ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Tagliavini, G., Cesarini, D., Marongiu, A. (2018). Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, 29(9), 2150-2163 [10.1109/TPDS.2018.2814602].

Availability:
This version is available at: https://hdl.handle.net/11585/652244 since: 2019-10-22

Published:

DOI: http://doi.org/10.1109/TPDS.2018.2814602

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2026

This is the post peer-review accepted manuscript of:

G. Tagliavini, D. Cesarini and A. Marongiu, "Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking," in IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 9, pp. 2150-2163, 1 Sept. 2018.

The published version is available online at:

https://doi.org/10.1109/TPDS.2018.2814602

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking

Giuseppe Tagliavini, *Member, IEEE,* Daniele Cesarini, Andrea Marongiu, *Member, IEEE*

Abstract—In recent years, programmable many-core accelerators (PMCAs) have been introduced in embedded systems to satisfy stringent performance/Watt requirements. This has increased the urge for programming models capable of effectively leveraging hundreds to thousands of processors. Task-based parallelism has the potential to provide such capabilities, offering high-level abstractions to outline abundant and irregular parallelism in embedded applications. However, efficiently supporting this programming paradigm on embedded PMCAs is challenging, due to the large time and space overheads it introduces. In this paper we describe a lightweight OpenMP tasking runtime environment (RTE) design for a state-of-the-art embedded PMCA, the Kalray MPPA 256. We provide an exhaustive characterization of the costs of our RTE, considering both synthetic workload and real programs, and we compare to several other tasking RTEs. Experimental results confirm that our solution achieves near-ideal parallelization speedups for tasks as small as 5K cycles, and an average speedup of 12$\times$ for real benchmarks, which is $\approx 60\%$ higher than what we observe with the original Kalray OpenMP implementation.

Index Terms—Heterogeneous Embedded Systems on Chip, Programmable Many-Core Accelerators, Tasking, OpenMP.

1 Introduction

Over the past decades, multi-core processors hit both high-performance computing (HPC) and embedded systems (ES) markets. The ever-growing computational capabilities and the related exponential increments in power consumption have progressively paved the way for the introduction of many-core systems and, ultimately, heterogeneous systems based on programmable many-core accelerators (PMCA) in both domains [1] [2] [3]. Embedded multi-processor on-chip systems (MPSoC) are increasingly adopting this paradigm, where a general-purpose *host* processor is coupled to a PMCA, onto which highly-parallel portions of an application can be offloaded to improve overall performance/watt.

In the heterogeneous computing paradigm, applications are split into multiple tasks that run in parallel on different, non-homogeneous cores, exacerbating an important challenge already faced by HPC designers at the multi-processor system scale: the extraction of parallelism from applications [4] [5]. 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 PMCA should be controlled by system software libraries and runtime environments (RTE).

Task-based parallelism (a.k.a. *tasking*) has the potential to provide such features, as it provides a powerful conceptual framework to exploit irregular parallelism in target applications [6]. In the HPC domain, parallel programming models such as Cilk++, Intel TBB, Apple GCD have demonstrated the effectiveness of tasking at simplifying application development. OpenMP, in particular, evolved over the years from a thread-centric programming style to include the concepts of task parallelism [7] and computation acceleration. Due to its ease-of-use and its friendly directive-based programming interface, OpenMP eventually became one of the most known and widely adopted parallel programming models in HPC, representing the *de-facto* standard for shared-memory systems.

The advantages of adopting OpenMP in the context of embedded systems have been discussed in a large literature body, highlighting the feasibility of implementing its semantics in RTEs sitting on top of resource-constrained middleware or bare-metal [8] [9] [10] [11]. Recent research has highlighted that the OpenMP tasking model has an additional advantage for real-time embedded systems software development, in that it retains certain similarities to the formalisms used to describe real-time applications (e.g., task graphs), which makes it a very appealing approach to fulfill both needs for a programming model for embedded PMCAs and a methodology to employ state-of-the-art techniques for scheduling with timing guarantees [12] [13] [14] [15] [16]. 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. Indeed, while time overheads in task-based programs are very relevant also in the HPC domain

G. Tagliavini and D. Cesarini are with the Department of Electrical, Electronic, and Information Engineering (DEI) of the University of Bologna, Italy. Email: {giuseppe.tagliavini, daniele.cesarini}@unibo.it
A. Marongiu is with the Department of Informatics, Science, and Engineering (DISI) of the University of Bologna, Italy. Email: a.marongiu@unibo.it
This work was partially supported by the EU FP7 project P-SOCRATES (g.a. 611016) and by the EU Horizon 2020 RIA project HERCULES (g.a. 688860).

[17] [18] [19], the granularity of HPC tasks is typically orders of magnitude coarser than that of embedded real-time tasks [20] [21], which inherently tolerates much larger overheads. Space (i.e., memory footprint) overheads are generally not at all a concern in HPC software, which poses to RTE designers much less stringent constraints concerning the choice of the support data structures and algorithms that can be employed.

State-of-the-art tasking RTEs for embedded PMCAs [20] [9] succeed in achieving low overheads and enabling high speedups for very fine-grained tasks, but only for simple single-level 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. A *tied* task is a schedulable work unit with a specific constraint: when 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 describe a lightweight OpenMP tasking RTE design for a state-of-the-art embedded PMCA, the Kalray MPPA 256 [22]. To overcome the limitations discussed above, our solution enables the *untied* task model. When suspended, *untied* tasks can be resumed by any available thread, thus significantly increasing the potential for parallelism exploitation. As a consequence, our solution also enables support for WFS. Supporting *untied* tasks requires 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. We provide a detailed and insightful discussion of the key design choices that minimize the cost for such modifications. *Cutoff* policies, which consist of mechanisms to prevent the saturation of runtime resources when task production rate is higher than the consumption rate, are also evaluated, as they proved to be among the most effective RTE features to reduce the overhead associated to *untied* task management in such scenarios.

As an additional contribution, we provide an exhaustive characterization of the costs of our lightweight tasking RTE, considering both synthetic workload (to sweep along the key tunable parameters in real applications, and stress relevant corner cases) and real programs from the Barcelona OpenMP Task Suite (BOTS) [23]. The key findings can be summarized as follows:

1) careful design of *untied* task support enables near-ideal speedups for very fine-grained (around 5K cycles), single-level task parallelism, on a par with much simpler RTEs only supporting *tied* tasks [20];
2) *cutoff* policies enable the same near-ideal speedups in (typically much costlier) recursive (nested) task creation patterns for tasks of the same small size (around 5K cycles);
3) WFS enables significantly higher speedups (up to 60%) than BFS when *untied* tasks are used in recursive patterns;
4) parallel execution of the BOTS benchmarks on 16 Kalray cores is on average $12\times$ faster than single-core execution with our RTE. This speedup is $\approx 60\%$ higher than what we observe when running the same programs with the original Kalray OpenMP implementation.

In addition, we compare our RTE design to several others from the HPC domain, 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. This is a key result, as it practically makes *untied* tasks an effective parallelization abstraction for embedded PMCAs.

The rest of the paper is organized as follows. Section II discusses the related work. Section III describes the Kalray MPPA-256 Bostan platform, the target architecture of this work. Section IV introduces the fundamental notions of OpenMP tasking, a formal analysis of the limitations of *tied* tasks and the basic infrastructure for task execution. Section V describes support for *tied* and *untied* tasks, providing an in-depth discussion of the supported features and the key design choices. Section VI discusses experimental setup and results. Section VII discusses conclusive remarks.

2 Related work

Tasking has been successfully used in the HPC domain to parallelize complex algorithms leveraging sophisticated control structures.

Cilk [24] and *Cilk++* [25] extend standard C and C++ with custom keywords. The liveness of *Cilk* tasks is determined by their lexical scope, which enables the compiler to allocate task descriptors on the stack avoiding the overhead of dynamic allocation. This reduces time and space overheads of the runtime, but it also restricts how tasks can be specified (in particular only function calls can be annotated). *Intel TBB* [26] is a C++ template library that introduces tasks as an abstraction to decouple parallel workload from the underlying threads. The creation of tasks is made transparent to the programmer by means of generic parallel patterns. Both *Cilk* and *TBB* save task descriptors in data structures which are local to the running thread to avoid the overhead of a concurrent data structure. A *work-stealing* policy [27] guarantees load balancing across threads.

