# Task Scheduling on Manycore Processors with Home Caches

Ananya Muddukrishna, Artur Podobas, Mats Brorsson, and Vladimir Vlassov

KTH Royal Institute of Technology, Sweden

**Abstract.** Modern manycore processors feature a highly scalable and software-configurable cache hierarchy. For performance, manycore programmers will not only have to efficiently utilize the large number of cores but also understand and configure the cache hierarchy to suit the application. Relief from this manycore programming nightmare can be provided by task-based programming models where programmers parallelize using tasks and an architecture-specific runtime system maps tasks to cores and in addition configures the cache hierarchy. In this paper, we focus on the cache hierarchy of the Tilera TILEPro64 processor which features a software-configurable coherence waypoint called the *home cache*. We first show the runtime system performance bottleneck of scheduling tasks oblivious to the nature of home caches. We then demonstrate a technique in which the runtime system controls the assignment of home caches to memory blocks and schedules tasks to minimize home cache access penalties. Test results of our technique have shown a significant execution time performance improvement on selected benchmarks leading to the conclusion that by taking processor architecture features into account, task-based programming models can indeed provide continued performance and allow programmers to smoothly transit from the multicore to manycore era.

## 1 Introduction

Faced with increasing performance demands, chip manufacturers have begun to introduce manycore processors with tens to hundreds of cores in major electronics domains. Manycore processors support a large number of cores using highly scalable architectural features such as a distributed cache hierarchy, a high bandwidth on-chip network and multiple memory controllers.

Writing software which can scale to all cores of a manycore processor is a formidable task. In addition to mapping parallelism efficiently onto cores, manycore programmers must pay close attention to memory behavior and on-chip communication of their applications. To further complicate the matter, manycore processors expose a variety of software-controllable architecture features which must be configured properly to achieve the best application performance.

Task-based programming models such as OpenMP, OmpSs [1], Cilk Plus and Intel TBB represent an important step towards easy parallel programming on shared memory machines. These models essentially allow the programmer to forget about threads and focus on expressing application parallelism using structures known as *tasks*. Tasks are internally scheduled on threads by a dynamic component called the runtime system.

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 357–367, 2013.

By balancing the task load on threads, existing runtime systems have been able to provide good portable performance on several generations of multicore processors [2]. To continue with the same performance trend on manycore processors, task-based runtime systems will have to turn a variety of architecture-specific knobs and schedule tasks on threads more intelligently by considering chip-level aspects such as task data affinity, thread binding, cache communication latency and memory controller bandwidth.

In this paper, we report our efforts to improve task-based runtime system performance on the Tilera TILEPro64 manycore processor which features a banked chip-wide distributed software-configurable L2 cache. The cache coherence protocol of the TILEPro64 orchestrates coherence actions for each cache block from a specific bank of the L2 cache known as the *home cache*. Depending upon the location of the home cache, the access latency experienced by cores for a missing cache block can vary. We first characterize the home cache dependent non-uniformity in access times to cache blocks. Next we show how home cache oblivious task scheduling by the runtime system incurs significant execution time performance degradation. We then present a home cache aware task scheduling technique in which the runtime system implicitly controls the home cache affinity of cache blocks and schedules tasks to minimize home cache access penalties. Finally we present and explain test results of our home cache aware scheduling technique which show a significant improvement in execution time performance in comparison to blindly load-balancing runtime systems.

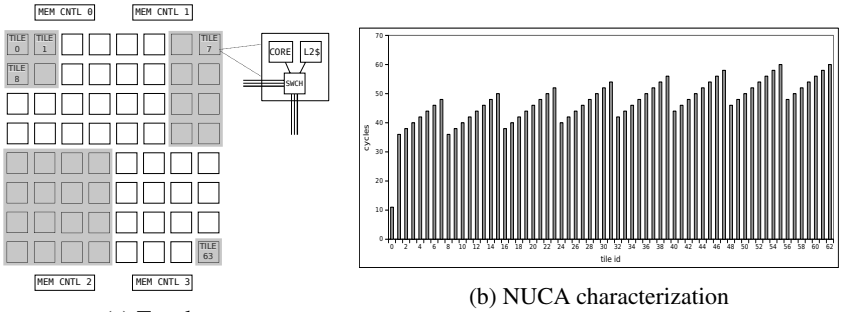
## 2 Manycore Architectures with Home Caches

In this section we introduce manycore architectural features by using the TILEPro64 as an example. In particular, we highlight the home cache feature and its impact on task scheduling performance.

