Enabling Fine-Grained OpenMP Tasking on Tightly-Coupled Shared Memory Clusters

Paolo Burgio, Giuseppe Tagliavini, Andrea Marongiu, Luca Benini
DEIS – Universitá degli Studi di Bologna – Viale Risorgimento 2, 40136 Bologna – Italy
Email: {paolo.burgio, giuseppe.tagliavini, a.marongiu, luca.benini}@unibo.it

***Abstract*—Cluster-based architectures are increasingly being adopted to design embedded many-cores. These platforms can deliver very high peak performance within a contained power envelope, provided that programmers can make effective use the available parallel cores. This is becoming an extremely difficult task, as embedded applications are growing in complexity and exhibit irregular and dynamic parallelism. The OpenMP tasking extensions represent a powerful abstraction to capture this form of parallelism. However, efficiently supporting it on cluster-based embedded SoCs is not easy, because the fine-grained parallel workload present in embedded applications can not tolerate high memory and run-time overheads. In this paper we present our design of the runtime support layer to OpenMP tasking for an embedded shared memory cluster, identifying key aspects to achieving performance and discussing important architectural support to removing major bottlenecks.**

I. Introduction

The multi-core design paradigm has been succesfully adopted since 2005 to attack the technology walls hindering Moore's law predictions and is currently in the *many-core* era, where hundreds of simple processing units (PU) are integrated on a single chip. To overcome the scalability bottlenecks encountered when interconnecting such a large amount of PUs, some recent embedded many-core accelerators leverage tightly-coupled *clusters* as a building block. Examples are Plurality's HyperCore Architecture Line (HAL) processors [1], ST Microelectronics STHORM [2], or massively data-parallel architectures such as GP-GPUs [3]. These products consider a hierarchical design, where PUs are grouped into small-medium sized subsystems (clusters) sharing high-performance local interconnection and memory, while scaling to larger system sizes is enabled by replicating clusters and interconnecting them with a scalable medium like a NoC.

While similar architectures can theoretically achieve tremendous Gops/watt targets, the burden of extracting this peak performance is nowadays mostly demanded to the software layer. Efficient programming abstractions (programming models, compilers, runtime systems) are paramount to achieving efficient exploitation of many-cores. In particular, modern embedded applications are increasing in complexity and often expose high degree of parallelism which is irregular in nature and/or dynamically generated. The *tasking* execution model represents a powerful abstraction to exploiting this kind of parallelism, as it enables asynchronous, dynamic creation of units of work in a simple and straightforward manner. Notable examples of this programming paradigm include Cilk [4], Apple Grand Central Dispatch [5], Intel Carbon [6] or the current OpenMP specification [7].

The tasking abstraction provides a powerful conceptual framework to exploit irregular parallelism in target applications, but its practical implementation requires sophisticated runtime system support, which typically implies important space and time overheads. The applicability of the approach is thus often limited to applications exhibiting units of work which are coarse-grained enough to amortize these overheads. While this is often the case for general-purpose systems and associated workloads, things are different when considering embedded many-core accelerators. Miminizing runtime overheads is thus a primary challenge to enjoy the benefits of tasking on these systems.

978-3-9815370-0-0/DATE13/

In this paper we describe the design of an optimized runtime environment supporting the OpenMP tasking model on an embedded shared-memory cluster. We identify the key aspects critical to performance and explore architectural (HW) support to minimizing the effect of major bottlenecks implied by the execution model. We validate our work on a cycle accurate virtual platform simulating the target cluster and the proposed architectural variants, discussing in details how our optimizations make tasking a suitable programming abstraction also in presence of very fine-grained workloads.

The paper is structured as follows. We discuss related work in Sec. II and the target architectural template in Sec. III. The design and implementation of OpenMP tasking is described in Sec. IV. Finally, we validate our approach and characterize the performance of our implementation in Sec. V, then summarize our main findings and discuss future work in Sec. VI.