Apple GCD [28] is a low-level API supporting C-family languages. *GCD* enables to describe tasks either as a function or as a syntactic *block*. A programmer adds a task to a specific *dispatch queue* and the *GCD* scheduler decides which thread to assign it to. *GCD* is fully integrated with the OS scheduler, which decides how much parallelism is required based on available system resources.

Wool [19] is a C library supporting fine-grained task parallelism by exposing a user-level task scheduler. *Wool* introduces two main macros to support programmers, `SPAWN`

and SYNC, which are equivalent to the *Cilk* keywords. The work-stealing policy is also similar to *Cilk* and *TBB*, but overall *Wool* provides more tuning parameters (e.g., the maximum arity of spawned functions). Wool is explicitly designed to provide minimum overhead for tasking support[1], and as such it does not depend on any specific system library or other legacy code. Unlike the rest of the discussed HPC programming models, this would make it easy to implement *Wool* on top of embedded real-time operating systems and/or middleware. However its programming style is extremely low-level, which has hindered its adoption.

OpenMP can be considered the *de-facto* standard for shared-memory parallel programming, thanks to its easy-to-use and intuitive directive-based interface. Since the specifications version 4.0 [29] OpenMP features sophisticated support for task parallelism with dependencies. Researchers have actively explored the effectiveness of OpenMP tasks in the context of HPC applications and systems [30] [31] [32] [32] [33]. Tasking is an appealing programming model for embedded PMCAs as well, not only because of the powerful conceptual framework it provides to abstractly outline parallelism at the algorithmic level, but also because of its closeness to traditional abstractions used in real-time scheduling analysis for embedded software development [12] [13] [14] [15] [16].

While OpenMP has gained much attention also in the embedded domain [8] [9] [10] [11], most of the work in this area only focuses on the core functionality provided before the introduction of tasks [34] [35]. Not much work has been done so far on demonstrating the benefits of tasking for fine-grained embedded workloads [21] or for proposing lightweight and efficient tasking implementations for embedded MPSoCs.

The overheads implied by the sophisticated OpenMP tasking execution model are the key limiter for its adoption in the embedded domain.

Burgio et al. [20] have developed streamlined OpenMP tasking support for a shared-memory, multi-core cluster, showing that their approach allows for near-ideal speedups when leveraging tasks as fine-grained as a few thousand cycles. The major limitation of this proposal lies in the fact that only *tied* tasks are supported, which substantially decreases the performance of irregular applications.

Agathos et al. [36] describe OpenMP support for the STHORM platform, where it is also mentioned support for tasks. The authors discuss the need for a less feature-rich implementation compared to HPC counterparts [31], to meet memory size restrictions, which implies the support of *tied* tasks only. The experimental results discuss only traditional loop parallelism, with no analysis on the benefits and costs of tasks.

Vargas et al. [37] present a lightweight runtime for OpenMP v4. The focus of this work is on discussing timing-predictability extensions to OpenMP, and the contribution is centered on presenting the compiler support to extract a task dependency graph for off-line application analysis and the runtime data structures and algorithms required to implement the static task schedule. As such, the work is fully orthogonal to what we present.

1. As shown in our experimental results section, *Wool* is the only programming model from the HPC domain that has comparable performance to our solution.

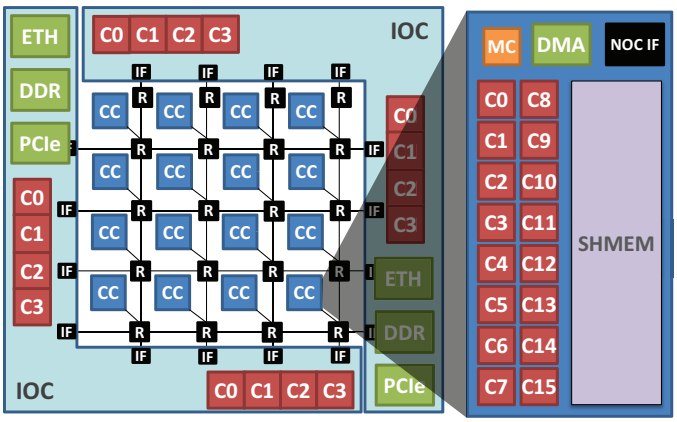

Fig. 1. Kalray MPPA-256 Bostan (block diagram).

Overall, the main limitation of the few OpenMP 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 RTE capable of enabling near-ideal speedups for recursive parallel patterns employing very fine-grained tasks. Different from previous work, we provide an exhaustive characterization of OpenMP tasking overheads, with both synthetic and real-life benchmarks running on a state-of-the-art embedded PMCA. We also show a comparison with other tasking runtimes (in Section 6.3) to highlight benefits and drawbacks of our solution.

3 Target Architecture

Embedded MPSoCs have historically relied on architectural heterogeneity for better energy efficiency, integrating on the same chip processors with different ISAs and various hardware acceleration blocks [38] [39] [35] [40]. Recently, a dominating heterogeneous design paradigm is that of coupling a general-purpose *host* processor to a PMCA.

To allow the efficient on-chip integration of hundreds of cores, PMCAs rely on optimized computing *clusters* as their key building block. Specifically, 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 network-on-chip (NoC).

As a concrete embodiment of such architectural template, we consider in this paper the Kalray MPPA-256 Bostan platform [22]. The MPPA-256 is a single-chip many-core processor manufactured in 28 nm CMOS technology for compute-intensive embedded applications, based on the MPPA (Multi-Purpose Processor Array) technology by Kalray. This product features 256 accelerator processors on a single die – plus 32 control cores – and it is composed of an array of 16 computing clusters (CC) and 2 I/O clusters (IOC) connected through a high-speed NoC. Figure 1 depicts the structure of a MPPA-256 Bostan chip.

Each compute cluster is composed of 16 identical cores (C0-C15), each featuring private L1 instruction and non-coherent data caches, plus a local management core (MC) and a shared L2 memory (2 MB). Inter-processor communication happens through this shared memory, which leverages a multi-bank design enabling low latency access. To reduce the probability of conflicts due to simultaneous accesses, the interconnect is capable of implementing cache-line-level address interleaving. It is also possible to configure the routing for contiguous addressing, which is useful to implement partitioned access to different memory banks from different cores. Communication to other clusters or to the main system memory can only happen via DMA transfers through the NoC, which provides a full duplex bandwidth of up to 3.2 GB/s between adjacent clusters.

Each I/O cluster includes two quad-core processors, which we refer to as the *host* throughout the paper. Any program is started on the I/O cores, which are then responsible to properly offloading computation to the clusters. These cores are based on the same VLIW architecture adopted within computing clusters. The MPPA-256 Bostan processor communicates with external devices through I/O clusters located at the periphery of the NoC. The I/O clusters implement various standard interfaces, including two DDR3 channels (64-bit with optional ECC, up to 12.8 GB/s), two PCIe Gen3 X8 and two Ethernet controllers.

4 Background

In this section we provide background information related to the OpenMP tasking model and the basic design choices.

4.1 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 *fork*ed and recruited into a parallel *team*. At the end of the *parallel region* (the boundaries of which are identified by the lexical scope of the `parallel` construct), an implicit *barrier region* is encountered, where the threads are *join*ed. 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. With the specification version 4.0 [29] support for task dependencies has been added.

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) dependencies among tasks; ii) immediate or deferred execution (overrides default scheduling policy); iii) `tied` or `untied` task *type* (to the thread that first encounters the task creation point).

All tasks bound to a given parallel region are guaranteed to have completed at the implicit barrier region at the end of the parallel region, as well as at any other barrier region associated to an 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.

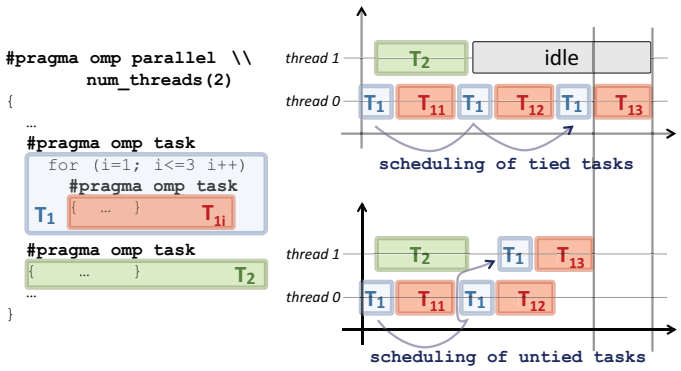

Fig. 2. Example of an OpenMP program using the `task` construct. The timing charts on the right side show the different scheduling outcome when applying the WFS policy to *tied* (top right) or *untied* (bottom right) tasks.

