ARTM: A Lightweight Fork-Join Framework for Many-core Embedded Systems

Maroun Ojail, Raphael David, Yves Lhuillier, Alexandre Guerre
CEA, LIST, Embedded Computing Laboratory, F-91191 Gif-sur-Yvette, France.
Email: *name.surname*@cea.fr

***Abstract*—Embedded architectures are moving to multi-core and many-core concepts in order to sustain ever growing computing requirements within complexity and power budgets. Programming many-core architectures not only needs parallel programming skills, but also efficient exploitation of fine grain parallelism at both architecture and runtime levels. Scheduler reactivity is however increasingly important as tasks granularity is reduced, in order to keep the overhead of the scheduling to a minimum. This paper presents a lightweight fork-join framework for scheduling fine grain parallel tasks on embedded many-core systems. The asynchronous nature of the fork-join model used in this framework permits to dramatically decrease its scheduling overhead. Experimentation conducted in this paper show that the overhead induced by this framework is of 33 cycles per scheduled task. Also, we show that near-ideal speedup can be obtained by the ARTM framework for data parallel applications and that ARTM achieves better results than other state of the art parallelization techniques.**

I. Introduction

In order to address the constant demand for increasing performance within the complexity and power budgets of embedded systems, current computing architectures are composed of multiple cores. Many-core architectures intend to move away from the trend of increasing performance through complex micro-architectures that support instruction-level parallelism (ILP), and embracing designs with multiple, simple cores on a chip to exploit task-level parallelism (TLP) and data-level parallelism (DLP). This change in computer architectures enables processors to be clocked at a lower frequency and to consume less power, while still getting better overall performance.

Fine grain parallelism is required for certain applications since it eases the work of the developer being a form of parallelism which is naturally present in applications and does not require heavy algorithm rewriting. Since coarse grain threads can limit opportunities for exploiting parallelism for those applications, an efficient way to use the resources of multi-core platforms, is by exploiting the inherent fine grain parallelism. Scheduler reactivity is however increasingly important as tasks granularity is reduced, in order to keep the overhead of the scheduling to a minimum. Thus, solutions based on the creation of new threads for more parallelism are not feasible since they induce too large time overheads in case of fine-grain tasks.

However, the fork-join model of parallelism, already used for coarse grain threads, remains among the simplest and most effective design techniques for obtaining good parallel performance [1]. We thus propose in this paper a lightweight, Asynchronous Reactive Tasks Management (ARTM) framework, based on the fork-join model to exploit fine grain parallelism at the lowest possible cost.

The remaining of the paper is organized as follows. Section II first presents related work to parallel programming models for multi-core architectures. It also describes basics of the fork-join programming model as well as the authors previous work in this domain. Details of the ARTM framework are then presented in section III. Section IV then details the experimentations conducted to validate the ARTM framework and to compare it with state of the art parallelism extraction techniques. Finally, the key advantages of this proposition and our future work are summarized in section V.

II. Related work

For the applications that are intuitively divided into coarse-grained parallel sections, popular multithreaded programming interfaces, such as Pthreads [2] or OpenMP [3], are sufficient. An OpenMP program begins execution as a single process, called the master thread of execution which then creates multiple threads to be executed in parallel [4]. However, thread-based scheduling is heavyweight and operations of creating, destroying and yielding threads incur significant overheads, and thus must be used sparingly. Therefore, these approaches are not effective for fine-grain parallel applications.

In order to solve this high overhead problem, software-based lightweight multithreaded environments have been proposed. These environments include Split-C [5], Cilk [6] and Intel's Threading Building Blocks (TBB) [7]. These environments rely on coarse grain standard threading libraries, such as Pthreads, where fine grain parallel jobs (called procedures in Cilk and tasks in TBB) become work units in the work queues of the Pthreads. When the work queues of the Pthreads become unbalanced, these environments use work stealing to distribute the workload. Another software-based programming model, is Apple's Grand Central Dispatch (GCD) [8]. GCD is an asynchronous way to manage parallelism on platforms consisting of multiple cores. It implements multiple serial and parallel queues. When dispatching tasks to a parallel queue, the function does not wait to join and continues its execution normally, hence the asynchronous model. This type of model has the advantage of capturing complex task graphs. However, to our knowledge, none of these schedulers have been validated on very fine grain tasks (A couple of thousands

978-3-9815370-0-0/DATE13/©2013 EDAA

or even hundreds of cycles per task). More recently, Marongiu et al. [9] proposed a software-based lightweight support for fine grain parallelism that shows great potential with respect to state of the art techniques. However, tasks are statically assigned to processing cores which directly creates a limit in load balancing as we show in section IV.

