Methodology

We run the SparseLU benchmark on a 30X30 matrix with 125X125 floating-point sub-matrices using HS, Nanos_BF and Nanos_Cilk scheduling policies for 50 threads on the TilePro64. We allocate the sparse matrix using the hashed scheme for tests with Nanos_Cilk and Nanos_BF. For the HS policy, we allocate the sparse matrix using the homed scheme and annotate the the task definitions with specific homing information. We also perform tests with vicinities of different sizes of 1, 4, 16 and 50. Note that a HS vicinity of 1 implies no stealing and a vicinity of 50 implies full stealing. We use performance counters on the TilePro64 to collect local and remotely homed cache access statistics.

5.2 Results

Figure 4 shows the execution times of Sparse LU factorization for the different scheduling policies. For the home schedule policy, we have used different vicinity settings stealing between 1 (no-stealing), 4, 16 or all 50 cores. We see here that home based scheduling with stealing performs the best with BF scheduling approximately as good. Home scheduling without work-stealing obviously leads to load imbalance also illustrated in figure 5.

In order to study the effectiveness of the home scheduling policy we have measured the number of L1 cache misses that have their homes in the local node or in a remote node, respectively. This is shown in figure 6. Clearly home based scheduling without work-stealing has the highest fraction of local accesses but poor performance without using work-stealing.

6 Related work

Distributed shared cache architectures are relatively new and efforts to exploit architectural locality on them are few and recent. Our work draws motivation from one of the early efforts to analyze the impact of data distribution across distributed on-chip caches by [8]. Realizing the need

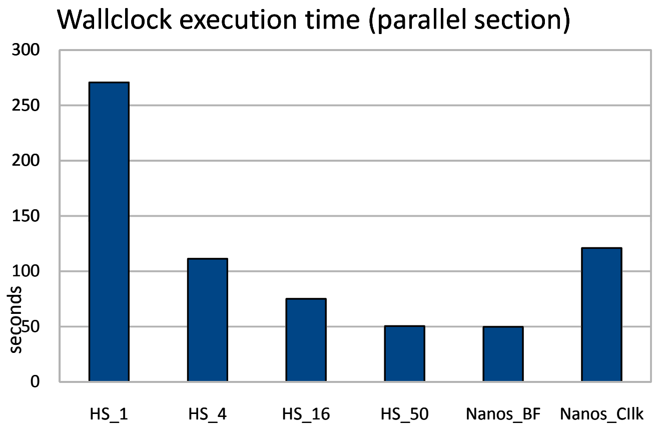

Fig. 4: Execution time of Sparse LU for different scheduling policies.

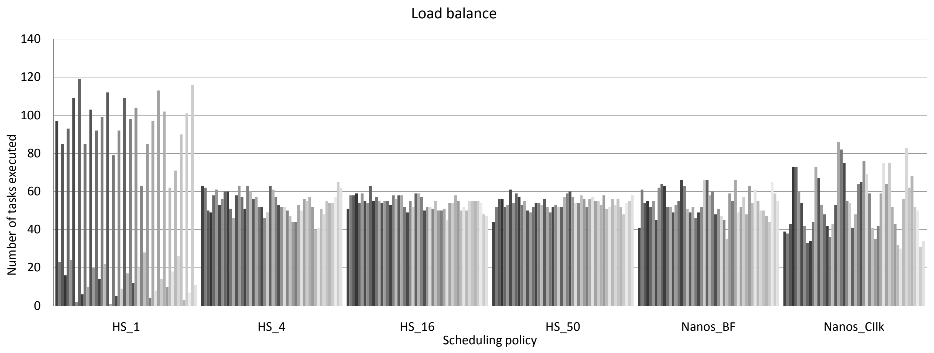

Fig. 5: Number of tasks executed per thread for different scheduler types.

to match program memory access behavior with hashed memory block mapping schemes imposed by hardware, they perform compiler-based data-layout transformations such that accesses to remote caches are minimized. Our work aims to perform the similar data-layout transformations dynamically at the runtime level supported by programmer hints and architecture awareness. [3] in their recent express concerns over deepening memory hierarchies on modern processors and implement runtime OpenMP thread and data placement strategies guided by programmer and compiler hints and by using architecture awareness. Our work aims to progress in a direction similar to theirs. The idea of keeping tasks

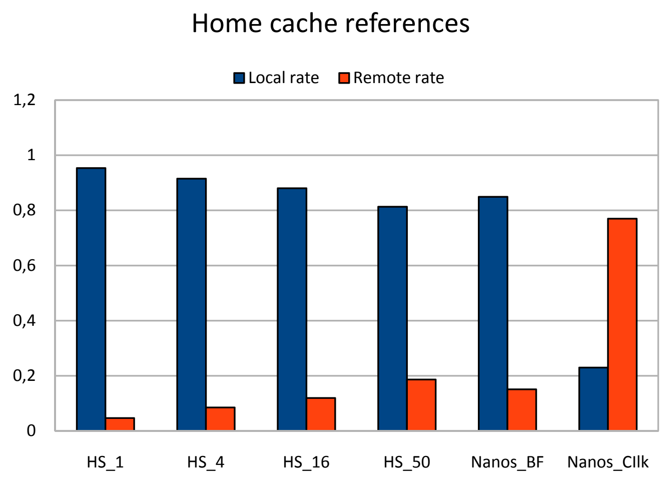

Fig. 6: The ratio between local vs remote memory accesses for the different schedulers.

and associated data close together on a given architecture is key in high-performance computing languages such X10 and Chapel. A recent work on these languages by [10] builds a tree like notion of the memory hierarchy and first allocates user defined data structures on this tree followed by an affinity-based placement of tasks. Our work is similar in principle, but currently relies on the programmer to perform allocation of data on the cache hierarchy.

7 Conclusion

We have presented an initial approach on how to deal with architectural locality for OpenMP tasks and exemplified it as a prototype implementation in a run-time system and measured some key aspect on the Tilera TilePro64 architecture.

While there are some obvious improvements to be made to our scheduling policy, we still have some encouraging results displaying how we can utilize the fact that the home mapping of cache blocks onto tiles to improve performance. Our immediate future work include, besides implementing the already outlined improvements, studying applications into more detail to understand when locality can and should be exploited.

References

1. Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
2. J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 2004, 2004.
3. François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-André Wacrenier, and Raymond Namyst. Structuring the execution of OpenMP applications for multicore architectures. In *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, GA, April 2010. IEEE Computer Society Press.
4. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP '09)*, page 124–131, Vienna, Austria, September 2009. IEEE Computer Society, IEEE Computer Society.
5. Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, 2008.
6. Robert Geva. Elemental functions: Writing data-parallel code in c/c++ using intel cilk plus. White paper, Intel Corp., url: http://software.intel.com/file/33747/.
7. Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. R-nuca: Data placement in distributed shared caches. In *International Symposium on Computer Architecture*, 2009.
8. Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-fook Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 348–357, Washington, DC, USA, 2009. IEEE Computer Society.
9. J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2007.
10. Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *LCPC'09*, pages 172–187, 2009.