An Optimized Task-Based Runtime System for Resource-Constrained Parallel Accelerators

Daniele Cesarini†, Andrea Marongiu†‡, and Luca Benini†‡
†DEI, University of Bologna, 40136 Bologna, Italy
‡IIS, Swiss Federal Institute of Technology, 8092 Zurich, Switzerland
daniele.cesarini@unibo.it, {a.marongiu, luca.benini}@iis.ee.ethz.ch

***Abstract*—Manycore accelerators have recently proven a promising solution for increasingly powerful and energy efficient computing systems. This raises the need for parallel programming models capable of effectively leveraging hundreds to thousands of processors. Task-based parallelism has the potential to provide such capabilities, offering flexible support to fine-grained and irregular parallelism. However, efficiently supporting this programming paradigm on resource-constrained parallel accelerators is a challenging task. In this paper, we present an optimized implementation of the OpenMP tasking model for embedded parallel accelerators, discussing the key design solution that guarantee small memory (footprint) and minimize performance overheads. We validate our design by comparing to several state-of-the-art tasking implementations, using the most representative parallelization patterns. The experimental results confirm that our solution achieves near-ideal speedups for tasks as small as 5K cycles.**

I. Introduction

Heterogeneous systems, pioneered in the High-Performance Computing (HPC) domain by General-Purpose graphic processing units (GPGPU), are nowadays adopted virtually at every scale due to the important benefits they brings in terms of energy efficiency. Multi-processor on-chip systems (MPSoC) are increasingly adopting heterogeneous designs where a general-purpose host processor is coupled to programmable many-core accelerators, where highly-parallel computation kernels of an application can be offloaded to improve overall performance/watt.

This clearly complicates application development and raises the need for parallel programming models capable of effectively leveraging hundreds to thousands of processors. As the complexity of software increases, it is widely acknowledged that totally laying the burden of handling performance scalability issues on the programmers is unfeasible. Application designers should focus on outlining available parallelism in an application, while efficient distribution of parallel tasks on a manycore should be controlled by system software libraries and runtime environments (RTE).

Task-based parallelism has the potential to provide such features, as it provides a powerful conceptual framework to exploit irregular parallelism in target applications. Several works have demonstrated the effectiveness of tasking in the HPC domain. However, a space- and performance-efficient design of a tasking RTE targeting MPSoCs is a challenging task, as embedded parallel applications typically exhibit very fine-grained parallelism. The applicability of the tasking approach to embedded applications and embedded manycore accelerators is often limited to coarse-grained parallel tasks, capable of tolerating the high overheads typically implied in a tasking RTE. State-of-the-art tasking RTEs for embedded manycores [1] [2] succeed in achieving low overheads and enabling high speedups for very fine-grained tasks, but only for simple flat parallel patterns (where all the tasks are created from the same parent task). The reason for this limitation lies in a key design choice: only *tied* tasks are supported. If a *tied* task is suspended (due to synchronization, creation of another task, etc.) only the thread that initially owned it is allowed to resume its execution. This clearly limits significantly the available parallelism when more sophisticated (and realistic) parallel execution patterns are considered, like nested tasking (found, for example, in programs that use recursion). Another limitation that follows from this design choice is the restricted set of scheduling policies available. *Breadth-first scheduling* (BFS) and *Work-first scheduling* (WFS) are the two most widely used policies for distributing tasks among available threads. When *tied* tasks are used, BFS is the only choice in practice, as WFS leads to a complete sequentialization of task executions when nested parallelism is adopted.

In this work we build upon the most lightweight tasking RTE design for embedded manycores [1] and extend it to support *untied* task. When suspended, *untied* tasks can be resumed by any available thread, thus significantly increasing the potential for parallelism exploitation. On top of this extended RTE we implement support for WFS and associated *cutoff* policies. Supporting *untied* tasks required major modifications to the RTE and potentially heavyweight ones, as we replace a simple BFS loop based on function calls with more sophisticated mechanisms for task context switching among multiple threads.

Our experimental results show that:

1) The careful design of the extensions allows us to achieve nearly identical speedup results to [1] for flat parallel patterns, enabling efficient support for very fine-grained tasking (almost ideal speedups for tasks beyond 5K cycles);
2) WFS enables significantly higher speedups (up to 60%) than BFS when *untied* tasks are used in recursive patterns;
3) Cutoff policies on top of the provided support for *untied* tasks allow to achieve nearly-ideal speedups for recursive patterns for tasks of the same small size of the flat pattern (around 5K cycles).

In addition, we compare our RTE design to several others and demonstrate that our solution is one order of magnitude more efficient than the second best, in terms of task granularity for which nearly-ideal speedups are achieved.