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. When a *tied* task is suspended, it can later only be resumed by the same thread that originally started it (i.e., the task region is tied to the executing thread). *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. 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[2]. When a thread encounters a TSP it can begin the execution of a new task, or resume a previously suspended one, provided that *task scheduling constraints* (TSC) are fulfilled.

Among TSCs, the second one reported in the specification (TSC#2) is particularly relevant to this work, as it limits the flexibility of *tied* task scheduling. TSC#2 recites: *Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a barrier region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendant task of every task in the set.*

Tied tasks are the default in OpenMP, as they attempt to establish a trade-off between ease of programming[3] and scheduling flexibility [7]. Their scheduling constraints, however, pose relevant limitations to the achievable performance, as we explain in the next section. It is worth mentioning that the limitations of *tied* tasks have been pointed out also in the context of providing timing guarantees via schedulability analysis to OpenMP programs [41], as they prevent the implementation of *work-conserving* schedulers.

2. For the sake of simplicity we restrict our discussion to the most relevant TSPs, for the full list the interested reader is referred to the OpenMP specifications.

3. 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.).

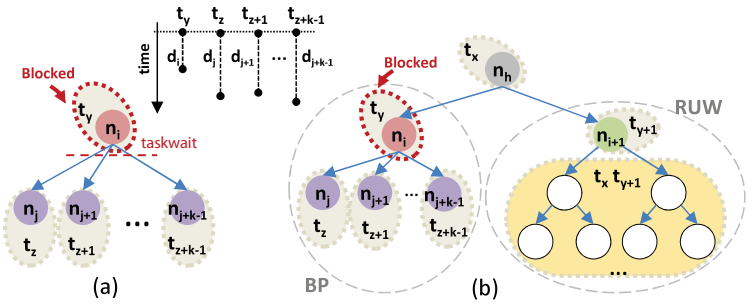

Fig. 3. a) A *blocking pattern (BP)* using *tied* tasks. The longest-lived descendant of task n_i determines blocking time of thread t_y. b) Thread t_y in a BP cannot participate in executing *recursive useful work (RUW)*, thus limiting parallelism.

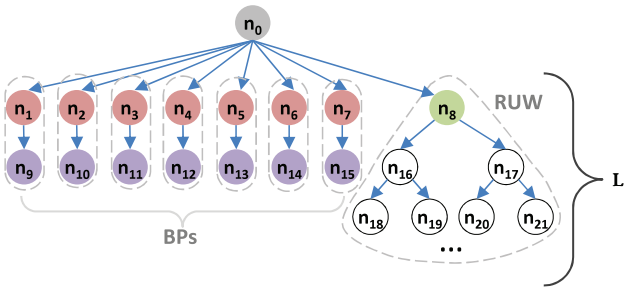

Fig. 4. Worst-case scenario for the performance loss due to thread blocking ($t = 16$).

4.2 Limitations of tied tasks

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 [32].

Figure 2 shows the behavior 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.

Due to the fact that suspended *tied* tasks can only be resumed by the thread that first started them (see Section 4.1), using WFS leads to a null speedup. For this reason, BFS is the only policy available for those OpenMP implementations that only support *tied* tasks. However, TSC#2 also limits the parallelism under BFS in certain situations.

Limitations inherent to the execution model - Let us consider the creation of a *k*-ary tree using *tied* tasks (implies BFS), as depicted in Figure 3 (a). The root task n_i performs i) the creation of k descendants; ii) the execution of the task workload; iii) a synchronization step (*taskwait*) to wait for the termination of the direct descendants. Thread t_y starts the execution of *tied* task n_i with duration d_i. As soon as they are created, descendant tasks n_j, n_{j+1}, ..., n_{j+k-1} with durations d_j, d_{j+1}, ..., d_{j+k-1} are assigned to k threads (t_z, t_{z+1}, ..., t_{z+k-1}). If these descendant tasks are leaf nodes, and at least one of the threads they have been mapped to is still executing when t_y reaches the `taskwait` placed after n_i, i.e.:

$$\exists \alpha \in [0, ..., k) \mid d_i < d_{j+\alpha}$$

then, because of the TSC#2, t_y is *blocked* for a time equal to $max_{\alpha \in [0,...,k)} \{d_{j+\alpha} - d_i\}$, reducing the available parallelism.

Figure 3 (a) illustrates the situation when thread blocking happens due to TSC#2. However for thread blocking to be detrimental to performance there need to be available work to be done in the system that the blocked thread(s) cannot execute. Performance loss due to thread blocking happens, for example, when many tasks are created from within a sibling recursive tree pattern, as shown in Figure 3 (b). Assuming that there are exactly $k + 3$ threads available, under BFS these would be assigned to nodes $n_h, n_i, n_{i+1}, n_j, n_{j+1}, ..., n_{j+k-1}$. In this case, all the tasks belonging to the sibling sub-tree rooted at node n_{i+1} can be executed by the free thread t_{y+1} and by the non-blocked thread t_x (the one that originally started the ancestor node n_h). Thread t_y is blocked, and cannot participate in the execution of these tasks until all the descendants of task n_i have completed (at worst, t_y remains blocked for the entire execution of the sub-tree rooted at node n_{i+1}). If we identify the sub-tree rooted at node n_i as the *blocking pattern (BP)*, and the sub-tree rooted at node n_{i+1} as the *recursive useful work (RUW)*, system-wide the worst case happens when:

1) there are multiple instances of the *BP*, each rooted at a node that only has one descendant (i.e., several blocked threads, only one task doing useful work per blocking pattern);
2) only one extra thread – besides the one that originally started execution of the originating root task – is available to execute the *RUW*.

Analytically, if t is the total number of threads in the system, in the worst-case scenario there are $t - 2$ threads involved in execution of the various BPs. Given that the root node of each BP only has one descendant, there are exactly $(t - 2)/2$ blocked threads in the system (the remaining half is executing useful work in the leaf nodes). Figure 4 depicts an instance of the worst-case scenario for $t = 16$. In this case, the speedup loss can be computed as follows: Nodes from n_9 to n_{15} are the leaves from the BPs, and L is the height of the RUW sub-tree having n_8 as root. We assume that the duration of all *leaf* nodes is d_{leaf}, while the duration of all

other nodes in the tree is d_{work}, and $d_{leaf} > d_{work}$. Threads executing nodes from n_1 to n_7 are blocked for $d_{leaf} - d_{work}$ time units.

The number N_l of nodes at the l-th level of a balanced k-ary tree can be computed as the sum of the first l terms of a geometric series:

$$N_l = \sum_{i=0}^{l} k^i = \frac{1-k^{l+1}}{1-k} \tag{1}$$

Considering that the RUW sub-tree in Figure 4 has L levels (numbered from 0 to $L-1$), its total number of nodes can be computed as:

$$N = \sum_{l=0}^{L-1} N_l = 2^L - 1 \tag{2}$$

We can compute the total duration for a sequential execution of the tree as:

$$T_{seq} = 9 \cdot d_{work} + 7 \cdot d_{leaf} + (2^L - 2) \cdot d_{work} \tag{3}$$

In detail, $9 \cdot d_{work}$ is the duration of nodes from n_0 to n_8, $7 \cdot d_{leaf}$ is the duration of nodes from n_9 to n_{15}, and $(2^l - 2) \cdot d_{work}$ is the duration of nodes from n_{16} to n_{N-1}.