A common feature for state of the art parallelization techniques like OpenMP and Cilk is the use of the fork-join model to express the parallelism in applications. The program starts on a single core, the master core. This master core forks a certain number of tasks that can be then scheduled by the other processors of the platform called slave processors. A slave core can, at its turn, declare itself as a master core to fork more tasks, creating by that nested levels of parallelism [4]. Fork and join operations can be either synchronous or asynchronous. In [10], we have presented a framework taking advantage of the synchronous fork-join programming model while being compatible with fine grain parallelism. Forked jobs are designated by the term task instead of thread in order to indicate that they are relatively lightweight jobs. In the synchronous model, the master core can only schedule the tasks that it has forked. When there are no more ready tasks to be scheduled, the master core waits until all tasks have finished their execution in order to join the tasks. Hence, it is the same processor that executes the fork and join operations. This model has the advantage of simplifying the programming and providing high reactivity. However, on top of the wasted time by the master processor while waiting to join the tasks, the synchronous fork-join model prevents conditional execution where some tasks can be separately joined and other tasks can be executed or not depending on a certain condition.

In this paper, the ARTM framework is based on the asynchronous fork-join programming model in order to eliminate these limitations and provide better load balancing among the multi-core resources. Asynchronous programming, as it will be shown in this paper, can provide better overall reactivity as well as programming flexibility. The description of ARTM framework is presented in the next section.

III. The ARTM framework

Asynchronous programming based on the fork-join model of parallelism consists in dissociating the fork and join operations. Hence, it is not mandatory to execute these two operations on the same processor. Therefore, the ARTM framework, based on the asynchronous model, presents, by construction, better load balancing than a synchronous one. This is achieved by not restricting the master processor to scheduling only tasks forked by itself. After forking tasks, the master processor can declare itself as a slave processor and schedule any ready task.

Moreover, this framework presents two additional functionalities: condition management and iteration management. Condition management allows for each of the forked tasks to be selectively activated depending on its condition of execution. Iteration management allows for the forked tasks to be executed a certain number of times before joining them. The conditions and the number of iterations can be dynamically set during the execution of an application in order to support data-dependent processing. Fig. 1 shows an example of a task graph based on the ARTM framework. In this figure, the master core forks 3 tasks `T0` to `T2`, then duplicates 2 times the task `T3` before going into slave mode. Tasks `T3` will execute `M` times before joining, while tasks `T0`, `T1` and `T2` will execute `N` times before joining. Task `T0` will only execute every other time and forks tasks `T4` and `T5` each time it is executed.

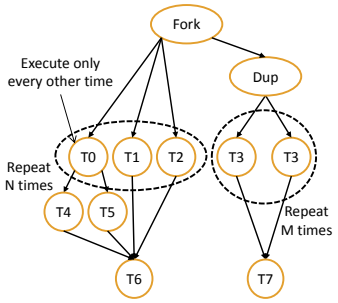

Fig. 1. Task graph example based on the ARTM framework

Programming with the ARTM framework is simple and its API mainly consists of two functions:

```
void fork(count, iter, cond, joiner,
                entrypoints, arg);
void dup(count, iter, cond, joiner,
                entrypoints, arg);
```

Both of the functions above take six arguments: the number of tasks to fork, the number of iterations to loop before joining, a pointer to the conditions of execution of the forked tasks, the joining task, a pointer to the tasks to fork and the argument to pass to these tasks. The only difference between these two functions is that the first forks tasks with different program codes thus extracting TLP from applications, whereas the second duplicates the same code a certain number of times taking advantage of the inherent DLP in target applications.

In the rest of this section we first detail the infrastructure of the ARTM framework and then we explain the fork and join operations via an example.

A. Implementation infrastructure

At boot time, each core is allocated a stack which will be used to process all the fine-grain tasks of the application. A main table of 32 elements is created, each element corresponds to a different fork/dup operation, and this, independently of the number of forked or duplicated tasks. Only one 32-bit variable is used to keep track of empty slots in this main table in order to keep the scheduling overhead to a minimum. An element structure contains ten fields:

- `entrypoint`: Pointer to the table of functions to fork or duplicate.
- `cond`: Pointer to the table of execute conditions for each task (`TRUE` or `FALSE`).
- `parent_id`: Parent ID in the table of the `tasks` structure.