II. Related work

Tasking has been successfully used in the high performance computing (HPC) domain to parallelize complex algorithms leveraging sophisticated control structures. Notable examples of task-based programming models are Cilk [4], Intel TBB [12], Wool [11], Apple GCD [5] and the current OpenMP specification [7]. Researchers have actively explored the effectiveness of OpenMP tasks in the context of HPC applications and systems [8] [16] [15] [21] [20]. While OpenMP has recently gained much attention also in the embedded domain [2] [17] [18], not much work has been done on demonstrating the benefits of tasking for fine-grained embedded workloads or for proposing lightweight and efficient tasking implementations for embedded MPSoCs. Kumar et al. [6] present an architecture for efficient execution of fine-grained tasks, demonstrating

978-3-9815370-0-0/DATE16/ © 2016 EDAA

the importance of reducing tasking support overheads for the model to be beneficial in embedded multicores. Burgio et al. [1] present an efficient implementation of OpenMP v3.0 for an embedded shared memory multicore cluster, and the one with the lowest overheads among the available ones in literature targeting embedded MPSoCs. We thus choose this implementation as a baseline for our work. The main limitation of this work (like all other implementations targeting embedded systems) is the lack of support for *untied* tasks and nested parallel patterns, which are the ones for which task-based parallelism is most beneficial. Our work addresses these shortcomings and proposes a lightweight tasking runtime capable of enabling near-ideal speedups for recursive parallel patterns employing very fine-grained tasks.

III. Target Architecture

In this section we describe the *cluster*-based architecture template targeted in this paper. *Clusters* are the central building block of several recent embedded many-cores. These products consider a hierarchical design, where simple processing units (PU) are grouped into small-medium sized subsystems (the *clusters*) sharing high-performance local interconnection and memory. Scaling to larger system sizes is enabled by replicating *clusters* and interconnecting them with a scalable medium like a NoC.

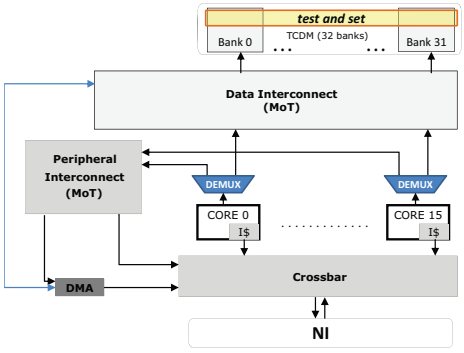

Fig. 1. On-chip shared memory cluster

The simplified block diagram of the target *cluster* is shown in Figure 1. It contains 16 RISC32 processor cores, each featuring a private instruction cache. Processors communicate through a multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). This shared L1 TCDM is implemented as explicitly managed SRAM banks (i.e., scratchpad memory), to which processors are interconnected through a low-latency, high-bandwidth data interconnect enabling 1-cycle L1 accesses. This is compatible with pipeline depth for load/store for most processors, hence it can be executed in TCDM without stalls – in absence of conflicts. Note that the interconnection supports up to 16 concurrent processor-to-memory transactions within a single clock cycle, given that the target addresses belong to different banks (one port per bank). Multiple concurrent reads at the same address happen in the same clock cycle (broadcast). A true data conflict takes place only when multiple processors try to access different addresses within the same bank. In this case the requests are sequentialized on the single bank port. To minimize the probability of conflicts i) the interconnection implements address interleaving at the word-level; ii) the number of banks can be configured to be an integer multiple M of the number of cores (in our setup M=1 never leads to a noticeable number of conflicts).

Processors can synchronize by means of standard read/write operations within an *aliased* TCDM address range. Specifically, adding a constant offset to any TCDM address provides *test-and-set* semantics via the interconnect (a single atomic operation returns the content of the target memory location and updates it).

This architectural template captures the key traits of existing cluster-based many-cores such as STMicroelectronics STHORM [3] or Kalray MPPA [13] in terms of core organization, number of clusters, interconnection system and memory hierarchy. As a concrete instance of this template we built a cycle-accurate SystemC simulator, based on the *VirtualSoC* virtual platform [14]. *VirtualSoC* is a prototyping framework targeting the full-system simulation of massively parallel heterogeneous SoCs. The topology that we consider in this paper consists of a single cluster, plus a memory controller to the off-chip main memory.

IV. Background

In this section we provide background information related to the OpenMP tasking model and the baseline implementation on top of which we build our work.

A. Basic Notions of OpenMP Tasking

OpenMP historically relied on a *fork/join* (FJ) parallel execution model. The program starts with a single thread of execution (the *master*); when a `parallel` construct is encountered, $n-1$ new threads (n being specified with the `num_threads` clause) are recruited into a parallel *team*. Several *worksharing* constructs are provided to specify how the parallel workload is distributed among threads. Since the specification version 3.0, on top of the FJ model OpenMP provides support for task-based parallelism, which is our focus.