Considering a parallel execution with t threads, the time required to execute nodes from n_0 to n_{15} is d_{leaf}[4]. The RUW sub-tree consists of all nodes of duration $d_{work} < d_{leaf}$, thus each of the non-blocked threads can execute $\left\lceil \frac{d_{leaf}}{d_{work}} \right\rceil$ tasks from the RUW sub-tree while leaf nodes execute[5]. Overall, after d_{leaf} time units have elapsed, all the t threads are available to execute what remains to be processed of the RUW sub-tree. If we indicate the number of non-blocked threads, available to execute the RUW sub-tree (rooted at node n_8) as $\hat{t}(n_8)$, we can compute the following execution time:

$$T_{par,subtree} = \frac{\left[(2^L - 2) - \hat{t}(n_8) \cdot \left\lceil \frac{d_{leaf}}{d_{work}} \right\rceil\right] d_{work}}{t} \tag{4}$$

And the maximum speedup can be obtained as follows:

$$\frac{T_{seq}}{T_{par}} = \frac{T_{seq}}{d_{leaf} + max\left\{0, T_{par,subtree}\right\}} \tag{5}$$

Assuming *tied* tasks, all the threads that started tasks n_1 to n_7 are blocked, and so only $\hat{t}(n_8) = 2$ threads are available until d_{leaf} time units elapsed (those that started n_0 and n_8). Assuming *untied* tasks, $\hat{t}(n_8) = 9$.

Considering the case of $d_{leaf} = l - 1$, $n = 5$ and $t = 16$ we can compute speedup values for both *tied* and *untied* models using formulas 3 and 5. We get $S_{tied} = 9.57$ and $S_{untied} = 13.4$, showing a significant gap in parallel performance. We get a different result if we suppose to add a synchronization point to leaf nodes, forcing the termination of nodes from n_8 to n_{15} to be concurrent with the termination of the recursive sub-tree. This additional constraint is equivalent to set $d_{leaf} = \frac{2^l-2}{R} + 1$. In this case the asymptotic speedup of the two solutions is $\lim_{l\to\infty} S_{tied} = 11.13$ and $\lim_{l\to\infty} S_{untied} = 16$. This case shows that the maximum parallelism of an application can be limited by *tied* tasking regardless of the application workload.

Implementation-induced limitations - Another important source of inefficiency of *tied* tasks is due to a wide-spread implementation choice to support their scheduling semantics. Several OpenMP runtime implementations [42] [20] [35] [43] further restrict TSC#2 by only allowing a new *tied* task to be scheduled if it is a ***first-level*** descendant of every suspended task. This choice is typically made to limit the large space/time overheads implied when supporting unmodified TSC#2 semantics. Being able to only track first-level descendants at any time is much faster and requires a much lower memory footprint.

On the other hand, this additional constraint further exacerbates thread blocking in idle state, to the detriment of the maximum achievable parallelism. Considering a balanced k-ary tree, with identical duration for all nodes ($d_i = d_{work}, \forall i$), each new *tied* task will be assigned to a different thread, until the number of tasks is greater (or equal) than the number of available threads t. The tree level L at which this condition is verified is:

$$\begin{aligned} \frac{1-k^{L+1}}{1-k} &\geq t \\ L &= \lceil \log_k \left[1 - t(1-k)\right] - 1 \rceil \end{aligned} \tag{6}$$

All the threads allocated to execute a task at a level $0 \leq l' < L-1$ are blocked, since the descendant tasks at level $l'+1$ are executed by different threads. The number of these threads is:

$$B_1 = \frac{1-k^{L-1}}{1-k} \tag{7}$$

The number of blocked threads at level $L-1$ depends on the number of threads available to execute at level L, since a thread is blocked if and only if all its children have been executed by a different thread. This value is:

$$B_2 = \left\lfloor \frac{t - k^{L-1} - B_1}{k} \right\rfloor \tag{8}$$

The total number of blocked thread is B, computed as:

$$B = B_1 + B_2 \tag{9}$$

Having the total number of blocked threads, the maximum ideal speedup is:

$$S_{ideal} = t - B \tag{10}$$

Overall, the reduction of speedup due to these implementation-induced limitations for *tied* tasks can be severe. For instance, considering $t = 16$ and $k = 2$, we derive $L = 4$ (Formula 6), $B_1 = 7$ (Formula 7) and $B_2 = 8$ (Formula 8). Applying Formula 10 we compute $S_{ideal} = 9$.

4.3 Basic infrastructure for tasking support

The basic support for task execution in the RTE is based on a centralized queue. Lightweight support for *push* and *pop* operations on the centralized queue (upon task creation and extraction, respectively) relies on fine-grained locking

4. These nodes are assigned to as many threads, so they execute in parallel, with the slowest dictating overall parallel execution time.

5. Note that d_{leaf} is expressed in $\frac{sec}{leaf_{task}}$ units and d_{work} is expressed in $\frac{sec}{RUW_{task}}$ units. Thus, their ratio is dimensionally a number of RUW tasks.

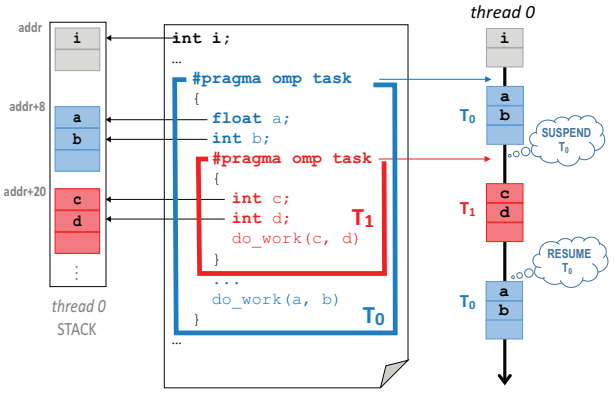

Fig. 5. *tied* task suspension in the baseline implementation [20].

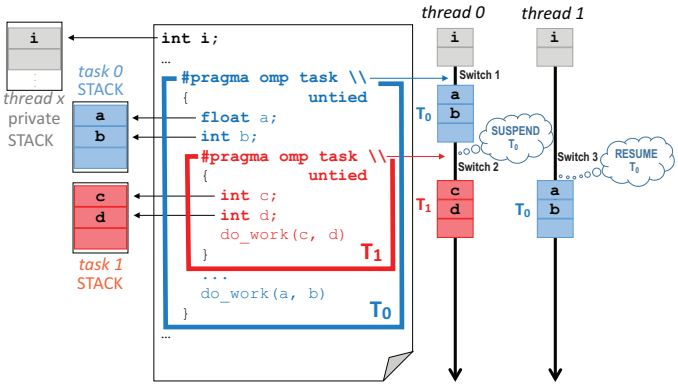

Fig. 6. *untied* task suspension with *task contexts* and per-task stacks.

mechanisms. TSPs are implemented using lightweight synchronization primitives (signal/wait on condition variables) provided by the OS layer, 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 inspects the queue to fetch another task or returns into sleeping mode only if no task is available at that time.

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) [20], 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 BFS, 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.

5 Runtime Design

Figure 5 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 is executing has an associated stack, depicted on the left side of Figure 5. 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. For instance, the stack frame for task T_0 is created at address $addr+8$ in Figure 5. The same thing happens at every nested `task` directive. For the example in Figure 5, the stack frame for task T_1 is created at address $addr + 20$. When a task is completed, the stack pointer (SP) 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 [44]. 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 memory of the cluster. This design choice ensures fast context switch, since any thread can access the shared stacks with an initial latency of 8 cycles, that is reduced to 1 cycle for subsequent cached accesses. Inline assembly is used to minimize the cost of the routines to save and restore architectural state.

Figure 6 shows how task suspension works in our approach for *untied* tasks, assuming the WFS policy described in Section 4.2. Initially the thread on which the code is executing uses its own private stack (depicted on the left side of Figure 6). When the outermost task region (T_0) is encountered (denoted by label *Switch 1*), 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 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 (*Switch 2*). Now the suspended T_0 can be *pull*ed out of and restarted by thread 1 (*Switch 3*).

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

runtime support.

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.

Speed up the taskwait - Task-level synchronization is widely used in recursive parallel patterns. Here typically a fixed number of tasks is created at every recursion level, and their execution is synchronized with a `taskwait` directive. When a task encounters a `taskwait`, it needs to wait until all the children (first-level descendants) have completed, but this does not translate in the hosting thread idling, as a `taskwait` constitute a TSP. Under the *untied* model, this thread can be rescheduled to executing any other task in the queue. Under the *tied* model, the thread can be rescheduled to executing any descendant of any suspended task. As we discussed in Section 4.2, most practical implementations of *tied* tasks only allow the thread encountering a TSP to be rescheduled to executing first-level descendants of the suspended task, to avoid costly multi-level list traversal operations. Practically, this is usually 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. Thus, this feature is is especially relevant for recursive parallel patterns, where the `taskwait` is heavily used.