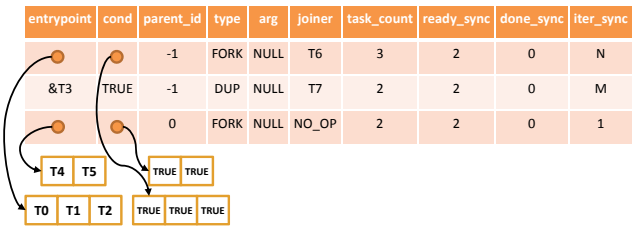

Fig. 2. Snapshot of the main ARTM table for the task graph in Fig. 1

- `type`: Type of operation (`FORK` or `DUP`).
- `arg`: Argument to pass to the forked or duplicated tasks.
- `joiner`: Task to execute after the all forked or duplicated tasks have finished their execution.
- `task_count`: Number of forked tasks or the number of times to duplicate the task.
- `ready_sync`: Synchronization variable which points to the next ready task to execute.
- `done_sync`: Synchronization variable which keeps track of the number of unfinished tasks.
- `iter_sync`: Synchronization variable which counts the number of remaining iterations before joining the tasks.

B. Forking and joining tasks

In Fig. 2, forking and joining tasks with the ARTM framework is schematically explained via an example. It shows a snapshot of the main table for the task graph depicted in Fig. 1 where only `T0` has begun execution. Three concurrent fork/dup operations are in the system. Since only one element of the main table is needed per fork/dup operation, scheduling weight is independent of the number of forked or duplicated tasks. The parent ID of the first 2 element is -1 indicating that these operations correspond to the first level of parallelism. Conditions can be changed at each iteration allowing by that the execution of `T0` only every other time.

The `ready_sync` variable of the first element is set to 2 instead of 3 because task `T0` has already begun execution and only 2 tasks remain ready. When `T0` begun execution, it called the `fork` function to fork tasks `T4` and `T5`. This implicated the creation of the third element in the main table. Forking time is independent of the number of the forked tasks since the `entrypoint` field is only a pointer to a table with the functions of tasks `T4` and `T5`. The joiner task of the third element is set to `NO_OP` as no operations are needed to be done when `T4` and `T5` join.

When the `done_sync` variable of an element is equal to the number of forked/duplicated tasks, and the iteration counts drops to 0, the ARTM framework is called to join the tasks by deleting the corresponding element from the main table and executing the `joiner` task.

IV. Experimentations on the ARTM Framework

This section presents the ARTM framework testing and validation on real applications, as well as comparisons to other parallelism extraction techniques.

TABLE I
ARTM framework overhead on a single core and comparison with the synchronous version

Operation	Asynchronous	Synchronous
Same level schedule	26	23
Diff. levels sched.	29	26
Fork/Dup	43	31
Join	40	22

As a first experiment, performance estimation of the code is done on a cycle accurate ISS of one STxP70-V4 processor of STMicroelectronics. We test the scheduling loop in two cases: scheduling tasks corresponding to the same fork/dup operation and scheduling tasks corresponding to different forks. In the first case, the overhead of the scheduling loop is 26 cycles while in the second case, this overhead increases to 29 cycles.

Concerning the fork/dup operations, the overhead is 35 cycles for the top operations (with a parent ID of -1) and 43 for the other operations. This difference of 8 cycles is due to an extra atomic post-increment operation at the parent level. As for the tasks joining, the simulations on the ISS show an overhead of 40 cycles per join operation.

Table I summarizes these results and presents a comparison with the synchronous version of the framework presented in [10]. This table shows that, according to simulations on a single core, the synchronous version of the framework presents better reactivity for all types of operations. However, as it will be shown next, the global scheduling overhead is reduced with the use of the ARTM framework.