When a thread encounters a `task` construct, a new *task region* is generated from the code contained within the task. Additional *data-sharing* clauses specify an associated data environment, while the execution of the new task can be assigned to one of the threads in the team, based on additional *task-scheduling* clauses that specify i) dependences among tasks; ii) (conditional) immediate or deferred execution; iii) task type, between `tied` and `untied` (to the thread that first encounters them).

Tied tasks are the default in OpenMP, as they attempt to establish a trade-off between ease of programming and scheduling flexibility (and thus, performance) [9]. If a *tied* task is suspended, it can later only be resumed by the same thread that originally started it. `Untied` tasks are not bound to any thread and so in case they are suspended they can later be resumed by any thread in the team. Using `untied` tasks has the potential for significantly increasing the achievable parallelism, but comes at the cost of a higher programming effort (the programmer is responsible for avoiding issues such as deadlock, thread-private memory, etc.).

All tasks bound to a given parallel region are guaranteed to have completed at the implicit barrier at the end of the parallel region, as well as at any other explicit `barrier` construct. Synchronization over a subset of explicit tasks can be specified with the `taskwait` construct, which forces the encountering task to wait for all its first-level descendants to complete before proceeding.

OpenMP defines *task scheduling points* (TSP) in a program, where the encountering task can be suspended and the hosting thread can be rescheduled to a different task. TSPs occur upon (1) task creation and completion, (2) task synchronization points such as `taskwait`, (3) thread synchronization points such as explicit and implicit barriers. When a thread encounters a TSP it can begin the execution of a new task, or resume a previously suspended one, provided that a set of *task scheduling constraints* (TSC) are fulfilled. Among TSCs, one is particularly relevant to this work, as it limits the flexibility of *tied* task scheduling. This TSC recites: *In order to start the*

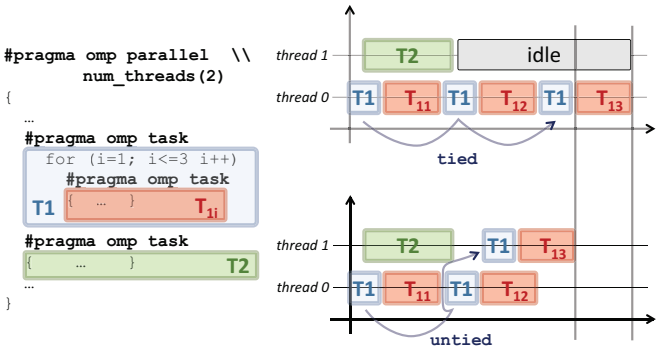

Fig. 2. Example OpenMP program. Tied and untied task scheduling.

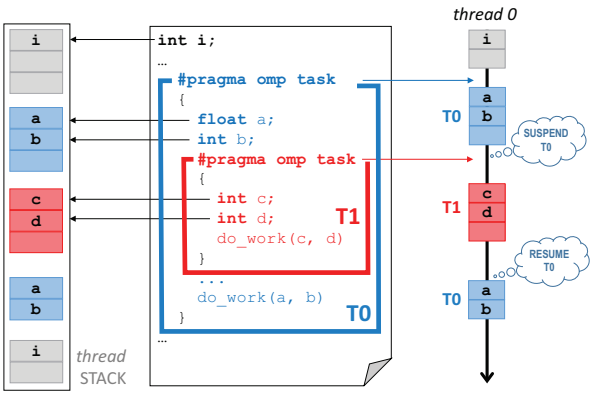

Fig. 3. *tied* task suspension in the baseline implementation [1].

execution of a new tied *task, the new task must be a descendant of every suspended task tied to the same thread [...]*[1].

B. Task schedulers

The two most widespread scheduling approaches for task-based programming models are *Breadth-first scheduling* (BFS) and *Work-first scheduling* (WFS). Upon encountering a task creation point: i) BFS will push the new task in a queue and continue execution of the parent task; ii) WFS will suspend the parent task and start execution of the new task. BFS tends to be more demanding in terms of memory, as it creates all tasks before starting their execution (and thus all tasks coexist simultaneously). This is an undesirable property – in general – and in particular for the resource-constrained systems that we target in this work, which makes WFS a better candidate. WFS also has the nice property of following the execution path of the original sequential program, which tends to result in better data locality [16].

