Design and Implementation Of A Cache Hierarchy-Aware Task Scheduling For Parallel Loops On Multicore Architectures

Nader Khammassi
Lab-STICC UMR CNRS 6285
ENSTA-Bretagne
29806 Brest Cedex 9, France
nader.khammassi@ensta-bretagne.fr

Jean-Christophe Le Lann
Lab-STICC UMR CNRS 6285
ENSTA-Bretagne
29806 Brest Cedex 9, France
jean-christophe.le_lann@ensta-bretagne.fr

***Abstract*— Effective cache utilization is critical to performance in chip-multiprocessor systems (CMP). Modern CMP architectures are based on hierarchical cache topology with varying private and shared caches configurations at different levels. Cache-aware scheduling has become a great design challenge. Many scheduling strategies have been designed to target specific cache configuration. In this paper we introduce a cache hierarchy-aware task scheduling (CHATS) algorithm which adapt to the underlying architecture and its cache topology. The proposed scheduling policy aims to improve cache performance by optimizing spatial and temporal data locality and reducing communication overhead without neglecting load balancing. CHATS has been implemented in the parallel loop construct of XPU framework introduced in previous works [1,7]. We compared CHATS to several popular scheduling policies including dynamic and static scheduling and task-stealing. Experimental results on synthetic and real workloads shows that our scheduling policy achieves up to 25% execution speed up compared to OpenMP, TBB and Cilk++ parallel loop implementations. We use our parallel loop implementation in two popular applications from the PARSEC benchmark suite and we compare it to the provided OpenMP, TBB and PThreads version on different architectures.**

***Index Terms*—Cache-aware Scheduling, Cache Locality, Parallel Loops, Multicore, Hierarchical Cache**

I. Introduction

Chip Multiprocessor (CMP) architectures are becoming widely available on many scales: from personal computers to embedded systems to high performance supercomputers... [17,18,19]. CMP cores count is growing continuously and their cache topologies are becoming increasingly hierarchical and deeper. Cache-aware scheduling has become a great design challenge in parallel programming for recent multicore architectures. Chip Multiprocessor (CMP) may exhibit different cache topologies with varying numbers of hierarchical shared and private caches at different levels. An effective task scheduling policy must take into account cache sharing not only at the SMT (Simultaneous Multi-Threading), CMP and SMP (Symetric Multi-Processor) levels but also at the different cache levels of a same chip.

Task scheduling is critical for execution efficiency especially in the case of parallel loops which are often a great performance multiplier. An efficient cache-aware scheduling policy for recent CMP should take into consideration three major parameters: spatial and temporal data locality in caches, communication and load-balancing. Hierarchical cache topology determines communication latencies between cores at the different levels of cache and thus, has a direct impact or these three critical scheduling factors. In this paper, we present a cache hierarchy-aware task scheduling (CHATS) policy which target to provide efficient hierarchical cache utilization without neglecting load-balancing in parallel loop implementation. CHATS consider spatial and temporal data locality (data reuse and communication) and load-balancing as the most critical parameters for delivering high performance and execution efficiency. We implemented this scheduling algorithm in the parallel loop construct “parallel_for” of the XPU framework and we compared it to parallel programming frameworks using different scheduling techniques. We used both synthetic workloads and real application from the PARSEC and Intel RMS Benchmarks [16].

In the next section, we gives an overview of several prior works on cache-aware scheduling, section 3 give a brief overview of some cache hierarchies used by modern general-purpose CMP and how they can be unified by an abstract model. Section 4 presents the design and implementation our cache-aware scheduling strategy. This scheduling policy is evaluated and compared to several popular scheduling strategies through an experiment using synthetic workloads in section 5. Section 6 shows benchmark results on two real applications: “Blackscholes” and “Fluid Animate” from the PARSEC Benchmark in comparison to different parallel prorgamming models.

II. Related Works

Traditional scheduling techniques such as dynamic scheduling [8] or task-stealing [9,10,11] make different tradeoffs between data locality and load-balancing but does not take into consideration cache hierarchy and communication latencies. Some prior works [20,15,5,14,13] target to design cache-aware scheduling policies which target to improve cache-utilization by focusing on one or more cache-related considerations. Processor-cache affinity scheduling [20] focused on temporal data locality and data reuse between threads. Thread clustering scheduler [15] detect sharing patterns between threads online using monitoring techniques and attempts to reduce cache-coherence overhead by clustering threads sharing same data onto close cores. CAB [5] aims to improve task stealing on hybrid SMP-CMP by reducing memory footprint and cache misses, It focus mainly on data sharing at the SMP level and try to reduce inter-socket communication. Constructive cache sharing [14] aims to reduce the memory footprint through exploiting potential overlap of shared data among co-scheduled threads. CATS [13] target to improve cache performance by considering data reuse, memory footprint and coherency misses. None of these prior works take into consideration the cache hierarchy of CMPs.