### 2.1 TILEPro64 Processor Architecture

The Tilera TILEPro64 is a 64-core tiled architecture processor whose tiles are connected by a 8X8 multi-link mesh on-chip network. Each tile contains a 32-bit VLIW integer core, a network switch, a private 16 KB L1I cache, a private 8 KB DL1 cache and a 64 KB bank of the shared L2 cache whose aggregated capacity from 64 tiles is 4 MB. The topology of the TILEPro64 processor is shown in Figure 1a. In the topology illustration, L1 caches are considered to be part of the core and not shown. The grayed sections are explained in a later section of the paper.

The TILEPro64 provides hardware cache coherence whose actions are configurable by user-level software. The cache coherence protocol allocates every cache block sized chunk of main memory in a specific bank of the L2 cache known as the *home cache*. For simplicity, we refer to the tile that contains the home cache of a memory block as the *home tile*, and those that do not as *remote tiles*. The home cache is used to satisfy load and store requests from all tiles. Upon a load miss in the L2 cache of a tile, the home cache is requested to provide the missing block. Depending on the software configuration, the block delivered by the home cache is allocated selectively in both or either of the L2 and L1 caches of the tile. Stores in a tile are always write-through to the home cache, with a store update if the block is found in the L1.




(a) Topology

(b) NUCA characterization

**Fig. 1.** TILEPro64 features

The home cache of any given cache block is selectable by software. The TILEPro64 provides the *hash-for-home* allocation scheme using which cache blocks in a main memory page are uniformly distributed on a system-wide set of home caches [3]. As an alternative, all cache blocks within a main memory page can be allocated in a single home cache. In addition, it is possible to change the home cache of a given cache block at a high migration cost.

The load latency of a missing cache block depends on the physical on-die location of the home cache. This asymmetry in cache load latency represents the Non-Uniform Cache Access (NUCA) nature of the TILEPro64 and is characterized in Figure 1b for a cache block whose home cache is in tile id 0. We can see that accesses from tiles other than tile id 0 (remote tiles) incur a 4 to 6 times increase in load latencies. We note the NUCA latencies shown in Figure 1b are measured for isolated tasks under minimal interference conditions. The actual observed NUCA latencies are indeed worse due to multi-programming, OS and hypervisor interference. We also note the NUCA latencies shown in Figure 1b are measured for tile ids up to 62 only. This is because one tile on the TILEPro64 is given up to run dedicated system software.

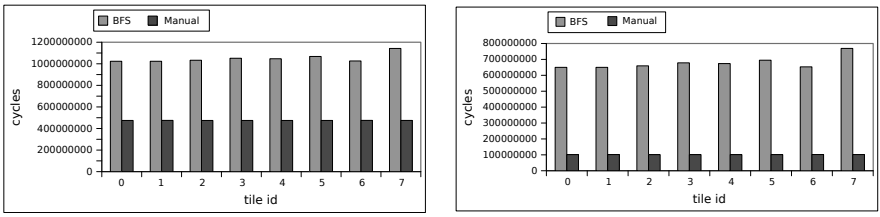
### 2.2 Home Cache Performance Impact

We now illustrate the performance degradation of home cache oblivious task scheduling using a synthetic program shown in Listing 1.1. The synthetic program is written using the OmpSs programming model which extends OpenMP with support for implicit synchronization of tasks using array-style data dependence annotations called `in, out` and `inout`. The synthetic program first allocates and initializes `N` blocks of data, each of size `SZ` using plain GLIBC `malloc` which internally uses the hash-for-home scheme by default on the TILEPro64. `N` tasks are then created to independently apply the transformation function, `transform`, on the `N` data blocks. We consider the performance of two different schedules for tasks of the synthetic program. The first schedule is a commonly used central queue schedule called the Breadth First Schedule (BFS). BFS represents a self-scheduled execution where newly created tasks are added to a central and made available to idle threads for execution. The second schedule is a manual schedule which is hard-coded by the programmer such that each data block has