However, since *tied* tasks are the default in OpenMP, RTE implementations typically use BFS. Figure 2 shows the behaviour of WFS if used in combination with *tied* and *untied* tasks. If all the tasks are generated from a parent task *T*_0, *untied* tasks will be distributed among threads in a balanced manner thanks to the capability of the system to resume a suspended task on a different thread. If *tied* tasks are used, at each creation point the parent task will be suspended and the hosting thread will be rescheduled to execute the child task. The suspended parent, however, cannot be resumed on a different thread, which will lead to a sequential execution.

C. Baseline Implementation

The baseline implementation [1] is based on a centralized queue. The authors focus on lightweight support for *push* and *pop* operations on the centralized queue (upon task creation and extraction, respectively), that relies on fine-grained locking mechanisms. TSPs are implemented using lightweight events rather than active polling, which avoids the massive contention implied by active polling. More specifically, idle threads on the TSP are put into sleep mode. When a task is created (i.e., *push*ed in the queue) the creator thread sends a signal which wakes up a single thread (selected using *round-robin*). After completing the task execution, the thread returns into sleeping mode.

The above described queue is implemented with a doubly-linked-list. This data structure allows to *push* and *pop* tasks from the queue and also remove a task in any position of the queue. This is key for low overhead, as tasks are not constrained to execute in-order (except when dependencies are specified), so their completion and removal from the queue is independent of their position. Note that a simple linked list doesn't allow this operation.

While this implementation shows excellent performance in presence of simple *flat* parallel patterns, where all the tasks are created from within a single level (i.e., a single parent task), it is not capable of supporting more sophisticated forms of parallelism, like nested parallel patterns found in programs that use recursion, and for which the tasking model was originally proposed. Consequently, *untied* tasks are not supported by this implementation. Due to the limitations of *tied* tasks described previously, the scheduling policy relies on BSC, and WFS is not supported.

In the following we describe how we extend this implementation to fully support nested parallel patterns and *untied* tasks, while keeping the implementation lightweight and not too memory-hungry. These both are key requirements for any implementation suitable for embedded MPSoCs, and our main goal is to achieve comparable efficiency in terms of task granularity for which near-ideal speedups are achieved.

V. Runtime Design

Figure 3 shows how task suspension works in most implementations supporting *tied* tasks (WFS is assumed, but the behavior is the same under BFS). The thread on which the code shown in figure is executing has an associated stack (depicted on the left). When a `task` directive is encountered the thread jumps to a runtime function that manages the creation of a new task from the enclosed code region. A new stack frame is activated for this task, like in every regular function call. The same thing happens at every nested `task` directive. When a task is completed, the stack pointer is reset to the top of the previous active frame. Since the semantics of *tied* task scheduling ensure that suspension/resumption can only happen on the same thread, no explicit bookkeeping to save/restore the context of a task is required.

The key extension required to support *untied* tasks is the capability of allowing to resume a *suspended* task on a different thread than the one that started and *suspended* it. To achieve this goal we rely on lightweight co-routines [10]. Co-routines rely on cooperative tasks which publicly expose their code and memory state (register file, stack), so that different threads can take control of the execution after restoring the memory state. Every time that a thread suspends or resumes a suspended cooperative task a context switch is performed. We place the required metadata to support task contexts (TC) in the shared TCDM, which ensures fast context switch (any thread can access the shared stacks with the same latency of just 1 cycle) and we use inline assembly to minimize the cost of the routines to save and restore

[1] unless the encountered task scheduling point is a barrier region.

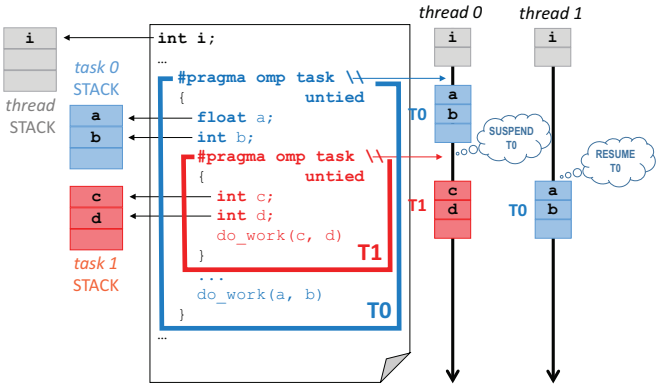

Fig. 4. *untied* task suspension with *task contexts* and per-task stacks.

architectural state.

Figure 4 shows how task suspension works in our approach for *untied* tasks (WFS is assumed). Initially the thread on which the code shown in figure is executing uses its own private stack (in gray). When the outermost task region (T_0) is encountered the context of the current task is saved in the TC (including the current SP), then the thread is rescheduled to executing the new task T_0. The SP of the thread is updated to the stack of T_0 (in blue) and the new task is started. When the creation point of the innermost task T_1 is reached an identical procedure is followed. The context of T_0 is saved in its TC, which is *push*ed back in the queue, then thread 0 is pointed to the stack of T_1 (in red). Now the suspended T_0 can be *pull*ed out of and restarted by thread 1.