No time wasters: only who's truly ready gets in the queue - The runtime design relies 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. Typical example include 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.

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 of this approach is memory occupation. Since our architectural target relies on a shared cluster memory with limited size, we have to wisely use the available space. A reasonable and practical design solution turned out to be to dedicate roughly 5-10% of the local memory to hosting tasking support data structures.

The basic task descriptor occupies 174 bytes, while the extension to support `untied` tasks require another 98 bytes for the contexts, plus the stacks. Private thread stacks are configured to be 1 KB (a common choice for embedded systems), while task stacks are by default 1/4 of that size[6]. Considering the 2 MB shared memory of the Kalray MPPA-256 clusters, with 10% of the cluster's shared memory allocated to task descriptors the runtime can host simultaneously 750 pre-allocated tied tasks or 512 untied tasks.

Optimize cutoff for fine-grained workloads - If the queue of available task descriptors is depleted during the program execution, a *cutoff* mechanism [33] is triggered. When this condition is met, the creation of new task descriptors must be suspended to avoid that runtime resources saturate when task production rate is higher than consumption rate.

Our runtime supports a *work-first* cutoff policy. In *work-first* cutoff, when a task creation point is encountered that task is executed in-place via a standard function call; in this case task descriptors for child tasks are not required, as the synchronization is enforced by serializing all the descendants on the same thread.

An important optimization for memory-constrained architectures is *single-stack cutoff*. `untied` task descriptors include task-private stacks, which in general need to be dimensioned to tolerate stack growth for worst-case recursion depth in cutoff state. This has intuitively a negative effect

6. Clearly all those values are parameters in our design, and can be changed depending on specific application requirements.

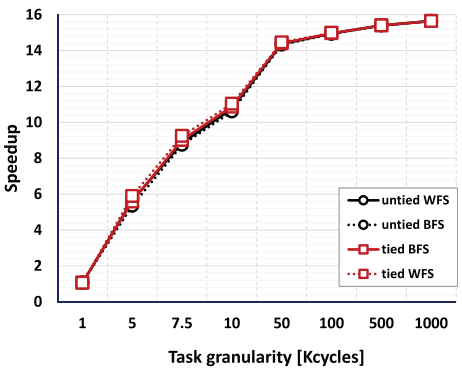

Fig. 7. LINEAR benchmark.

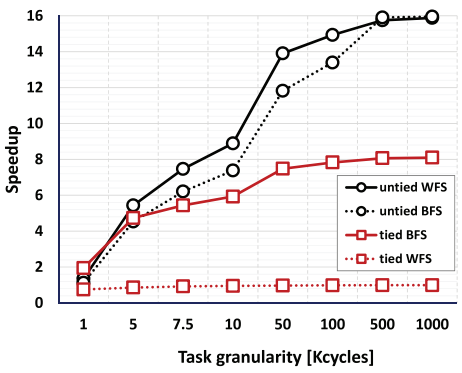

Fig. 8. RECURSIVE benchmark.

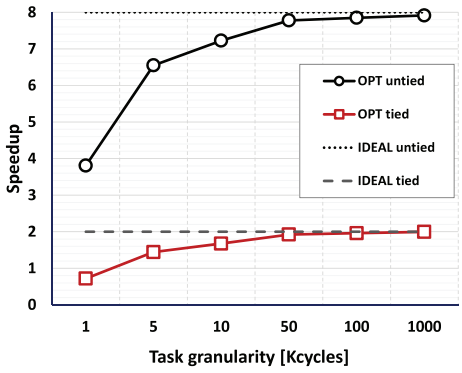

Fig. 9. MIXED benchmark.

on the maximum number of `untied` task descriptors that the runtime can hold, if the memory allocated to runtime data structures is fixed as discussed, or on the available memory the application can exploit, if the runtime footprint is allowed to grow. To handle this problem, when the producer task enters cutoff mode we activate a *cutoff stack frame*, which is the only that needs to be sized for worst-case recursive execution.

This effectively increases the speedup of fine-grained workloads generating a large number of tasks, as our experimental results show in the comparison between configurations with and without *cutoff* mode enabled (see Section 6.2).

6 Experimental Results

The key drawback of *untied* tasks is their large overheads. While such overheads can be tolerated by large applications exploiting coarse-grained tasks, this is usually not the case for embedded applications, which rely on fine-grained workloads. Our experimental results are largely aimed at studying this effect in-depth. Thus, we consider a set of synthetic benchmarks in which tasks are characterized by a tunable workload consisting of ALU operations (e.g., *add* on local registers).

The LINEAR benchmark consists of N identical tasks, each with a workload of W ALU instructions. The main task creates all the remaining $N-1$ tasks from a simple loop (one task created per loop iteration), then performs a `taskwait` to ensure that all tasks have completed execution.

The RECURSIVE benchmark generates the same N tasks by building a binary tree of depth L recursively. Each task creates two descendants, executes W ALU instructions, then waits on a `taskwait` directive for children execution completion.

The MIXED benchmark implements the worst-case execution pattern introduced in Section 4.2 and depicted in Figure 4.

The results of the experiments on synthetic benchmarks (from Section 6.1 to Section 6.3) show the speedup of parallel execution on a single Kalray-MPPA cluster (i.e., 16 cores) compared to the sequential execution of the same benchmark on a single core in the cluster. The sequential code does not rely on the OpenMP RTE for execution (it runs on the same hardware abstraction layer (HAL) on top of which all the OpenMP RTEs considered in this paper sit). Task granularity is reported on the x-axis (expressed in clock cycles), and it is roughly equivalent to the number of ALU operations that each task contains. Since Kalray does not provide distributed shared memory for inter-cluster communication, the execution times scale linearly with the number of clusters whenever the application has enough parallelism to exploit. Having these assumptions, we consider a single cluster without loss of generality.

Using these benchmarks we show i) a comparison between *tied* and *untied* tasks (Section 6.1); ii) the effect of *cutoff* policies (Section 6.2); iii) a comparison to other tasking RTEs, both from the embedded and from the HPC domains (Section 6.3).

For completeness, we also evaluate our design using a representative set of real-life workloads, the Barcelona OpenMP task suite (Section 6.4).

6.1 TIED vs UNTIED

Figures 7 and 8 show the speedups enabled by our *tied* and *untied* task design for the LINEAR and RECURSIVE benchmarks, respectively. The total number of generated tasks for both benchmarks is $N = 512$. We test both WFS and BFS policies.

Under the linear task generation pattern we observe no relevant difference between WFS and BFS, and between *tied* and *untied* tasks. Under the recursive generation pattern only *untied* tasks achieve the maximum speedup, when WFS is used. As we discussed in Section 4.2, *tied* tasks have zero speedup under WFS, and their speedup under BFS is limited as described by Formula 10. In this experiment we consider a binary tree ($k = 2$) being recursively explored using $t = 16$ threads. Applying the formula we get $S_{ideal} = 9$, which is confirmed by the experimental results.

The advantage of using *untied* tasks is particularly evident for the MIXED benchmark, which includes both linear and recursive task creation patterns. Figure 9 shows the results for this benchmark. From Formula 4.2 we can estimate the maximum speedup achievable for this pattern. Using *tied* tasks, 14 threads are allocated to execute the linear part of the application, 7 of which are blocked by the `taskwait` directive. The ideal speedup of the application is $2\times$, which our implementation reaches for granularities of around 10 Kcycles. Using *untied* tasks only 7 threads are allocated to the linear part, which brings the ideal speedup to $9\times$. The reason for a lower measured speedup of $8\times$ is

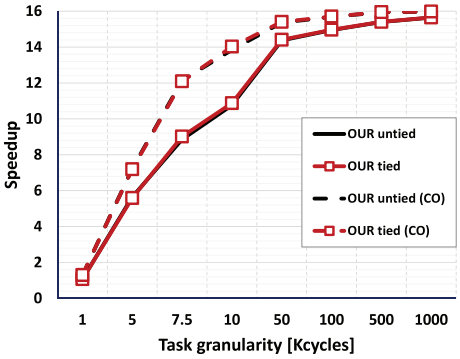

Fig. 10. LINEAR benchmark (with *cutoff*).