III. Overview Of CMP Architectures

Multicore processor employs a cache structure to simulate a fast common memory. This cache structure may display different cache sharing degrees at different levels. It is mainly composed of hierarchical private and shared caches. Figure 1 shows a set of CMP architectures from different vendors with varying cache configurations. For example, while the Intel Nehalem architecture associates a private L1 cache for each core, a private L2 cache and a shared L3 cache between all cores, Intel Dunnington architecture is uniformly hierarchical: a private L1 cache is associated to each core, an L2 cache is shared between each 2 cores and finally an common L3 cache is shared between all cores. The Sun UltraSPARC T2 Processor uses a private L1 cache for each core and a shared L2 caches between all cores.

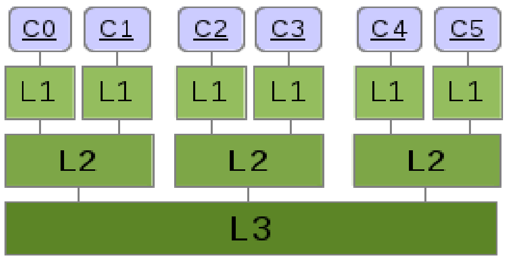

Fig. 1. Intel Dunnington

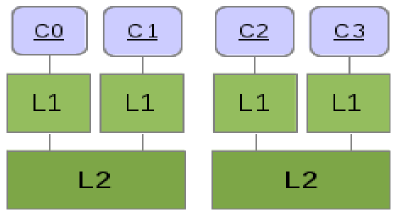

Fig. 2. Intel “Hapertown”

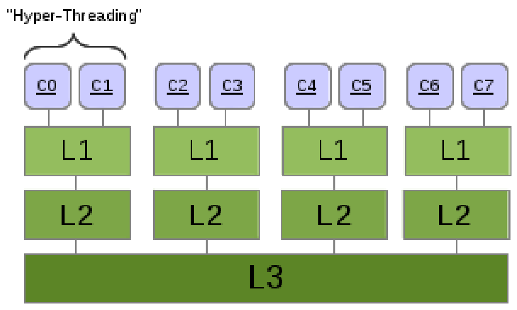

Fig. 3. Intel “Nehalem”

Cache level count and cache sharing degrees at each level are key information for our scheduling policy. The variation of sharing degree at different levels force programmer to make explicit and architecture-specific program optimization in order to get efficient execution. In order to be provide an efficient execution on various possible underlying architectures with different cache topology, a cache-aware scheduling algorithm should be dynamically adaptable to the target architecture. Consequently, such scheduling algorithm should have a detailed description of the cache topology of the underlying CMP. This description can be established through dynamic exploration of the target platform at the initialization of the run-time system. Modern operating systems provide means to obtains cache hierarchy details at high level either through system files such as in the Linux OS or through native API such as Windows [21].

Variation of the cache level count and cache sharing degrees raise the need to unify them under a single abstract description. The Unified Multicore Architecture Model (UMAM) [22] that can be used to provide a unified description for different CMP architectures. Memory hierarchy including cache levels and main shared memory can be described using UMAM as shown in Tab 1 which gives an example of three different platforms. The two first columns gives the cache-levels and cores count, the next columns gives the count of cores sharing $\boldsymbol{L_i}$ caches or $\boldsymbol{M_i}$ memory ($\boldsymbol{i} \in \boldsymbol{1..n}$, $\boldsymbol{n}$: Memory hierarchy levels count).

Architecture	Parameters					
	Cache Levels	*Cores*	L_1	L_2	L_3	M_4
Nehalem Core i7 Q920M	4	8	2	2	8	8
Nehalem 2 x Xeon E5620	4	8	2	2	8	16
Dunnington Xeon X7460	3	6	1	2	6	6

TABLE I. DESCRITION OF CACHE AND MEMORY HIERARCHY OF SOME CMP ARCHITECTURES IN UMAM

IV. CACHE HIERARCHY-AWARE TASK SCHEDULING

The design and implementation of CHATS rely a several basic building blocks which allows partitioning of the main work of a parallel loop into a set a little works which can be executed concurrently by several threads. We start by defining the components of our run-time system.