On top of this basic mechanism, a number of other design choices were made to minimize the cost of our runtime support.

*a) **Beware the Zombies**:* Supporting nested tasks requires to keep in the runtime a *tree* data structure that represents the task hierarchy. A parent task has a link to its children and vice versa, to facilitate exchange of information about execution status. For example, a parent task needs to be informed about execution completion of its children to support `taskwait`. When a parent task completes execution its children become orphans, and should not care to inform the parent. The fastest solution to handle parent task termination in terms of bookkeeping would be not to delete the descriptor, but just to maintain the task in a *zombie* status until all children have completed. This operation would require a simple update to the descriptor, which can be executed in very short time. However, this solution brings to a memory occupation that is not acceptable for our constrained platform. Thus, we opt for a costlier removal of the descriptor from the *tree*. As a consequence, all child tasks must receive an update from the parent to avoid dangling pointers to a deallocated descriptor.

*b) **Speed up the taskwait**:* Task level synchronization is widely used in recursive-based parallel patters. Here typically a fixed number of tasks is created at every recursion level, and their execution is synchronized with a `taskwait` directive. When a parent task encounters a `taskwait` it should wait until all the children (first-level descendants) have completed, but typically for performance the thread hosting the parent task is allowed to switch to executing one of the children tasks. In the baseline implementation this feature is implemented by just traversing the list of children tasks in the *tree* data structure, and inspecting their status to verify that it is set to *WAITING*.

We changed this mechanism to rely on two queues per task, to directly reference children in the *WAITING* and *RUNNING* states, respectively. Upon creation, a task is inserted in the *WAITING* queue. Every time that a task starts to execute, the runtime moves this task from the *WAITING* queue to the *RUNNING* queue, and vice versa in case of suspension.

Decoupling waiting and running tasks requires a costlier bookkeeping upon task insertion and extraction, but allows faster support for `taskwait`, as it is no longer required to search the tree for *WAITING* tasks. In the baseline implementation this benefit was not evident, as the `taskwait` is virtually useless for flat parallel patterns. On the contrary, in recursive parallel patterns it is extensively used, and this design choice pays off.

*c) **Only who's truly ready gets in the queue**:* The runtime design rely on a centralized queue where all tasks in the *WAITING* state are ready for extraction and execution. Suspended tasks are also pushed back in this queue. We found that in presence of recursive parallel patterns it is important to distinguish between suspended tasks that could be resumed at any time, and tasks that are suspended due to a scheduling constraint that needs to be unblocked. A typical example is, again, tasks suspended upon a `taskwait` (or due to a data dependence). As already mentioned, recursive parallelism extensively relies on such form of synchronization, thus hosting this type of suspended tasks in the central queue used to lead to a situation where we would repeatedly *pop* from there a task just to realize that the scheduling constraint was still unsatisfied. We would then have to *push* back the task in the queue and retry. Checking the status of the task before extracting it does not entirely solve the problem, as it requires time-consuming search operations. To deal with this problem we changed the implementation so as to not re-insert in the queue suspended tasks with a unresolved dependence. Such tasks are kept floating instead, and it is up to the task that will eventually resolve the dependence to *push* them back into the queue. This modification requires some additional checks to deal with the above mentioned case, but greatly improves the performance of recursive parallel programs.

*d) **Pre-allocate is the watchword**:* To minimize the overhead for dynamic resource allocation (memory, locks, task descriptors, ..) we have extensively used pools of pre-allocated resources. This is significantly faster than `malloc`-like primitives and does not require lock-protected operations, as we adopt thread-private resources. The downside is memory occupation. Since the targeted architecture relies on scratchpad memory (the TCDM) rather than data caches, we have to wisely use the available L1 space. The specific platform instance that we consider features 256KB TCDM. A reasonable design solution would be to dedicate roughly 10% of this fast memory to hosting tasking support data structures (≈25KB). The original task descriptor has a size of 80 bytes, while the extensions that we introduced require another 72 bytes for the contexts, plus the stacks. Private thread stacks are configured to be 1KB (a common choice for embedded systems), while task stacks are by default 1/4 of that size. Clearly all those values are parameters in our design, and can be changed depending of the available L1 memory.

Considering these values, our system can host simultaneously ≈300 *tied* tasks (without context) or 64 *untied* tasks (or a mix of the two) in the fast L1 TCDM. When the available space is over we fallback to main memory (with much slower access time to the descriptors) or we enable our cutoff policies.