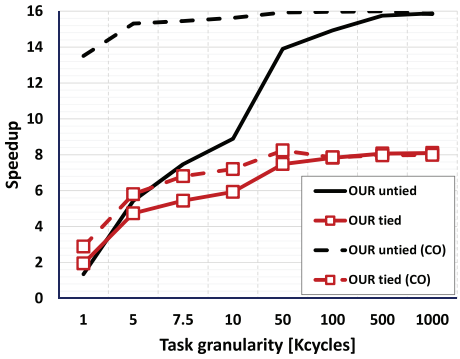

Fig. 11. RECURSIVE benchmark (with *cutoff*).

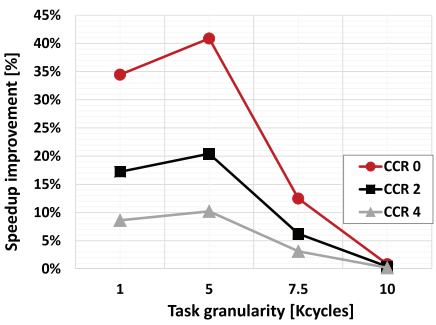

Fig. 12. Optimized *cutoff*. Speedup improvement for the LINEAR benchmark.

a limitation of the tracing (performance monitoring) of the Kalray platform. Since it is impossible to gather coherent timing if the task performing the measurement is allowed to migrate from one core to another, we were forced to declare the root node of the tree as *tied*.

This limits the maximum achievable speedup to $8\times$, which our *untied* tasks reach for granularities above 10 Kcycles. Overall, *untied* tasks enable up to $4\times$ faster execution than *tied* tasks for application featuring mixed generation patterns.

6.2 Cutoff

We repeated the experiments with LINEAR and RECURSIVE microbenchmarks considering a number of tasks equal to $N = 524288$ and a recursion depth of $L = 19$. This configuration saturates the runtime data structures and activates *cutoff* mode. Figure 10 and Figure 11 show the results for this new setup.

In the RECURSIVE benchmark the use of *cutoff* policies proves extremely beneficial, with nearly-ideal speedups for very fine-grained tasks (in the order of thousand cycles). The reason for this is that *cutoff* mode replaces costly task descriptor creation/management with much cheaper function calls from within a single thread. The benefits are clearly more relevant for ultra-fine-grained tasks, where the payload is minimal (i.e., most of the measured execution time is spent in runtime management routines). In our experimental setup *cutoff* mode is entered after the allocation of 512 tasks descriptors. By construction this happens after the execution of tasks at level $l = 8$ of the recursive tree, since applying Formula 2 we have $2^{l+1} - 1 = 511$. Consequently, the tasks belonging to subtrees rooted at level $l+1$ and with depth $L-(l+1) = 10$ are executed in *cutoff* state, each with a cumulative workload equal to $W_{cutoff} = W \cdot (2^{10} - 1)$. For any W in our setup, the results from the previous subsection show that W_{cutoff} is such that our RTE delivers near-ideal speedup.

As discussed in Section 5, *single-stack cutoff* proved an important optimization to avoid the inflation of runtime metadata (task stack conservative oversizing) to fit the worst-case application recursion pattern. Intuitively, in a memory-constrained system there is a correlation between reduction of runtime memory footprint and application speedup; the larger the application dataset that can be accommodated in memory, the coarser the parallel workloads we can outline and thus, the higher the speedups (overheads are amortized). Looking at Figure 10, it can be seen that a significant speedup increase is obtained if we increase task granularity in the range from 1000 to 10000 cycles. Considering a fully memory-bound task (e.g., a memory copy task), such an increase in granularity immediately translates into an increase in memory processed per task. Given a parallelization scheme (i.e., number of tasks), the amount of memory operations per task is constrained by the total amount of physical memory.

As the memory intensiveness of a task decreases, so does the potential speedup increase due to reduced runtime metadata footprint. This is well expressed by traditional computation to communication ratio (CCR) metrics:

$$CCR = \frac{\text{\# ALU operations}}{\text{\# LD/ST operations}} \tag{11}$$

Single-stack cutoff allows to significantly reduce the default task stack size (in our case, by three quarters; 1 KB to 256 bytes). This increases the available memory to application data by 30%. For a fully memory-bound task ($CCR = 0$), this corresponds to an increase in the task granularity of 30%. To model tasks with different CCR we built a synthetic benchmark with parametric and independent number of memory and ALU operations.

Figure 12 shows the speedup improvement when increasing the application memory by 30%. We consider different task granularities (x-axis) for the LINEAR benchmark, considering CCR equal to 0, 2 and 4. For heavily memory-bound tasks this optimization enables up to 40% speedup increase for tasks around 5 Kcycles.

6.3 Comparison with other tasking RTEs

We compare the performance of our tasking runtime to other various implementations, both from the HPC realm (where the tasking model has been originally proposed) and from the embedded domain.

6.3.1 HPC runtimes

In this section we compare tasking parallelization efficiency enabled by our RTE for embedded accelerators to what is available in the HPC domain with state-of-the-art RTEs and

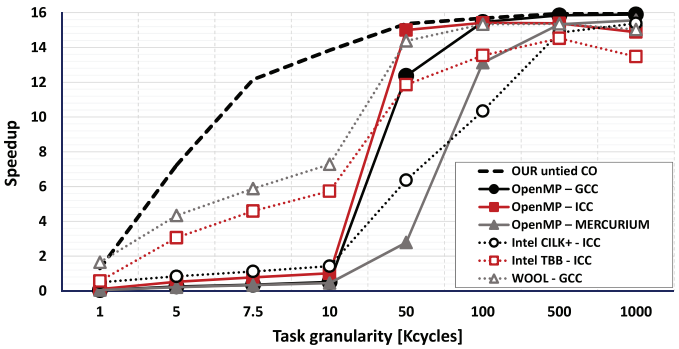

Fig. 13. Comparison to HPC tasking RTEs (LINEAR).

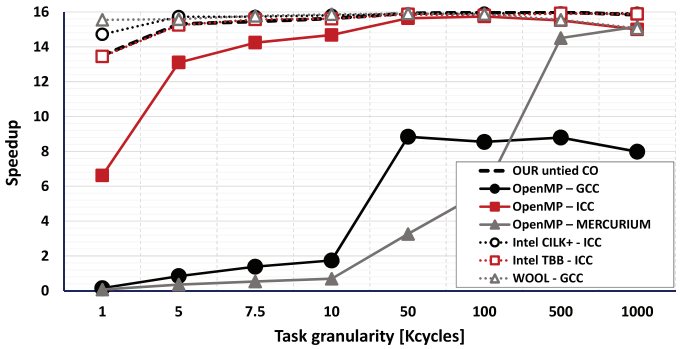

Fig. 14. Comparison to HPC tasking RTEs (RECURSIVE).

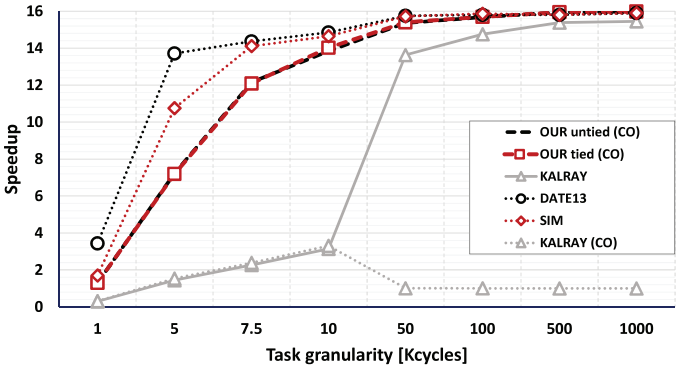

Fig. 15. Comparison to other embedded tasking RTEs. Speedup of the LINEAR benchmark (512 tasks, no *cutoff*).