an unique home cache and each task executes on the home tile of its data block. For N=8 and SZ=32 KB, Figures 2a and 2b respectively show the per core execution time and data cache stall cycle performance of the two schedules. We can clearly see that execution time of the home cache oblivious BF schedule suffers due to non-uniform data cache stall cycles resulting from the hash-for-home allocation of data blocks. In comparison, the manual home cache aware schedule outperforms BFS since all tasks benefit from local access to the home cache of data blocks.

```
for(int i=0; i<N; i++) {
 list[i] = malloc(sizeof(int) * SZ);
 initialize (i, list[i], SZ);
}
for(int i=0; i<N; i++) {
#pragma omp task inout(list[i][0:SZ-1])
transform(list[i], SZ);
}
#pragma omp taskwait
```

**Listing 1.1.** Synthetic program to illustrate impact of home caches




(a) Execution time

(b) Data cache stall cycles

**Fig. 2.** Synthetic program schedule performance

The manual schedule has an obvious drawback - it requires the programmer to use TILEPro64 specific routines to explicitly set the affinity of application memory and tasks. Such an explicit responsibility is counter-productive to the programmer-friendly goal of task-based programming models which expect the programmer to only define tasks. In order to benefit from home cache aware scheduling and yet retain the programmer-friendliness of task-based programming, we decided that home cache assignment and home cache affinity based task scheduling had to become implicit runtime system responsibilities. Our implemented of the same is described in the next section.

## 3 Implicit Home Cache Aware Task Scheduling

We implemented implicit home cache aware task scheduling using two abstracted runtime system functions called the *memory allocation policy* and the *scheduling policy*. The memory allocation policy controls the assignment of home caches to dynamically

allocated application memory using TILEPro64 specific system calls. In our implementation, the memory allocation policy exposes two programmer interfaces called `rt_malloc` and `rt_free` as shown in Listing 1.2. These interfaces are designed similar to GLIBC `malloc` and `free` to allow trivial translation by the programmer and or source-to-source compiler.

```
void* rt_malloc(void* ref_ptr, size_t size_in_bytes);
void* rt_free(void* ptr);
```

**Listing 1.2.** Memory allocation policy interfaces

We implemented two types of memory allocation policies:

1. Round allocation policy: This policy assigns a single home cache to all memory pages requested by the call to `rt_malloc`. To assign the home cache, the policy chooses from pre-configured set of home caches in a round-robin manner. If the call to `rt_malloc` has a valid `ref_ptr` argument, the round allocation policy assigns the home cache of data pointed by`ref_ptr` to the currently requested allocation.
2. Hashed allocation policy: This policy uses hash-for-home allocation over a pre-configured set of home caches for all memory pages requested by the call to `rt_malloc`. The policy overrides the default system-wide set of home caches used by hash-for-home. The policy ignores the `ref_ptr` argument of `rt_malloc`.

The type of memory allocation policy is selected by a person or software that we call the *tuner*. We assume that the tuner is responsible for configuring the startup parameters of the runtime system. The tuner also decides the set of home caches used by the round and hashed policies. The memory allocation policy maintains a record of all allocations made using `rt_malloc`. Information from this record can be queried from other components of the runtime system components such as the scheduling policy.

The other half of our home cache aware scheduling mechanism is the scheduling policy called NUCA Schedule (NUCAS). During startup, NUCAS binds threads to cores using a 1:1 map. Therefore while describing NUCAS, we use the terms thread and core interchangeably. NUCAS schedules each task on the core which has the least latency of access to the home caches assigned to the dependences of the task. NUCAS determines the home caches of task dependences by querying the records of the memory allocation policy. Access latencies to task dependences are computed using a core-to-home-cache communication cost matrix whose entries are populated by calculating on-chip network latencies during runtime system initialization. Indeed, the NUCA characterization in Figure 1b graphically depicts the first row of the core-to-home-cache communication cost matrix used by NUCAS. Since the write buffer on the TILEPro64 absorbs store latencies to home caches, we designed NUCAS to consider only input dependences of tasks.