VI. Experiments

To validate our design we performed an extensive set of experiments considering two microbenchmarks that allow us to tune the granularity of the tasks considering two relevant parallel patterns: i) LINEAR (flat parallelism); ii) RECURSIVE (nested parallelism).

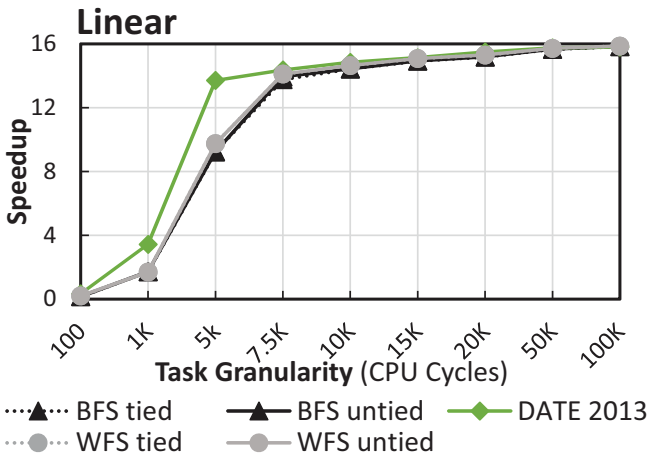

Fig. 5. Speedups achieved with the LINEAR policy

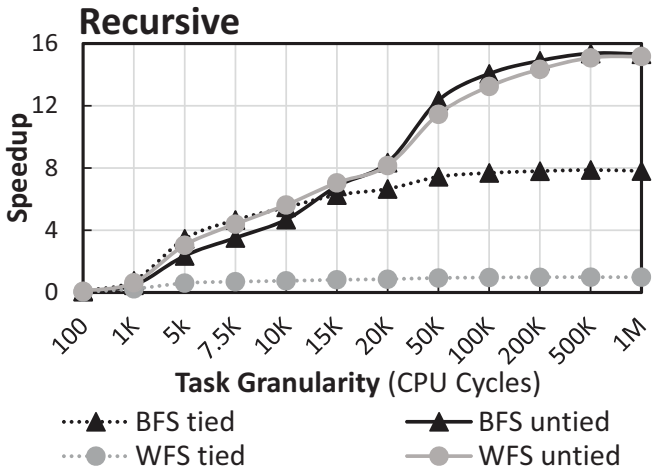

Fig. 6. Speedups achieved with the RECURSIVE policy

Figure 5 shows the speedups enabled by our design for increasing task sizes. In this experiment we directly compare to [1] (the DATE 2013 curve), to see the impact that our enhancements to support *untied* tasks have on this lightweight implementation. We test WFS and BFS scheduler using *tied* and *untied* tasks. The figure shows task granularity (cycles, or ALU operations) on the horizontal axis and speedup on the vertical axis. The LINEAR microbenchmark is based on a simple loop from which 1048 tasks are created (one per loop iteration). Compared to [1] our solution performs only slightly worse in the very fine grained task region (around 5K cycles). Beyond that point our implementation is equivalently efficient.

Figure 6 shows the efficiency of our runtime for the recursive parallel pattern, considering *tied* and *untied* tasks and BFS and WFS policies. The RECURSIVE microbenchmark builds a binary tree of depth N = 11 (2048 tasks) recursively. This is similar to a classical Fibonacci algorithm, where each of the two recursive calls is enclosed in a `task` directive. A `taskwait` directive is placed after the creation of the two tasks. The first result that we observe is that only *untied* tasks can achieve the maximum speedup, when WFS is used.

WFS and *tied* tasks imply completely sequential execution, as we already discussed in Section IV. BFS and *tied* tasks (the default for OpenMP, and representative of what could be achieved by [1]) has a peak at 8× speedup. This effect is due to the behavior of `taskwait` in presence of *tied* tasks. If a *tied* task is stuck on a `taskwait` and there are no children tasks in the *WAITING* state (e.g., few tasks generated at each recursion level, like in the binary tree), that task is bound to wait until the children have finished. For the binary tree example this leads to exactly half of the threads getting stuck, which explains the maximum speedup observed in this configuration. This problem is circumvented by *untied* tasks, which can reschedule the threads hosting the stuck tasks to other ready tasks.

In general, it is possible to see that RECURSIVE implies much higher overhead than LINEAR. This is justified by a significantly increased contention for shared data structures (queues, trees, etc.), as in this pattern multiple threads are concurrently creating tasks. Even if we have struggled to make the lock-protected operations to operate on shared data structures as short as possible, their sequentialization over multiple requestor is evident. As a result, it takes an order of magnitude coarser tasks than in the LINEAR case to achieve nearly-ideal speedups.