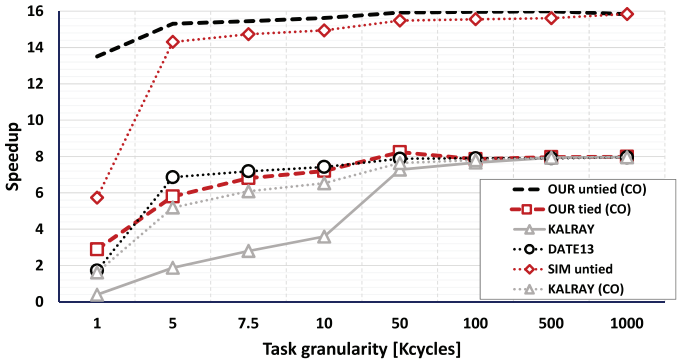

Fig. 16. Comparison to other embedded tasking RTEs. Speedup of the RECURSIVE benchmark (512 tasks, no *cutoff*).

platforms. In this context, we evaluate three OpenMP tasking implementations for HPC systems: *GNU GCC OpenMP*, *Intel ICC OpenMP* and *Mercurium/Nanos OpenMP*. In addition, we also consider three widespread HPC tasking models (other than OpenMP): *Intel CILK+*, *Intel TBB* and *Wool*[7] [19].

The LINEAR and RECURSIVE microbenchmarks have been used for this experiment, considering *untied* tasks and a BFS policy. The results for our RTE are measured on the Kalray platform, while those for other HPC RTEs are measured on a compute server equipped with two Intel Haswell with 8 cores @ 2.40 GHz. We use clock cycles as a performance metric to abstract away different machines' processor frequencies.

Figure 13 shows the results for the LINEAR pattern, where our runtime achieves up to 14× higher speedup compared to other OpenMP RTEs, for tasks below 10 Kcycles. The speedup improvements are very relevant also for non-OpenMP tasking RTEs, where in the same range of granularities we do 2× to 10× better. Overall, our runtime achieves nearly-ideal speedups for one order of magnitude smaller tasks, compared to HPC RTEs. It is very important to stress the relevance of this result. The coarse granularity of the tasks typically employed in HPC applications makes them way more tolerant to the huge overheads imposed by the tasking RTEs from this domain. The situation is very different for embedded applications, which feature much smaller tasks [21] [20], and which would be overwhelmed by such overheads. Reducing by an order of magnitude the granularity at which ideal speedups can be observed really makes *untied* tasks an effective parallelization abstraction for embedded accelerators.

Figure 14 show the results for the RECURSIVE pattern. It is evident that tasking was designed to handle the type of irregular parallelism that is found in such recursive patterns, as all the non-OpenMP solutions perform nearly on-par or slightly better than our runtime. This is a very significant result, as the OpenMP tasking model inherits several design restrictions from the original fork-join execution paradigm, which makes it impossible to streamline the implementation to what is done, for example, in Wool. Compared to other OpenMP solutions, our runtime clearly outperforms both GCC and Mercurium implementations, and achieves more than 2× higher speedups than Intel OpenMP for very small tasks.

6.3.2 *Embedded runtimes*

As a term of comparison from the embedded domain, we compare to two OpenMP tasking implementations. The first is the one provided with the original Kalray SDK (based on GCC). The second is the work from Burgio et al. [20], which we used as a starting point for our extensions to support *untied* tasks. The results published in [20] targeted a slightly different cluster-based architecture compared to the Kalray MPPA 256, and modeled after the STMicroelectronics STHORM architecture [45]. The key difference between the two platforms lies in the internal memory hierarchy, and specifically: i) while on the Kalray platform the cores within

7. *Wool* is the most lightweight tasking implementation for the general-purpose/HPC domain.

TABLE 1
Cost of a parallel construct (in cycles).

Platform	Kalray SDK (L1 cache)	OUR with L1 cache	OUR with L1 scratchpad
Cycles	85750	20500	1000

a cluster share a L2 memory that is accessed via cacheable (non-coherent) transactions in private L1 data caches, on the STHORM platform the cores do not have data caches, but explicitly access a shared L1 scratchpad memory (SPM); ii) accessing the shared memory on STHORM (L1 SPM, 1 cycle) is much faster than on Kalray (a miss in the L1 D$, 8 cycles). For a fair comparison, we also execute the experiments with our runtime on top of a cycle-accurate, SystemC-based virtual platform [46] that models the same architectural template assumed in [20]. The results for this configuration are labeled as *SIM* in Figures 15 and 16. The results for Burgio et al. [20] are labeled as *DATE2013*.

The results show that the original Kalray runtime can achieve near-ideal speedups tasks larger than 100 Kcycles. For smaller tasks the maximum achievable speedup is $3\times$. In this fine-grained task region our runtime consistently achieves $4\times$ higher speedup. The limitations to the speedup for the RECURSIVE benchmark are consistent with our previous discussion about *tied* tasks (the only ones supported by the original Kalray runtime). It is also worth noting that *cutoff* mode is not properly supported for LINEAR patterns in the original Kalray runtime. Enabling *cutoff* mode in this configuration simply seems to disable parallelism completely.

Compared to *DATE2013*, our runtime performs only slightly worse in the very fine-grained task region (around 5 Kcycles), since this implementation was optimized to support *tied* tasks on this specific architecture. Beyond that point our implementation is equivalently efficient.

Figure 16 reports the values of both simulations for the RECURSIVE benchmark. In this case the comparison with Burgio et al is not meaningful, as the implementation of *tied* support is limited by the execution of direct descendants. It is noteworthy that our solution could perform even better on architectures providing a L1 scratchpad – that can be explicitly managed to host the key RTE data structures – in place of the non-coherent, two-level cache hierarchy available on Kalray. Table 1 gives an insight of this effect by depicting the cost of a `parallel` construct (i.e., the time to open and close an empty parallel region). Our implementation running on Kalray is $4\times$ faster than that provided by the original Kalray OpenMP RTE. If we, however, run our implementation on the virtual platform modeling the scratchpad-based clusters of STHORM, we observe a $20\times$ faster execution of the `parallel` directive, compared to the original Kalray OpenMP RTE running on the Kalray platform. This huge difference is motivated by different access time to the RTE control data structures. On STHORM, or any other similar architecture featuring shared L1 SPM, reading or writing such data structures is very cheap (typically one cycle). On Kalray, due to the lack of a coherency protocol for the L1 data caches, both the RTEs (our RTE and the original Kalray RTE) force costly line flushes every time that a thread updates a control data structure[8].

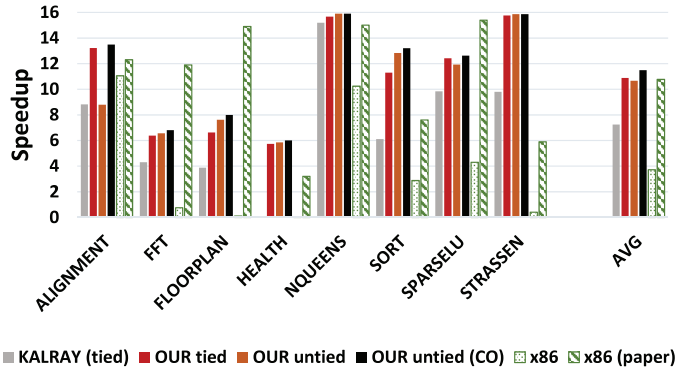

Fig. 17. Speedup of the Barcelona OpenMP Task Suite benchmarks executed on three RTEs with different configurations: Kalray (tied tasks), OUR (tied, untied and untied with *cutoff*) and x86 libgomp (tied tasks with restricted or full dataset).

6.4 Barcelona OpenMP Task Suite

To assess the performance of our runtime on applications that are typical of the HPC domain, we execute a set of benchmarks from the Barcelona OpenMP Task Suite (BOTS) [23], which includes real-life applications parallelized with OpenMP tasks. Due to the limitation on local shared memory, we scaled the datasets to fit the available space.

Figure 17 shows the speedup of applications for different configurations, comparing the Kalray SDK ("KALRAY" bars) with different configurations of our runtime, using tied tasks ("OURS tied" bars), untied tasks ("OURS untied" bars) and untied tasks with *cutoff* ("OURS untied CO" bars). On average, our runtime achieves a parallelization speedup of $11.50\times$, which is $\approx 60\%$ faster than the Kalray RTE (which achieves on average $7.25\times$ parallelization speedup).

The improvement of untied w.r.t. tied tasking is quite limited in most cases. This result is due to the low degree of recursion exposed by these applications, with the exception of SORT and FLOORPLAN. NQUEENS is another application in this suite adhering to a recursive pattern, but it is characterized by coarse grain tasks and consequently the advantages of the untied model are limited (as discussed in previous section).