Given a task with multiple input dependences whose home caches are spread out, NUCAS checks the access latency of each core to the home caches of all task dependences and chooses the core with the least access latency. To speed up the checking process, NUCAS uses a simple heuristic for tasks with `D` equal sized dependences.

The heuristic simply chooses the core associated with the home tile of the last dependence in the list of `D` dependences. The heuristic is reasonable since scheduling a task for local access to the home cache of a single dependence on the TILEPro64 is latency-wise comparable to scheduling the same task for non-local but least latency access to home caches of all dependences. The choice of `D` is left to the tuner.

NUCAS associates a task queue with every core part of the runtime execution. To balance the load on the task queues, cores are grouped into fixed size vicinities and are allowed to steal tasks from other cores belonging to the same vicinity. The idea behind vicinity-based task stealing is that access latencies to home caches within a vicinity are logically considered to be the same. Vicinity sizes of 1, 4, 8, 32 and 63 tiles are made available to the tuner and their arrangement is shown in grayed sections of Figure 1a. A vicinity size of 1 implies cores never steal and a size of 63 implies all cores steal from each other. In our implementation, the default vicinity size is 1.

## 4 Experimental Setup

Since the TILEPro64 has many configurable knobs, we made the following assumptions to fix the architecture. We assumed that all loads and store requests issued by cores are processed by home caches only. To realize the assumption, we disabled local L1 and L2 caching on the TILEPro64. By disabling local L2 caching, we minimized the adverse effect of evicting local L2 cache entries to memory. By disabling L1 caching, we brought the NUCA impact of home caches out to the front. This move additionally allowed us to make a technology projection to home cache based architectures larger and slower than that of the TILEPro64.

We used four task-based applications as benchmarks to test the performance of NUCAS. Two of the benchmarks are synthetic but with real world execution patterns. The benchmarks are described below:

1. Map: This is a synthetic benchmark whose execution resembles the map phase of the Map-Reduce programming framework. The benchmark allocates data in chunks and creates tasks which independently operate on unique data chunks. The benchmark is exactly similar to the synthetic program shown in Listing 1.1.
2. Aggregator: This is a synthetic benchmark whose execution resembles the merge phase of the Mergesort benchmark of the Barcelona OpenMP Task Suite (BOTS) [4]. The benchmark's task dependence graph begins with a execution similar to the Map benchmark and later unfolds as an inverted tree. Each task in the inverted tree reads from two to three input dependences, operates on the read data and writes results to a single output dependence, thereby performing a reduction operation.
3. Vector Multiplier: This benchmark uses tasks to perform vector transformations commonly seen in scientific applications. Each task of the benchmark iteratively applies a set of multiply and add operations on two input vectors and stores back the result into one of the input vectors. The input vectors are not co-allocated in the same home cache. Tasks of the benchmark are independent, therefore the dependence graph of the benchmark is flat.

4. SparseLU: This is a benchmark based on the SparseLU benchmark of BOTS. This benchmark uses tasks to perform the LU factorization of the input sparse matrix. Each task of the benchmark reads from two to three blocks of the sparse matrix and stores back the result into one of the blocks. Two classes of tasks access one of the input blocks more intensely than others. Tasks of the benchmark are not independent and the task dependence graph is complex.

While selecting benchmark parameters, we had to ensure that the NUCA impact of home caches was not masked by other architectural features of the TILEPro64. We considered integer versions of the benchmarks to rule out effects of slow floating-point operations which on the TILEPro64 are emulated using software. The benchmark data sizes were carefully selected to minimize off-chip memory accesses which take about 180 cycles on the TILEPro64. In addition, we zeroed the interval of the operating system tick scheduler to minimize thread switching effects. Finally, the benchmarks were run in isolation to avoid cache pollution effects from other applications. The benchmark input parameters and related information are summarized in Table 1.