The rest of this section is structured as follows. First, we present the target hardware platform on which the ARTM test and validation experiments are conducted. Then we perform experimentations on three different applications. Since in [10] we use the VC-1 decoder based on the SMPTE ST 421M standard [11] to evaluate the synchronous reactive tasks management technique, we reimplement the same application with the ARTM framework in order to compare to its synchronous predecessor [10]. Then, we use the Viola-Jones classification cascade [12], which inherently presents high DLP and a dynamic task graph, in order to evaluate the ARTM framework on highly dynamic applications and to compare its performances to OpenMP and Pthreads. Finally, the Strassen matrix multiplication algorithm is parallelized and implemented with ARTM. We use this algorithm to evaluate the effect of tasks granularity on the framework and to compare it to another state of the art lightweight parallelization extraction technique.

A. Target platform

ARTM targets cluster-based many-core architectures, where scaling to large systems is made possible by duplicating clusters of processors. We choose the STMicroelectronics heterogeneous low power many-core architecture (STHORM) formerly known as Platform 2012 or P2012 [13] as the architecture on which we conduct our tests. STHORM is a many-core computing fabric that is highly modular, as it is based on multiple clusters implemented with independent

TABLE II
ARCHITECTURAL PARAMETERS USED FOR ARTM TESTS

STxP70-V4 cores	1 to 16	L1 banks	32
$I\$_i$ size	16 kB	L1 size	256 kB
$I\$_i$ line	4 words	L3 latency	200 cycles
t_{hit}	1 cycle	L3 size	256 MB
AC number	128	AC size	32 bits

power and clock domains. The STxP70-V4 cores in a cluster are connected through a low-latency, multi-banked, high bandwidth logarithmic interconnect [14]. Communications between the cluster cores is ensured by a tightly coupled L1 data memory. Accesses to this memory are done with a two cycles latency. Concurrent accesses to different banks of the L1 memory is possible because the number of ports is equal to the number of banks. An L3 memory, shared by all clusters can also be accessed by all cores. Moreover, STHORM includes synchronization resources, called Atomic Counters (AC), which are memory-mapped registers that provide hardware semaphores [15]. For the tests presented in this paper, we use these AC for the ARTM synchronization variables presented in section III. In this paper, the ARTM framework is tested on a single STHORM cluster. Scheduling applications on multiple clusters can be achieved by coupling the ARTM-based intra-cluster scheduling, with coarse grain scheduling in order to dispatch work to the different clusters. Table II summarizes the architectural parameters of the STHORM instantiation on which the ARTM tests are conducted.

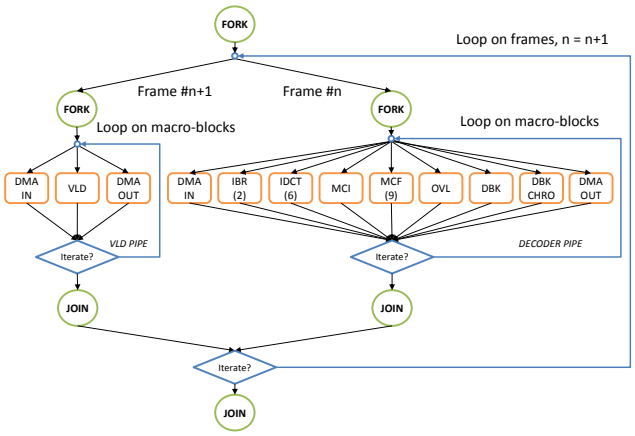

Fig. 3. VC-1 decoder structure with ARTM framework

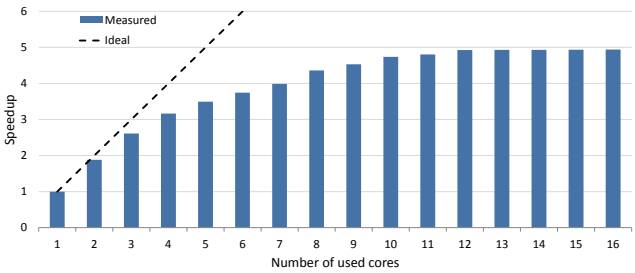

Fig. 4. Speedup of the VC-1 decoding application with respect to the number of used cores

B. VC-1 decoding application

In this section, we use the VC-1 decoder in order to compare the ARTM framework to its synchronous version presented in [10]. This application was designed to take advantage of fine grain parallelism. The program structure of this application is as follows: The main program forks two tasks. First, the VLD PIPE which forks 3 tasks. Second, the DECODER PIPE which forks 9 tasks. Also, IBR, IDCT and MCF are each duplicated a certain number of times. Each frame of the input stream is divided into macro-blocks. The two functional pipes are executed for all macro-blocks and all these operations are repeated for all the frames composing the video stream. Fig. 3 shows the task graph of this application with the ARTM framework. The ability of this framework to manage loop iterations and conditional execution is exploited to minimize the scheduling overhead by avoiding to invoke the fork and join operations for each iteration.