A. Runtime System

The run-time system is based on a worker (thread) pool able to execute tasks. Each worker have a FIFO (First-In-First-Out) work (task) queue. A scheduler can submit tasks to the workers through this work-queue. A worker remains idle until a task is pushed into its work queue so it wake up and execute its task. Submitting a task follows a one-to-one communication scheme between the main thread holding the scheduler and each worker to reduce communication overhead. Figure 4 gives and overview of the run-time system.

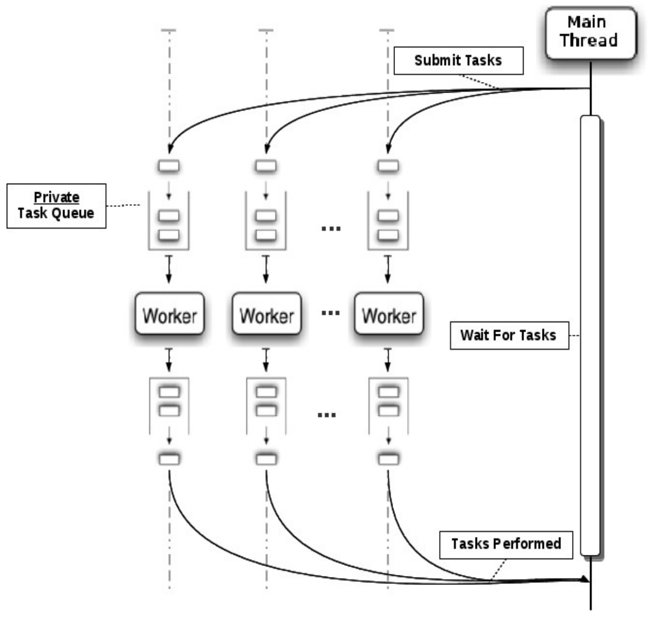

Fig. 4. Worker Pool-Based Run-Time System With Private Work-Queue

B. Work Unit

A work unit is a task which should be executed on a range of iteration then a range of shared iterations. In XPU "parallel_for" loop, a work unit is composed of:

- **Range** : Specify a range of iterations to process (min, max, progression step)
- **Shared Range** : As "Range", it specify a range of iteration, however, it allows "stealing" of iterations by concurrent threads.
- **Task** : The code which will process each iteration of a given range and/or shared range(s).

C. Data Partitioning

Data partitioning is a primordial step in parallelization of a loop. In our case we use basic a quasi-fair partitioning algorithm which decompose a "**Range**" into **N** "**Range**" and **M** "**Shared Range**". The algorithm ensure that the generated partitions are quasi-equals.

D. Parallel For Loop : Data Partitionning and Cache Topology

Let's consider a "for loop" **F** defined by : **i=0..n** by **1**

F corresponds to a "**Range**" which can be partitioned into **N** "**Range**" and **M** "**Shared Range**". Determining **M** and **N** depends directly on the underlying architecture:

N = Workers count ~ Cores count

M = Number of shared caches at all levels (consecutive cache levels shared by the same cores are considered as one)

P = **N**+**M** : Total partition count

Let's consider a Nehalem Intel Core i7 Q720 with 8 Hardware Threads and 4 Physical Cores (Fig.3). Data partitioning is described in Figure 5, so **P** is equal to **13** in this case. Green ranges are private "**Range**" so a worker doesn't share them with other workers. Orange box correspond to "**Shared Ranges**" which are shared among two co-scheduled SMT workers (threads) sharing **L1/L2** caches. Finally, red box is a "**Shared Range**" from which all workers can steal iterations, it corresponds to the L3 cache.