**Table 1.** Study benchmark parameters

| Benchmark | Input | Tuner Input | Number of tasks |
|---|---|---|---|
| Map | 63 chunks, each 16 KB | Cores=63, D=1 | 63 |
| Aggregator | 48 chunks,each 16 KB | Cores=48, D=3 | 94 |
| Vector Multiplier | 128 integer vectors, each 4096 integers | Cores=63, D=2 | 4096 |
| SparseLU | 32X32 blocks, each 36X36 integers | Cores=63, D=3 | 3281 |

We implemented NUCAS and memory allocation policies using an in-house experimental task-based runtime system called MIR. MIR supports task scheduling and implicit task synchronization. In addition, MIR records core execution states and hardware performance counter events and dumps trace files viewable in Paraver [5] which is a powerful parallel execution visualization tool developed by the Barcelona Supercomputing Center (BSC).

To compare NUCAS, we used the BFS schedule. The execution of BFS is based on a similarly named scheduling policy described in a study of OpenMP task scheduling strategies by Duran et al [6]. BFS associates a single FIFO task queue with all cores part of the runtime execution. BFS queues all application tasks into the single queue. When idle, each core removes and executes the task from the FIFO queue. Due to this random self-scheduling nature, BFS is fast and provides aggressive load-balancing for medium-grained tasks. We did not implement a distributed queue scheduler for comparison since our study benchmarks have medium grained tasks all created by a single thread [2]. We also stress that we do not intend to make an apples-to-apples comparison of scheduling policies. The idea behind our choice of comparison was to judge the performance of NUCAS which optimized for home caches against BFS which did not.

We selected the following combinations of scheduling and memory allocation policies: BFS-hashed, BFS-round and NUCAS-round. The BFS-hashed policy can be considered as the only option available to the tuner on the TILEPro64 in the absence of the home cache aware scheduling mechanisms. We consider the BFS-round case as an interesting interconnect bandwidth optimizing experiment in which the programmer explicitly allocates application data on different home caches but the scheduler is oblivious to the distribution and schedules tasks randomly. Finally in the NUCAS-round case, we characterize a runtime system execution where application data is implicitly allocated on different home caches and passed on as useful information to the scheduler which in turn schedules tasks to minimize NUCA penalties of home caches. We do not consider the NUCAS-hashed case since it unfeasibly involves minimizing NUCA penalties to cache blocks spread out on all available home caches. To add, system software of the TILEPro64 does not provide an user-level interface to obtain the home cache mapping on a cache block basis for hash-for-home allocated data.

## 5 Experimental Results

Figure 3 shows the execution time performance of our study benchmarks under different scheduling and memory allocation policy combinations which we simply refer to as *schedules*. The time measured corresponds to the parallel section of the benchmarks. For the Vector Multiplier and SparseLU benchmarks, performance of task stealing under different vicinity sizes is also shown. Since the Map and Aggregator benchmarks do not improve with vicinity-based task stealing, we only show the NUCAS-round result for a vicinity size of 1 for these benchmarks. Figure 3 clearly shows that NUCAS-round produced the fastest schedule for all the benchmarks.

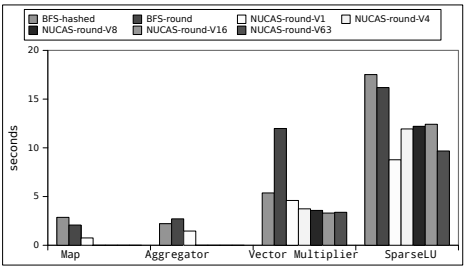



**Fig. 3.** Execution time performance of study benchmarks

In order to ascertain that home cache aware task scheduling, or the lack of it, was the reason behind the performance results seen in Figure 3, we observed core execution traces complemented with cycle counter and data cache stall cycle counter values on Paraver. Columns 4a- 4d in Figure 4 shows the Paraver *view* of core execution traces for all study benchmarks. The rows of Figure 4 from top to bottom respectively show the views of BFS-hashed (B-h), BFS-round (B-r) and NUCAS-round (N-r) schedules. The NUCAS-round view corresponds to the best performing vicinity size in Figure 3.