The forked tasks durations vary from a couple of tens of cycles for the smallest ones to 2148 cycles for the largest one with an average of 660 cycles per task. A video stream of 150 frames is chosen as input. On this stream, the average time spent in the ARTM framework per scheduled task is measured to 33 cycles. The synchronous version, tested in the same conditions, shows an average overhead of 58 cycles per scheduled task [10]. The ARTM version thus reduces the global scheduling overhead by 43%. This overhead reduction is mainly due to the fact that the ARTM framework allows the internal management of the loop iterations. For example, in the VLD pipeline in Fig. 3, only one fork and one join operations are invoked per frame. The synchronous version lacks such management and fork and join operations are necessary for each processed macro-block. Moreover, due to the support of conditional execution, the functions in the pipe are activated only when they are needed, avoiding by that unnecessary function calls.

Fig. 4 shows simulation results of the VC-1 decoding application on STHORM with 1 to 16 STxP70 cores. It can be seen that, for a small number of cores, good speedups can be obtained (for example 3.2× for 4 cores). However, data-level parallelism is very limited in this application because of high data-dependency in the processing pipelines. Thus, the tests on a higher number of processing cores can not be conclusive.

C. Viola-Jones classification cascade

In this section, we use a dynamic application, depicting high DLP to evaluate the effectiveness of the ARTM framework for scheduling data-dependant tasks. The implementation of the Viola-Jones classification cascade used in this work is part of a pedestrian detection application. The classification takes as input 10 integral images calculated beforehand from a VGA image (640×480 pixels). It also takes as input multiple boxes or Regions Of Interest (ROIs). For each one of the boxes, a 10 stages loop is executed in order to determine if this ROI contains a pedestrian or not. A positive box (one that actually

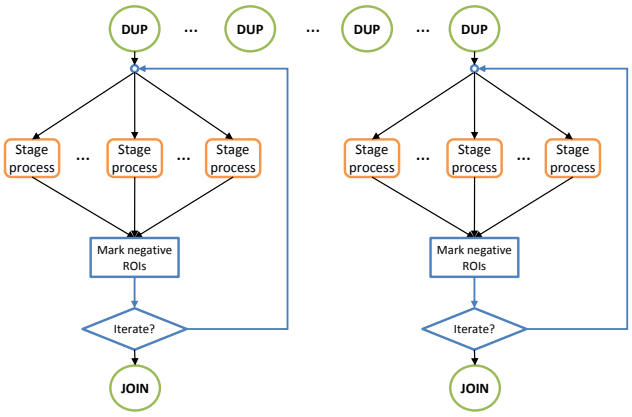

Fig. 5. Classification cascade structure with the ARTM framework

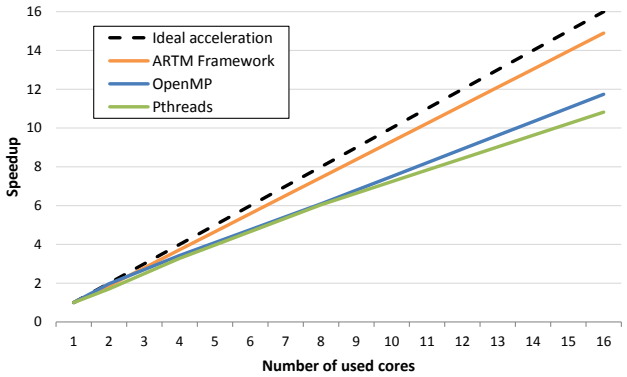

Fig. 6. Classification cascade speedup with ARTM, OpenMP and Pthreads

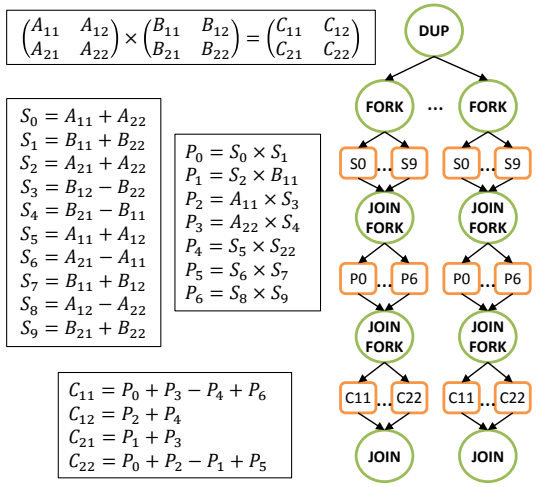

Fig. 7. Strassen algorithm as depicted in [9] and its task graph with ARTM framework