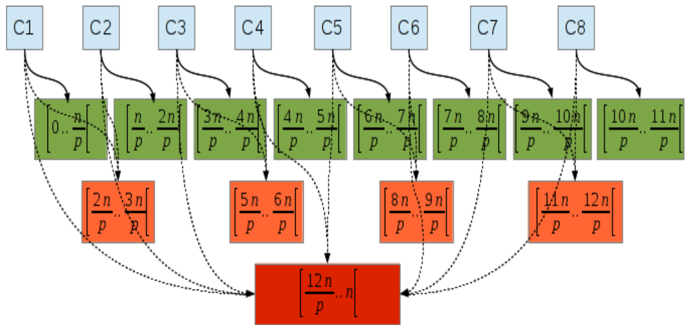
C1
C2
C3
C4
C5
C6
C7
C8
$[0..\frac{n}{p}[$
$[\frac{n}{p}..\frac{2n}{p}[$
$[\frac{3n}{p}..\frac{4n}{p}[$
$[\frac{4n}{p}..\frac{5n}{p}[$
$[\frac{6n}{p}..\frac{7n}{p}[$
$[\frac{7n}{p}..\frac{8n}{p}[$
$[\frac{9n}{p}..\frac{10n}{p}[$
$[\frac{10n}{p}..\frac{11n}{p}[$
$[\frac{2n}{p}..\frac{3n}{p}[$
$[\frac{5n}{p}..\frac{6n}{p}[$
$[\frac{8n}{p}..\frac{9n}{p}[$
$[\frac{11n}{p}..\frac{12n}{p}[$
$[\frac{12n}{p}..n[$

Fig. 5. CHATS Data Partitioning on an 8 Hardware Threads Intel Nehalem Processor

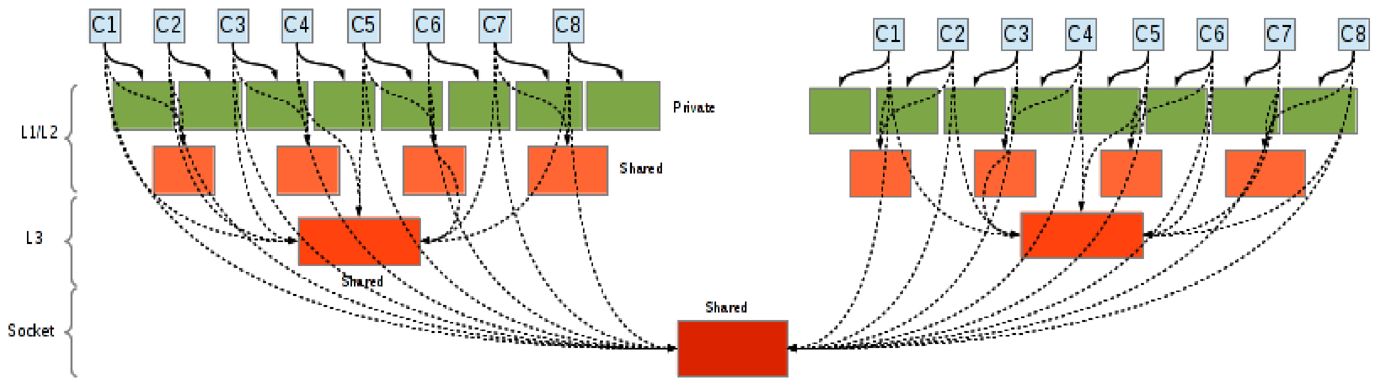

Fig. 6. CHATS Data Partitioning on an on hybrid SMT-CMP-SMP platform with 16 Hardware threads (2 x Intel Xeon E5620 at 2.4 GHz)

Figure 6 shows the data partitioning scheme for an SMP platform containing two Intel Nehalem Processors having eight hardware threads (4 Physical cores with Hyper-Threading enabled).

E. Workload Scheduling

Once data partitioned into "**Ranges**" and "**Shared Ranges**", we can submit works to our workers running on the different cores. Submitted work will specify a **Task**, a **Range** and one ore more **Shared Ranges**. If we take the partitioning pattern of the Figure 5. "***Worker 0***" running on "**Core 0"** will get a work containing:

- One "**Range**" : **[0 .. n/p[**
- Two "**Shared Ranges**" : **[2n/p .. 3n/p[** and **[12n/p .. n[**

Analogously, the other workers will gets their three ranges of iterations.

F. Execution Semantics

"Worker 0" will execute the task code on each iteration of its private range without any communication with the other threads. Once finished, it will try to steal iterations from the shared ranges if available. Iteration stealing requires communication (locking) between threads working on the same shared range. This communication overhead is reduced by the fact that threads communicates through shared caches. So, the communication introduced buy concurrent accesses to "Shared Range" **[12n/p .. n[** is more costly than the one introduced by concurrent accesses to **[2n/p .. 3n/p[.** However, we note that low level caches-associated "**Shared Range**" are fewer thant those associated to high level caches (1 associated to **L3** and 4 associated to **L1/L2**).