Within each view, the Y-axis marks cores and while the X-axis shows cycles. Each view shows a per core timeline, i.e., when and for how long each core executed tasks, using colored bars. The color of each bar is encoded using a gray gradient. Light and dark gray bars respectively indicate low and high data cache stall cycle values. The three schedule views of a benchmark are relative to each other - they are cycle aligned to the largest execution time and are semantic aligned to fit the entire range of data cache stall cycles values seen among all schedules.

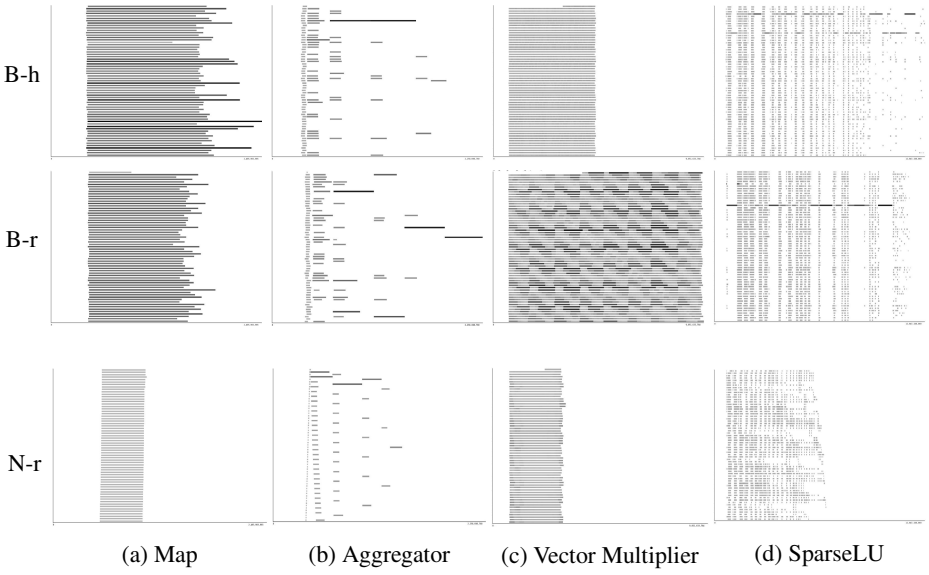



**Fig. 4.** Paraver views of study benchmark execution under different schedules

For the Map benchmark case, the views agreed with our motivation for home cache aware scheduling presented in Section 2.2. Views for both BFS-hashed and BFS-round schedules, which are home cache oblivious, showed long dark gray bars, indicating long task execution times due to large number of data cache stall cycles. On the other hand, the view for the home cache aware NUCAS-round schedule showed short light gray bars, indicating relatively smaller task execution times due to small number of data cache stall cycles. We reasoned about the performance of different schedules for the Aggregator benchmark similarly.

The BFS-round schedule performed poorly for the Vector Multiplier benchmark showing a large number of dark gray bars in its view. In comparison, views of both the BFS-hashed and NUCAS-round showed only light gray bars. However, the NUCAS-round schedule surpassed the performance of BFS-hashed schedule showing relatively lighter gray bars indicating that tasks suffered lower data cache stall cycles in comparison. In addition, vicinity-based task stealing improved execution time performance due to load-balancing over larger vicinity sizes.

The views of the SparseLU benchmark presented an interesting case. Both the BFS-round and BFS-hashed schedules showed that almost all tasks save a few experienced uniform data cache stall cycles and short execution times. However, those few tasks which incurred large data cache stall cycles (dark gray bars) increased the critical path of execution. In comparison, the NUCAS-round view showed that all tasks experienced uniform data cache style cycles the reason for which is the NUCAS heuristic which was able to optimize task execution for local access to the home cache of the most intensely accessed dependence. Vicinity-based stealing, which is oblivious to dependence access intensity, effectively removed the benefit of heuristic.