II. Related work

The tasking (a.k.a. *work-queue*) programming model is a well known paradigm in the domain of general-purpose computing and in last decade it has been successfully adopted on several multi-core architectures. Cilk [4], Intel Carbon [6], Apple Grand Central Dispatch [5] and OpenMP [7] are successful technologies embodying this model. Recently, some attempts were made to explore its applicability also to heterogeneous systems (i.e., CPU + GPU). The most representative example in this sense is the Fusion series from AMD [8], where a centralized queue system coupled to a task-based programming model enables distributed dispatching of work units between a generic (x86) CPU and a GPU-like accelerator. Programming effort is anyhow significant, since task execution and data transfers must be manually orchestrated using OpenCL.

Currently, there are several freely available open source implementation of the OpenMP 3.x specifications [9], [10], [11]. The GCC-OpenMP (GOMP) framework [11] implements tasking on top of *pthreads*. The overheads implied by such a layer are significant, as evidenced by many researchers [9][10].

The cited works target general-purpose computing, using lightweight threading libraries to ensure portability and efficiency. However, embedded platforms are typically more resource-constrained than general-purpose systems, thus requiring different design choices for the implementation of tasking. For example, Ayguadé et al.[12] consider tasks with a duration of 10μs (which considering their 1.67 GHz cores and assuming a CPI of 1 translates in 16K cycles). Similarly, Kumar et al. [6] consider an average of 5K clock cycles for fine-grained tasks. Agathos et al. [9] can afford a 4MB stack for their threads. Clearly, all these numbers need to be significantly scaled down when considering embedded applications and the hardware they run on.

To the best of our knowledge, we are the first to implement OpenMP tasks on an embedded shared memory cluster.

III. Architecture

The reference architecture is inspired by the tightly-coupled clusters in ST Microelectronics STHORM [2]. In Figure 1 we show a simplified block diagram of a cluster composed

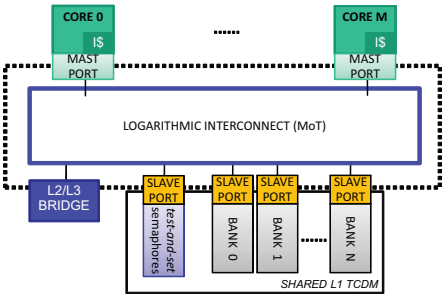

Fig. 1. On-chip shared memory cluster template

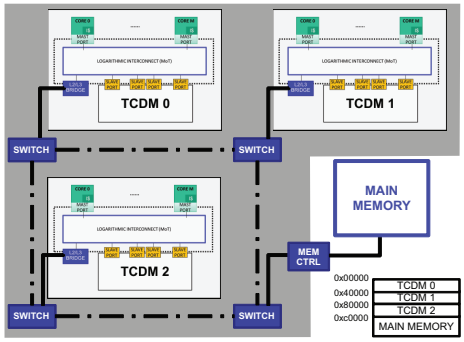

Fig. 2. Multi-cluster architecture and global address space

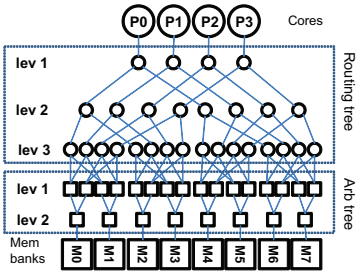

Fig. 3. Mesh of trees 4x8

of (up to) 16 RISC-32 processors connected through a low-latency, high bandwidth logarithmic interconnect similar to the ones proposed by Plurality LTD [1] or Rahimi [13]. The logarithmic interconnect is built as a parametric, fully combinational Mesh-of-Trees (MoT) (see Figure 3). Processors communicate through a fast multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM), which is configured as a shared, software-managed scratchpad memory. The number of ports and banks is a multiple of the number of processors to increase bandwidth. In case there are no bank conflicts, concurrent accesses by multiple cores to the TCDM are served simultanously by the MoT. Bank conflicts result in a higher latency, due to contention, which is resolved based on round-robin arbitration. The crossing latency of the MoT is one clock cycle, and word interleaving enables fast concurrent accesses to adjacent memory locations. As a consequence, conflict-free TCDM accesses have two-cycle latency. The interconnection supports read-broadcast: when multiple processors read the same memory location at the same time all the requests are serviced in two cycles.