We outline that "**Shared Ranges**" aims to provide good load-balancing at the lower possible cost in term of communication overhead : when worker finish their work on private works, they does not remain idle, instead, they steal works from shared ranges or more precisely the "closed" shared range to their high level caches.

G. XPU: Implementation and programming Interface

The modified the default scheduler of XPU [7] to implement CHATS. Parallel for loop can be easily implemented using XPU. Figure 7 shows how to express data parallelism through a parallel for loop.

```
int process(int from, int to, int step, image* images) {
  for (int i=from; i<to; i+=step)  ...
}
void main()
{
  image * images = … ;
  task process_t(process, 0,0,0, images);
  task_group * pf;
  pf = new parallel_for(0, image_count, 1, &process_t);
  pf->run();
}
```

Fig. 7. An example of parallel for loop implementation using XPU

V. Performance Evaluation

We compare our CHATS implementation to several popular programming models implementing static scheduling, dynamic scheduling and task stealing which we present briefly:

1) Static scheduling

Static scheduling is the most straightforward scheduling technique: data is statically partitioned into N equal or pseudo-equal chunks, these chunks are then processed respectively by N parallel threads. This scheduling scheme avoid communication between threads, offer good data reuse when the parallel loop is executed several time. However, this method may result in load unbalancing, especially in the case of heavy workload, since faster threads remains idle, waiting for other threads until finishing their work.

2) *Dynamic scheduling*

Dynamic scheduling provide better load balancing since threads does not remains idle as long as chunks are available in the common work queue. Unfortunately, while improving workload distribution, this technique may introduces a costly communication between threads accessing concurrently to the common work queue (Many-To-Many Communication). This may results into ineffective uses of processor caches. Also, this technique provide poor data reuse since a same chunks may be processed by different threads on different cores when the parallel loop is executed multiple time. Bad data reuse may amplify consequently cache-miss rate.

3) *Task-stealing*

Task-stealing is a popular scheduling algorithm which is introduced in Cilk [10]. Task-stealing attempts to combines advantages of the two previous scheduling policies by making another trade-off between efficient cache utilization and load-balancing, In task stealing, each thread (worker) has a task pool in which its tasks are stored. Whenever a worker finishes its current task, the worker try to get another task from its task pool. If there's no more work (its task pool is empty), the worker select randomly a “victim” worker and try to steal a task from its task pool. If succeeded, it execute the stolen task, otherwise, it try to steal a task from another randomly-chosen worker [5]. Task-stealing performs good load-balancing since no thread (worker) remains idle as long as there is available “works”, i.e. ,available tasks in the task pools of workers. However, task-stealing may introduce significant communication overhead since “victim” threads are chosen randomly without considering cache-hierarchy or communication latencies. Deep cache hierarchies introduce non-uniform communications between cores, consequently, the choice of the “victim” thread becomes critical for performance: stealing a task from a “close” thread (sharing high-level cache with the stealer) is much cheaper than stealing a task from a “far” thread (running on a core which does not share any cache with the stealer).

B. Experiment : Parallel For On Synthetic Workloads

In order to evaluate the performances of our approach we designed an experiment which aims to evaluate cache utilization efficiency and global performance of a configurable target application. We generate a synthetic work load witch allow us to control unit workload and global workload and simulate data reuse. Thus, in order to achieve efficient execution, a scheduling strategy should provides good spatial and temporal data locality and an efficient load balancing.

The used unit workload is a sequential function performing a “quicksort” on a small vector. We control the unit workload by varying the size of this small vector. So, our input data is a set of small vector, our program perform a “quicksort” in each of these small vector. “quicksort” sort a vector performing multiple compare and swap so intensive intensive read/write accesses to data. This make it good candidate to evaluate efficient cache utilization.

In this experiment we try to evaluate the efficiency of our scheduling policy CHATS to: static scheduling, dynamic scheduling and task stealing. We run our synthetic workloads on an hybrid SMT-CMP-SMP platform with 16 Hardware threads (2 x Intel Xeon E5620 at 2.4 GHz) and we measure average execution time for different workload as well as cache-misses for each of the scheduling policies.

C. Results

As shown in Figure 8, results shows that XPU processes the heaviest workload about 20% faster than the second fastest candidate. We notes also that XPU become more efficient as the workload is bigger.