It is well known from literature that cutoff policies are extremely effective at mitigating this problem [16] [15]. Figure 7 shows results for the same microbenchmark when a simple cutoff policy is applied on top of BFS and WFS schedulers. The cutoff policy simply disregards `task` directives encountered in the code when a maximum number of tasks is already present in the queues. From that point on new tasks will be folded into a coarse-grained one executing sequentially on the same thread. Clearly this may lead to unbalanced execution if the amount of work already distributed to threads is not evenly distributed at that point. This can be seen in Figure 7, where BFS for *untied* tasks reaches a plateau at 10× due to poor load balancing. WFS for the same task type achieves ideal speedup due to better workload distribution. Clearly these effects heavily depend of the chosen cutoff policy [15].

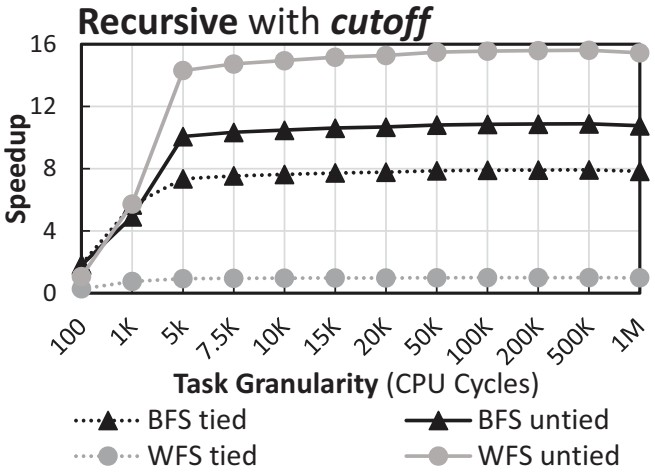

Fig. 7. Speedups achieved with the RECURSIVE policy considering cutoff.

The most important result that we can see from Figure 7 is that adding cutoff policies brings back the efficiency of our runtime support to what was observed in the LINEAR microbenchmark an in [1]. Near-ideal speedups are achieved for tasks as small as 5K cycles.

A. Comparison to other tasking runtime implementations

For completeness, we compare our implementation to other representative commercial and academic ones, targeted at general purpose and high performance computing systems:

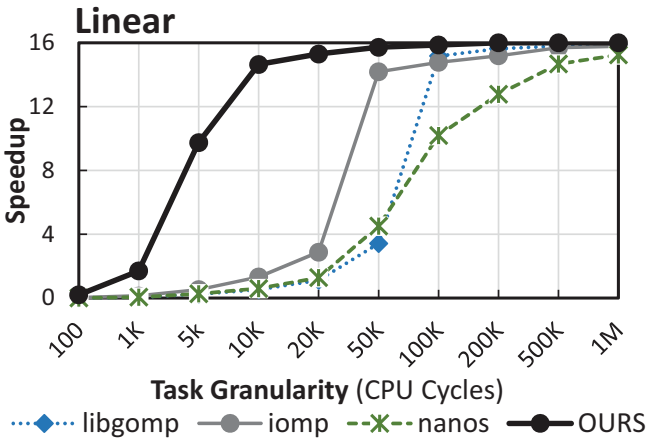

Fig. 8. Speedup of various tasking runtimes using the LINEAR policy.

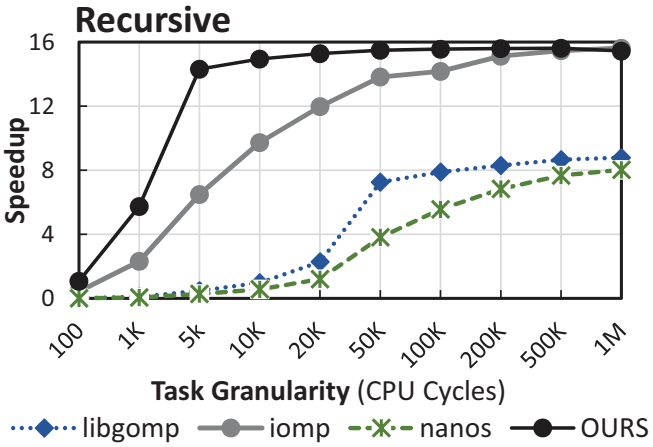

Fig. 9. Speedup of various tasking runtimes using the RECURSIVE policy.

- ***libgomp***: the GNU OpenMP implementation (GCC 4.9.2);
- ***iomp***: the Intel OpenMP implementation (ICC 15.0.2);
- ***nanos***: the BSC OpenMP implementation (Mercurium 15.06 + Nanos++);
- ***OURS***: our OpenMP implementation.