contains a pedestrian) has to pass all 10 stages in order to be identified as such. Whereas a negative box can be rejected at any of the 10 stages. And since the processing complexity highly augments from one stage to the next, the classification part is hence dynamic in a sense that the processing power needed for each box is different and depends on the input data. The processing complexity for each box, although varying, remains low with an average of 2000 cycles per box. But the overall computational power needed for the classification part is very high since 1 million and 600 thousands boxes have to be tested for each image.

Fig. 5 presents the structure of the parallelized classification using the ARTM framework. The resulting task graph is highly dynamic since the processing time for each ROI is not the same and depends on the input image. The core function that has to be applied to all ROIs is the stage processing function. Due to the asynchronous property of the ARTM framework, the duplication functions are non-blocking. Hence, at the beginning of the application, one core performs a parallel duplication of the stage processing function and each one duplicates the processing on 16 different boxes. This results in data-parallel fine grain tasks, each targeting a different ROI. The ARTM internal conditional management is used to mark negative ROIs, preventing further processing on them.

Multiple simulations were conducted on a VGA image with up to 16 STxP70 cores. Fig. 6 shows the speedup due to the parallelization with respect to the mono-processor version. The application runs nearly 15 times faster on 16 cores. This near-ideal (only 6.25% decrease) speedup is due to the excellent load balancing that the ARTM framework provides since is does ensure that no core is idle when there is tasks to execute, even for a dynamic task graph like in this case.

Also, and in order to compare the ARTM framework to other parallelization techniques, the classification cascade is parallelized and implemented using the OpenMP API and standard Pthreads. Because OpenMP is not available on STHORM, this implementation was done on a AMD Opteron(TM) Processor 6276 [16] which contains 16 processing cores. For each implementation, the speedup is calculated with respect to the mono-processor implementation on the corresponding platform. Fig. 6 shows the speedup for the different implementations. The ARTM framework presents the highest speedup. OpenMP and Pthreads based implementations achieve lesser speedup due to, first the high overhead they present compared to the ARTM framework, and second the limited load balancing they can achieve in the case of a dynamic application.

D. Strassen matrix multiplication

In this section, we test the effect of tasks granularity on the scheduling performance of the ARTM framework and compare it to the lightweight scheduling approach presented in [9]. The Strassen algorithm was used in [9] on an architecture with the same properties as STHORM. We thus implemented the same algorithm for our tests.

Fig. 7 shows this algorithm with its task graph using the ARTM framework. The input matrices A and B are decomposed in four sub-matrices, which can be processed in parallel. Sub-matrices undergo a number of sums/substractions and multiplications. All these operations are fully data-parallel. Tasks are duplicated in order to take advantage of DLP. Further details on the Strassen algorithm can be found in [9].

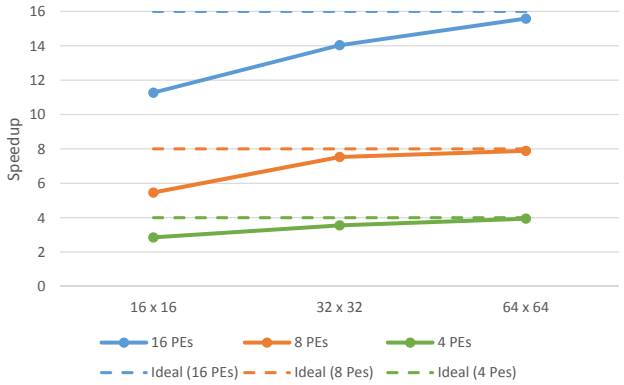

Fig. 8. Strassen algorithm speedup with ARTM framework

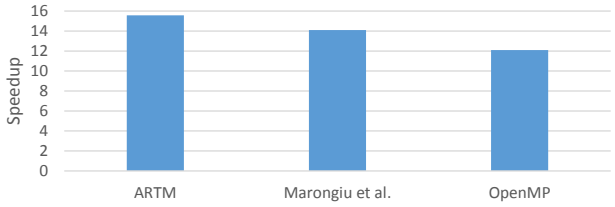

Fig. 9. Strassen algorithm speedup comparison for 64×64 matrices on 16 cores (ARTM vs. prior art in [9])