Figure 9 shows that XPU generate a low cache-miss rate in comparison to the other candidate. XPU cache-miss rate remains close to the static scheduling based candidate. Static scheduling is known to offer very good data locality and doesn't introduces communication overhead.

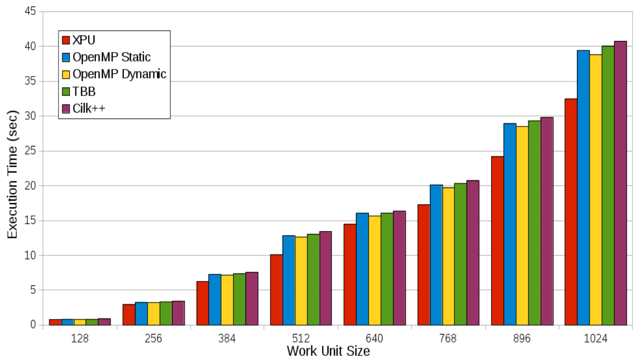

Fig. 8. Average processing time of different workload size.

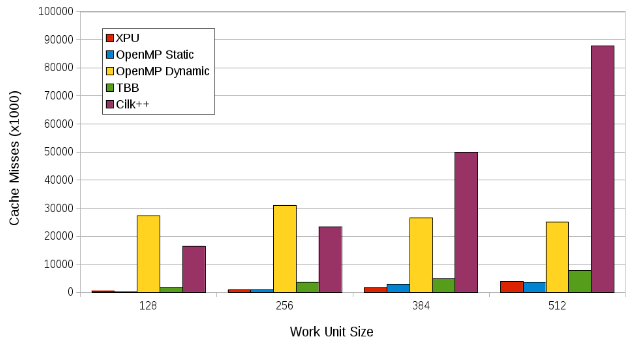

Fig. 9. Cache-miss rate for different problem size

VI. Applications

In order to evaluate performances of our scheduling policy, we use our parallel loop implementation to parallelize two popular applications from the PARSEC benchmark suite [16] which are also a part of the Intel RMS benchmark. The first application is the Blackscholes options pricing application and the second one is the fluid animation application.

For each application, the benchmark includes a serial version and several parallel versions using different parallel programming models including POSIX Threads [6], OpenMP [8] and Threading Building Blocks [12]. We use the serial version to build our parallel applications using the XPU framework [7]. More precisely, we use the “parallel_for” pattern to parallelize the main loop of both applications at thread level. We modified the default scheduler to implement our cache-aware scheduling algorithm.

The Blackscholes application exhibits massive parallelism thanks to its main parallel loop which process options with almost no communication between threads. The Fluid Animation workload processes large amount of particles through a parallel loop but suffer from significant inter-thread communication overhead.

A. Blackscholes

We parallelize the “Black-Scholes” workload at thread level using the XPU's "parallel_for" construct running on top of the cache hierarchy-aware scheduler. At the instruction level, we use the vectorization capability provided by XPU through a built-in vectorized type (vec4f) implemented in top of SSE to support SIMD. We used the sequential code of the “blackscholes” application as provided in PARSEC Benchmark Suite. The main processing loop is parallelized at the cost of 3 lines of extra-code. Vectorization is introduced simply by replacing regular float type by the XPU's "vec4f" vectorized type. We compare the performance achieved by our application to the five parallel versions provided in the same benchmark suite: OpenMP, TBB, Pthreads, OpenMP/SSE and POSIX Threads/SSE. We use the Intel C++ Compiler v12.0.5 and we executed our benchmark on different multicore platforms.

Fig. 10 and 11 shows the measured execution time for each parallel version. The XPU-based application provides higher performance than the other versions and execute up to 25% faster than POSIX Thread/SSE one. It takes advantage of the ability of the scheduler to provide both load-balancing, efficient cache utilization and low communication overhead to outperform the POSIX Thread version which use basic static scheduling achieving good cache utilization but poor load-balancing. The impact of this poor load-balancing issue becomes more visible as workload grows.

B. Fluid Animation

The fluid animation application is parallelized the same way as the Blackscholes one. Fig. 7 and 8 shows the measured execution time on two different platforms. The first one is an SMT-CMP processor which displays two cache level sharing. The second one is an hybrid SMT-CMP-SMP platform containing two Intel Nehalem processors and exhibiting three levels cache sharing.