The benefits of *cutoff* are minimal, since the bottleneck is limited parallelism in the application rather than runtime overhead. The marginal improvements, where present, are usually due to better memory usage (*untied* tasks in *cutoff* use less memory for the runtime, which is used for application data instead).

Figure 17 also includes the results reported by Duran et al. [23] for BOTS execution on x86 platforms ("x86 (paper)" bars). To perform a fair comparison, we executed on a x86 machine a set of experiments using the reduced dataset used for the Kalray platform ("x86" bars). Compared to the "x86" set, the average improvement of our runtime is $3\times$.

8. Note that bypassing the L1 caches to directly access the shared L2 memory is not possible on Kalray MPPA 256. Even if that was possible, however, it would have almost ten times higher cost.

7 Conclusion

Task-based parallelism has the potential to provide efficient exploitation of embedded PMCAs, 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 the Kalray MPPA 256 (and similar cluster-based PMCAs). 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 PMCAs. When compared to OpenMP implementation for high performance computing systems, our design achieves near-ideal speedups for one order of magnitude smaller tasks. We observe an average parallelization speedup of 12× for real benchmarks, which is ≈ 60% higher than what we measure from the original Kalray OpenMP implementation.

References

[1] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving Exascale Capabilities through Heterogeneous Computing," *IEEE Micro*, vol. 35, no. 4, pp. 26–36, July 2015.

[2] P. Vogel, A. Marongiu, and L. Benini, "Lightweight Virtual Memory Support for Zero-Copy Sharing of Pointer-Rich Data Structures in Heterogeneous Embedded SoCs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1947–1959, July 2017.

[3] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel *et al.*, "Invasive manycore architectures," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 2012, pp. 193–200.

[4] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips, "Workload Partitioning for Accelerating Applications on Heterogeneous Platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2766–2780, Sept 2016.

[5] E. Hwang, S. Kim, T. k. Yoo, J. S. Kim, S. Hwang, and Y. r. Choi, "Resource Allocation Policies for Loosely Coupled Applications in Heterogeneous Computing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2349–2362, Aug 2016.

[6] S. Shudler, A. Calotoiu, T. Hoefler, and F. Wolf, "Isoefficiency in Practice: Configuring and Understanding the Performance of Task-based Applications," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 131–143.

[7] E. Ayguade, 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, March 2009.

[8] S. Zhu, S. Chandrasekaran, P. Sun, B. Chapman, M. Winter, and T. Schuele, "Exploring Task Parallelism for Heterogeneous Systems Using Multicore Task Management API," in *European Conference on Parallel Processing*. Springer, 2016, pp. 697–708.

[9] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture," in *International Workshop on OpenMP*. Springer, 2014, pp. 202–214.

[10] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini, "Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 957–967, Aug 2015.

[11] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *2009 IEEE International Symposium on Parallel Distributed Processing*. IEEE, May 2009, pp. 1–8.

[12] R. Vargas, E. Quinones, and A. Marongiu, "OpenMP and Timing Predictability: A Possible Union?" in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. EDA Consortium, 2015, pp. 617–620.

[13] L. M. Pinho, V. Nlis, P. M. Yomsi, E. Quiones, M. Bertogna, P. Burgio, A. Marongiu, C. Scordino, P. Gai, M. Ramponi, and M. Mardiak, "P-SOCRATES: A parallel software framework for time-critical many-core systems," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1190 – 1203, 2015.

[14] M. A. Serrano, A. Melani, M. Bertogna, and E. Quinones, "Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 1066–1071.

[15] M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quinones, "An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling," in *Real-Time Distributed Computing (ISORC), 2017 IEEE 20th International Symposium on*. IEEE, 2017, pp. 193–202.

[16] M. Damschen, L. Bauer, and J. Henkel, "Timing analysis of tasks on runtime reconfigurable processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 294–307, 2017.

[17] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset, "Bridging the Gap Between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2794–2807, Oct 2017.

[18] A. Podobas and S. Karlsson, "Towards Unifying OpenMP Under the Task-Parallel Paradigm," in *International Workshop on OpenMP*. Springer, 2016, pp. 116–129.

[19] K.-F. Faxén, "Wool-A Work Stealing Library," *SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 93–100, June 2009.

[20] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, "Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. EDA Consortium, 2013, pp. 1504–1509.

[21] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. ACM, 2007, pp. 162–173.

[22] "Kalray MPPA products," http://www.kalray.eu, last accessed: 2017-10-19.

[23] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *2009 International Conference on Parallel Processing*. IEEE, Sept 2009, pp. 124–131.

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. ACM, pp. 207–216.

[25] C. E. Leiserson, "The Cilk++ concurrency platform," in *46th ACM/IEEE Design Automation Conference (DAC'09)*. IEEE, 2009, pp. 522–527.

[26] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[27] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)*. IEEE, 2009, pp. 1–12.

[28] K. Sakamoto and T. Furumoto, "Grand central dispatch," in *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 2012, pp. 139–145.

[29] "OpenMP 4.0 Specification," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[30] A. Podobas, M. Brorsson, and K.-F. Faxén, "A comparative performance study of common and popular task-centric programming frameworks," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 1, pp. 1–28, 2015.

[31] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, "Design and Implementation of OpenMP Tasks in the OMPi Compiler," in *2011 15th Panhellenic Conference on Informatics*. IEEE, Sept 2011, pp. 265–269.

[32] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *International Workshop on OpenMP*. Springer, 2008, pp. 100–110.
[33] A. Duran, J. Corbalan, and E. Ayguade, "An adaptive cut-off for task parallelism," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov 2008, pp. 1–11.
[34] C. Wang, S. Chandrasekaran, B. Chapman, and J. Holt, "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*, ser. PMAM '13. ACM, 2013, pp. 83–92.
[35] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault, *OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip*. Springer Berlin Heidelberg, 2013, pp. 114–127.
[36] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, July 2013, pp. 180–187.
[37] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quinones, "A lightweight OpenMP4 run-time for embedded systems," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, Jan 2016, pp. 43–49.
[38] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "KAHRISMA: a novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 819–824.
[39] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: rotating instruction set processing platform," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 791–796.
[40] M. Dehyadegari, A. Marongiu, M. R. Kakoee, S. Mohammadi, N. Yazdani, and L. Benini, "Architecture support for tightly-coupled multi-core clusters with shared-memory HW accelerators," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2132–2144, 2015.
[41] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing Characterization of OpenMP4 Tasking Model," in *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '15. IEEE Press, 2015, pp. 157–166.
[42] D. Novillo, "Openmp and automatic parallelization in gcc," *the Proceedings of the GCC Developers Summit*, 2006.
[43] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: a Research Compiler for OpenMP," in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004, p. 56.
[44] C. D. Marlin, *Coroutines: a programming methodology, a language design and an implementation*. Springer Science & Business Media, 1980, no. 95.
[45] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications," in *DAC Design Automation Conference 2012*. IEEE, June 2012, pp. 1137–1142.
[46] D. Bortolotti, A. Marongiu, and L. Benini, "VirtualSoC: a research tool for modern MPSoCs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 1, p. 3, 2016.

Giuseppe Tagliavini received the MSc degree in computer engineering in 2010 and the PhD degree in electronic engineering in 2017 from the University of Bologna, Bologna, Italy. His current research interests include programming models and run-time optimization for many-core embedded accelerators, software design for high-performance embedded systems, and compiler support for emerging computing architectures.

Daniele Cesarini received the MS degree in computer engineering from the University of Bologna, Italy, in 2014, where he is currently a PhD student in the Department of Electrical, Electronic and Information Engineering (DEI). His research interests concern parallel programming models and middleware for embedded and High Performance Computing (HPC) systems, with special emphasis on thermal-aware task scheduling based on linear optimization techniques.

Andrea Marongiu received the MSc degree in electronic engineering from the University of Cagliari, Italy, in 2006 and the PhD degree in electronic engineering from the University of Bologna, Italy, in 2010. Since 2013 he has been a Research Fellow at ETH Zurich. He currently is an Assistant Professor at the University of Bologna. His research interests concern parallel programming model and architecture design in the single-chip multiprocessors domain, with special emphasis on compilation for heterogeneous architectures, efficient usage of on-chip memory hierarchies and SoC virtualization. He has published more than 80 papers in peer reviewed international journals and conferences. He is a member of the IEEE.