Experimentations are done for 16×16 (average of 225 cycles per task), 32×32 (average of 1585 cycles per task) and 64×64 (average of 11800 cycles per task) matrices. For each matrix size, 4, 8 and 16 cores are used to process the algorithm. Fig. 8 summarizes the speedups obtained with respect to a sequential execution on a single core. The effect of task granularity is clear where the overhead of the ARTM framework gets bigger for very fine-grain tasks.

In order to compare the ARTM framework with the scheduling technique presented in [9], we consider the case of 64×64 matrices on 16 cores because this is the only result shown in their work for the whole application. As seen in Fig. 9, the ARTM framework achieves better speedup than the scheduler in [9] as well as the OpenMP based implementation presented in the same paper. In fact, not only the ARTM scheduler is 13% faster than the solution of [9], it also present high dynamicity. The authors in [9] statically assign the tasks to the processing cores and thus they get a theoretical limitation of nearly 14× on 16 cores. The ARTM framework dynamically assigns the tasks to free cores achieving by that the best possible load balancing and thus a better speedup (15.8×) while also providing simplified programming.

V. Conclusion and Future Work

This paper presented a lightweight reactive fine grain tasks management framework for homogeneous multi-core architectures. It incorporates important functionality like conditional execution and loop management which allows it to support a wide range of task graphs and to lower its scheduling overhead. This overhead is measured to 26 cycles per task, and to 83 cycles per a couple of fork and join operations. Simulations showed only a 6.25% decrease of the speedup with respect to the ideal one. Moreover, near-ideal speedup was achieved on a dynamic application, the Viola-Jones classification cascade. Comparison with state of the art parallelism extraction techniques showed a significant advantage of the ARTM framework in obtaining the best possible load balancing and speedup. Coupling the ARTM framework with coarse grain parallelization techniques will be investigated as future work in order to extend the tests on multiple clusters. Coarse grain jobs will be dispatched to clusters and the ARTM framework can schedule tasks on the cores inside the clusters.

Acknowledgment

This work is partly supported by the European cooperative CATRENE project CA104 COBRA.

References

[1] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00. New York, NY, USA: ACM, 2000, pp. 36–43.

[2] D. Butenhof, *Programming with POSIX Threads*. Addison-Wesley, 1997.

[3] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, January 1998.

[4] M. Sato, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in *Proceedings of the 15th international symposium on System Synthesis*, ser. ISSS '02. New York, NY, USA: ACM, 2002, pp. 109–111.

[5] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in split-c," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 262–273.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, pp. 212–223, May 1998.

[7] C. Pheatt, "Intel® threading building blocks," *J. Comput. Small Coll.*, vol. 23, pp. 298–298, April 2008.

[8] Apple Inc. (2010, May) Grand Central Dispatch (GCD) Reference.

[9] A. Marongiu, P. Burgio, and L. Benini, "Fast and lightweight support for nested parallelism on cluster-based embedded many-cores," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 105 –110.

[10] M. Ojail, R. David, K. Ben Chehida, Y. Lhuillier, and L. Benini, "Synchronous reactive fine grain tasks management for homogeneous many-core architectures," *ARCS 2011*, 2011.

[11] SMPTE, "SMPTE ST 421M: VC-1 Compressed Video Bitstream Format and Decoding Process," www.smpte.org, March 2011.

[12] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, 2001, pp. I–511 – I–518 vol.1.

[13] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 983 –987.

[14] A. Rahimi, I. Loi, M. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1 –6.

[15] F. Thabet, Y. Lhuillier, C. Andriamisaina, J.-M. Philippe, and R. David, "An Efficient and Flexible Hardware Support for Accelerating Synchronization Operations on the STHORM Many-Core Architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, march 2013.

[16] AMD, "The new AMD Opteron(TM) 6200 Series processor," http://server.amd.com/LP=237, 2012.