## 6 Related Work

Task and data affinity mechanisms discussed in our work are greatly inspired by the large body of research on NUMA optimizations for OpenMP runtime systems. The implicit memory allocation and architectural locality based scheduling mechanisms we implemented in the runtime system are inspired by a similar work on NUMA systems by Broquedis et al [7]. The memory allocation policies in our work are similar in principle to the *bind* and *cyclic* policies of the Minas memory allocator framework [8] for large scale NUMA machines. We cannot compare the techniques of our study with NUMA-based research because of the large difference in permissible scheduling overheads - NUCA penalties cost tens of cycles whereas NUMA penalties cost thousands of cycles. Our work improves on random task scheduling by reducing the impact of cache access penalties. We found a similar motivation in the SLAW [9] and the MTS [10] work-stealing schedulers for machines with hierarchical memory hierarchies. Both SLAW and MTS are designed to restrict off-chip work-stealing and execute parent and sibling tasks on the same multicore processor to utilize the state maintained in the central last-level cache. Our work relies on simple memory allocation policies to distribute task data on home caches during run-time. We found a more complicated counterpart in the work of Lu et al [11] where loops of affine programs are transformed at compile-time to minimize the NUCA impact of hash-for-home style of allocation manycore processors.

## 7 Conclusions

Manycore processors have hit the market and will grow larger in size and complexity in the coming years. In our work, we have shown that manycore processors demand careful architecture configuration and scheduling to achieve scalable parallel performance. Using the performance impact of home caches on the Tilera TILEPro64 processor as an example, we have shown how the runtime system of task based programming models can implicitly configure the manycore architecture for improved task scheduling performance. We believe that techniques similar to our memory allocation policy design, which essentially pushes application memory management as a runtime system responsibility, will rise in prominence as manycore processors become commonplace. Our home cache aware scheduling technique has shown significant execution time improvement in comparison to plain aggressive load-balancing schedules for selected study benchmarks. Reliance on the tuner, chunked application memory allocation, fixed steal

vicinity size, static memory allocation policy, unknown trigger points to recalculate the core-to-home-cache communication cost matrix and a limited set of study benchmarks constitute the currently being addressed limitations of our work. Scheduling is a game of overheads and with the arrival of manycore processors, we have shown that task-based runtime systems can and should begin thinking about what really goes on inside the processor.

**Acknowledgment.** The research leading to these results has received funding from the European Commission's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (`www.encore-project.eu`), grant agreement nr. 248647. The authors are members of the HiPEAC European network of Excellence (`http://www.hipeac.net`). The authors acknowledge great support in understanding Paraver from Xavi Aguilar at PDC Center For High Performance Computing at KTH, Xavier Teruel and Alejandro Rico Carro, both at BSC.

## References

1. Duran, A., Ayguad, E., Badia, R., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(2), 173 (2011)
2. Podobas, A., Brorsson, M.: A comparison of some recent task-based parallel programming models. In: Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers, MULTIPROG 2010, Pisa (January 2010)
3. Tilera: Tile processor user architecture manual, `http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf` (accessed June 14, 2012)
4. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: International Conference on Parallel Processing, ICPP 2009, pp. 124–131. IEEE (2009)
5. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: WoTUG-18, pp. 17–31 (1995)
6. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP Task Scheduling Strategies. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 100–110. Springer, Heidelberg (2008)
7. Broquedis, F., Furmento, N., Goglin, B., Namyst, R., Wacrenier, P.A.: Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 79–92. Springer, Heidelberg (2009)
8. Pousa Ribeiro, C., Méhaut, J.F.: Minas: Memory Affinity Management Framework. Research Report RR-7051, INRIA (2009)
9. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: Slaw: a scalable locality-aware adaptive work-stealing scheduler. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, IPDPS, pp. 1–12. IEEE (2010)
10. Olivier, S., Porterfield, A., Wheeler, K., Prins, J.: Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, pp. 49–56. ACM (2011)
11. Lu, Q., Alias, C., Bondhugula, U., Henretty, T., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P., Chen, Y., Lin, H., et al.: Data layout transformation for enhancing data locality on nuca chip multiprocessors. In: 18th International Conference on Parallel Architectures and Compilation Techniques, PACT 2009, pp. 348–357. IEEE (2009)