The same microbenchmarks described above have been used for this experiment, considering *untied* tasks and a BFS policy. As a target platform for these experiments we used a compute server equipped with two Intel Haswell with 8 cores @ 2.40 GHz. Figures 8 and 9 show that our implementation allows to achieve near-ideal speedups for one order of magnitude smaller tasks compared to the others, both for the LINEAR case and the RECURSIVE case.

VII. Conclusion

Task-based parallelism has the potential to provide efficient exploitation of manycore accelerators, offering flexible support to the fine-grained and irregular parallelism found in embedded applications. In this paper, we have presented an optimized implementation of the OpenMP tasking model for embedded parallel accelerators. To the best of our knowledge, the proposed design is the first to enable support for *untied* tasks and recursive parallel patterns for the targeted class of computing systems. We demonstrate that, despite the significant extensions in the supported semantics, our solution does not degrade the efficiency of the most lightweight OpenMP implementation for embedded manycores. When compared to OpenMP implementation for high performance computing systems, our design achieves near-ideal speedups for one order of magnitude smaller tasks.

VIII. Acknowledgements

This work has been supported by the EU FP7 Project P-SOCRATES (g.a. 611016) and EU ERC Project MULTITHERMAN (g.a. 291125).

References

[1] P. Burgio et al. Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters.. In Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13). EDA Consortium, San Jose, CA, USA, 1504-1509.

[2] G. Mitra et al. Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture. Using and Improving OpenMP for Devices, Tasks, and More Volume 8766 of the series Lecture Notes in Computer Science pp 202-214.

[3] D. Melpignano et al. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications *Design Automation Conference, 2012*, pp.1137-1142.

[4] R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[5] Apple, Inc. Grand Central Dispatch. https://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html. 2010.

[6] S. Kumar et al. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News*, 35:162–173, June 2007.

[7] OpenMP Application Program Interface v.4.5.0. http://www.openmp.org/mp-documents/openmp-4.5.pdf. November 2015.

[8] A. Duran et al. Evaluation of OpenMP task scheduling strategies. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 100–110, Springer-Verlag, 2008.

[9] E. Ayguadé et al. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, Mar. 2009.

[10] Marlin, C. D. Coroutines: A programming methodology, a language design and an implementation *Springer Science & Business Media*, (No. 95)

[11] Faxén, K. F. Wool-a work stealing library. *ACM SIGARCH Computer Architecture News*,36(5), 93-100.

[12] J. Reinders, Intel Threading Building Blocks. *O'Reilly Media, Inc.*, 2007

[13] Kalray Corporation, Many-core Kalray MPPA, 2012. [Online]. Available:http://www.kalray.eu

[14] Bortolotti, D., Pinto, C., Marongiu, A., Ruggiero, M., & Benini, L. (2013, May). VirtualSoC: A full-system simulation environment for massively parallel heterogeneous system-on-chip. *In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (pp. 2182-2187). IEEE.

[15] Duran, A., Corbalán, J., & Ayguadé, E. (2008, November). An adaptive cut-off for task parallelism. *In High Performance Computing, Networking, Storage and Analysis*, 2008. SC 2008. International Conference for (pp. 1-11). IEEE.

[16] Duran, A., Corbalán, J., & Ayguadé, E. (2008). Evaluation of OpenMP task scheduling strategies. In OpenMP in a new era of parallelism (pp. 100-110). *Springer Berlin Heidelberg*.

[17] Marongiu, A., Capotondi, A., Tagliavini, G., & Benini, L. (2015). Simplifying Manycore-Based Heterogeneous SoC Programming with Offload Directives. *Industrial Informatics, IEEE Transactions* on (Volume:11 , Issue: 4)

[18] Chapman, B., Huang, L., Biscondi, E., Stotzer, E., Shrivastava, A., & Gatherer, A. (2009, May). Implementing OpenMP on a high performance embedded multicore MPSoC. *In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on (pp. 1-8). IEEE*.

[19] Wang, C., Chandrasekaran, S., Chapman, B., & Holt, J. (2013, February). libEOMP: a portable OpenMP runtime library based on MCA APIs for embedded systems. In Proceedings of the 2013 *International Workshop on Programming Models and Applications for Multicores and Manycores* (pp. 83-92). ACM.

[20] Podobas, A., Brorsson, M., & Faxén, K. F. (2015). A comparative performance study of common and popular task-centric programming frameworks. *Concurrency and Computation: Practice and Experience*, 27(1), 1-28.

[21] S. Agathos et al. Design and Implementation of OpenMP Tasks in the OMPi Compiler. In *15th Panhellenic Conference on Informatics*, pages 265–269, 2011.