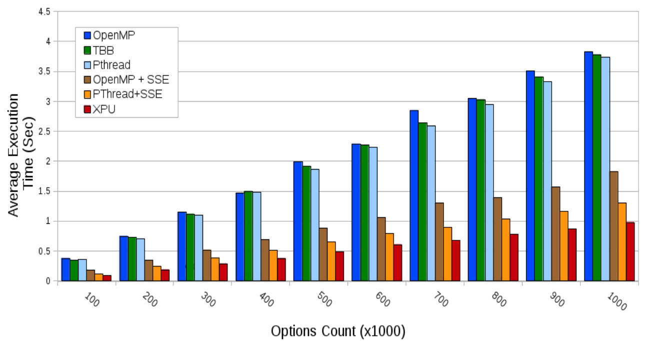

Fig. 10. Execution time of the “Blackscholes” application for different problem size on hybrid SMT-CMP Nehalem Processor with 8 Hardware threads (Intel Core i7 Q720)

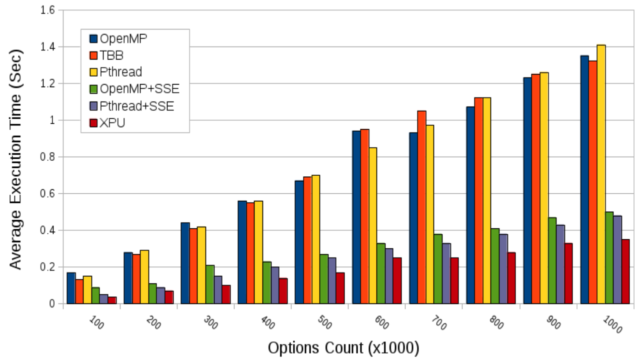

Fig. 11. Execution time of the “Blackscholes” application for different problem size on hybrid SMT-CMP-SMP platform with 16 Hardware threads (2 x Intel Xeon E5620 at 2.4 GHz)

The parallel “fluidanimate” program introduces significant communication between threads making spatial and temporal data locality in caches critical for achieving high performances. This gives an advantage to static scheduling techniques but doesn't reduces the impact of efficient load-balancing.

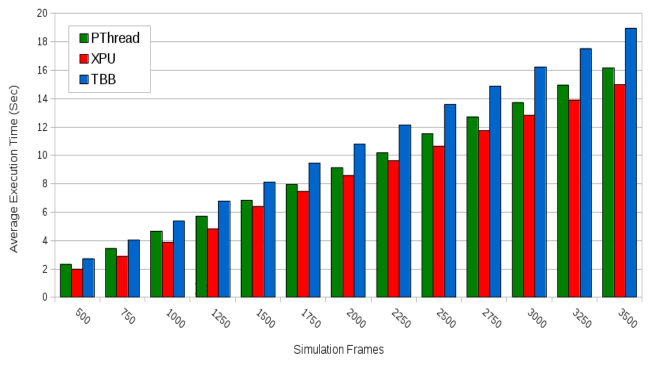

Fig. 12. Execution time of the “Fluid Animation” application for different problem sizes on hybrid SMT-CMP Nehalem Processor with 8 Hardware threads (Intel Core i7 Q720)

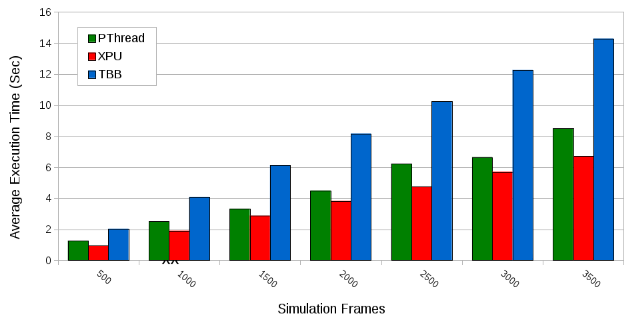

Fig. 13. Execution time of the “Fluid Animation” application for different problem sizes on hybrid SMT-CMP-SMP platform with 16 Hardware threads (2 x Intel Xeon E5620 at 2.4 GHz)

VII. Conclusion and Future Works

In this paper, we presented a cache-hierarchy aware scheduling which aims to provide efficient cache utilization without neglecting load-balancing. We described the CHATS scheduling policy and how it can improve spatial and temporal data locality in hierarchical caches. We shown how we can provide good load-balancing without generating significant communication overhead. Our experimental results on synthetic workloads outlined the high cache-misses rate introduced by some traditional scheduling policies implying arbitrary threads communications such as task-stealing or dynamic scheduling. These experiments demonstrated also that channelizing inter-thread communications through hierarchical sharing groups which communicates through shared caches reduces significantly this communication overhead and generates much lower cache-miss rate. Our implementation of CHATS algorithm in the XPU scheduler and its use on popular real applications from the PARSEC benchmark confirms our experimental results on synthetic workload and shows high performances in comparison to many others popular parallel programming models implementing different scheduling policies such OpenMP, POSIX Threads or Threading Building Blocks.

CHATS has been designed to adapt dynamically to the underlying CMP architecture by exploring its cache-topology at run-time. At the moment of writing this paper, dynamic cache topology exploration and CHATS scheduler have been developed separately. These two pieces will be put together to make CHATS adapt dynamically to various multicore architectures.

We can conclude that CHATS scheduling strategy may be good alternative to traditional scheduling techniques especially in the cases where spatial and temporal data locality in processor caches are critical for execution efficiency.

References

[1] N. Khammassi, J.C. Le Lann, J.P. Diguet and A. Skrzyniarz, “MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology”, HPCC '12 Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication, 71-80, Liverpool UK, June 2012

[2] G. Blake, R. G. Dreslinski and T. Mudge, “A Survey of Multicore Processors”, IEEE Signal Processing , vol. 26, n. 6, pp. 26-37 November 2009

[3] L. J. Karam, I. Alkamal, Alan Gatherer, G. A. Frantz, D. V. Anderson and B. L. Evans, “Trends in Multicore DSP platforms”, IEEE Signal Processing , vol. 26, n. 6, pp 38-49, November 2009

[4] W. Wolf, “Multiprocessor System-on-Chip Technology”, IEEE Signal Processing vol. 26, n. 6, November 2009

[5] Q. Chen, Z. Huang and M. Guo, “CAB: Cache Aware Bi-tier Task Stealing in Multi-socket Multi-core Architecture”, International Conference on Parallel Processing (ICPP), 2011

[6] D. Butenhof, Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[7] XPU Framework, “http://www.xpu-project.net/”

[8] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 3, pp. 404-418, 2009.

[9] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in 37th International Conference on Parallel Processing, pp. 536-545, IEEE, 2008.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, (Montreal, Quebec, Canada), pp. 212-223, ACM, June 1998.

[11] C. Leiserson, "The Cilk++ concurrency platform," in Proceedings of the 46th Annual Design Automation Conference, pp. 522-527, ACM, 2009.

[12] J. Reinders, Intel threading building blocks. O'Reilly, 2007.

[13] T. F. Yang, C. H. Lin and C. L. Yang, “Cache-Aware Task Scheduling on Multi-Core Architecture”, International Symposium on VLSI Design Automation and Test (VLSI-DAT), 2010

[14] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsas, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson., "Scheduling threads for constructive cache sharing on CMPs", In Proceedings of the 19th Annual Symposium on Parallel Algorithms and Architectures, pages 105–115. ACM, 2007.

[15] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors", In Proceedings of the 2nd SIGOPS/EuroSys European Conference on Computer Systems, pages 47–58. ACM, 2007.

[16] C. Bienia, S. Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, October 2008.

[17] G. Blake, R. G. Dreslinski and T. Mudge, "A Survey of Multicore Processors", IEEE Signal Processing , vol. 26, n. 6, pp. 26-37 November 2009

[18] L. J. Karam, I. Alkamal, Alan Gatherer, G. A. Frantz, D. V. Anderson and B. L. Evans, "Trends in Multicore DSP platforms", IEEE Signal Processing , vol. 26, n. 6, pp 38-49, November 2009

[19] W. Wolf, "Multiprocessor System-on-Chip Technology", IEEE Signal Processing vol. 26, n. 6, November 2009

[20] M. S. Squillante and E. D. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling", IEEE Transactions on Parallel and Distributed Systems, Vol 4, pp.131-143

[21] Processor information, "http://msdn.microsoft.com/en-us/library/windows/desktop/ms683194%28v=vs.85%29.aspx"

[22] J. E. Savage, M. Zubair, "A unified model for multicore architectures", IFMT '08 Proceedings of the 1st International Forum on Next-generation multicore/manycore Technologies