The L1 scratchpad (TCDM) has limited size of 256KB, thus program code and most of the data are typically stored in larger L2 or L3 memory, while the content of the TCDM is manually updated to the most referenced subset of data at any time. A cluster thus features a L2/L3 bridge for communication with the outer world. In this work we consider a two-level memory system, with an off-cluster main memory, and we assume a global address space. Scaling to larger system sizes with this architectural template is achieved by interconnecting several clusters through a NoC as shown in Figure 2 (see [2]). However, in this paper we consider a single cluster and leave exploration of multi-cluster for future work.

Synchronization among the processors is achieved through a segment of the local TCDM address space featuring *test-and-set* semantics. As we will explain in Section IV, the way the test-and-set memory is physically implemented has a big impact on the performance of our tasking support. In the following we will thus describe different implementations.

IV. Design and Implementation

OpenMP 3.0 introduces a task-centric model of execution. The new **`task`** construct can be used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. When a thread encounters the **`task`** construct, it prepares a task *descriptor* consisting of the code to be executed, plus a data environment inherited from the enclosing structured block. **`shared`** data items point to the variables with the same name in the enclosing region. New storage is created for **`private`** and **`firstprivate`** data items, and the latter are initialized with the value of the original variables at the moment of task creation. The execution of the task can be immediate or deferred until later by inserting the descriptor in a *work queue* from which any thread in the team can extract it. This decision can be taken at runtime depending on resource availability and/or on the scheduling policy implemented (e.g., breadth-first, work-first [10]). However, a programmer can enforce a particular task to be immediately executed by using the **`if`** clause. When the conditional expression evaluates to *false* the encountering thread suspends the current task region and switches to the new task. On termination it resumes the previous task. Specifications also enable work-unit based synchronization. The **`taskwait`** directive forces the current thread to wait for the completion of every tasks generated from the current task region. *Task scheduling points* (TSP) specify places in a program where the encountering thread may suspend execution of the current task and start execution of a new task or resume a previously suspended task.

Figure 5 shows our layered approach to designing the primitives for the tasking constructs. These constructs are

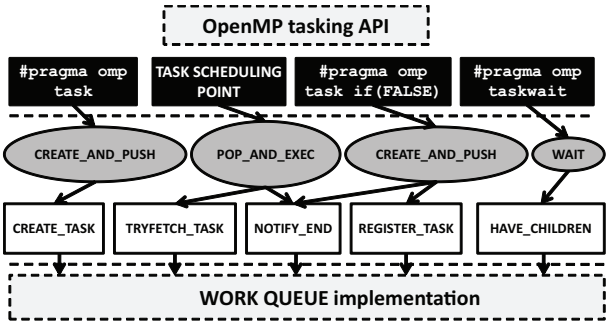

Fig. 5. Design of tasking support

depicted in the top layer blocks (in black). To manage OpenMP tasks we rely on a main *work queue* where units of work can be pushed to and popped from (bottom layer block). The gap between OpenMP directives and the *work queue* is bridged by an intermediate runtime layer (gray blocks), which operates on the queue through a set of basic primitives (white blocks) to implement the semantics of the tasking constructs.

A. Design of the work queue

Our design relies on a centralized queue with breadth-first, LIFO scheduling. Tasks are tracked through *descriptors* which identify their associated *task regions* and which are stored in the *work queue*. The two basic operations on the queue are task insertion and extraction. Inserting a task has two effects: i) creating a new descriptor for it, and ii) registering it as a child of the executing task (its *parent*). We formalize these semantics as a primitive that we call **`CREATE_TASK`**.

Extracting a task from the *work queue* retrieves its descriptor for execution. To this aim we consider a **`TRYFETCH_TASK`** primitive, which returns the task descriptor in case of successful extraction, or a NULL pointer if the *work queue* is empty. Task extraction should only return the descriptor to the caller, not detach it from the *work queue* until the task has completed execution. This is necessary for correctly

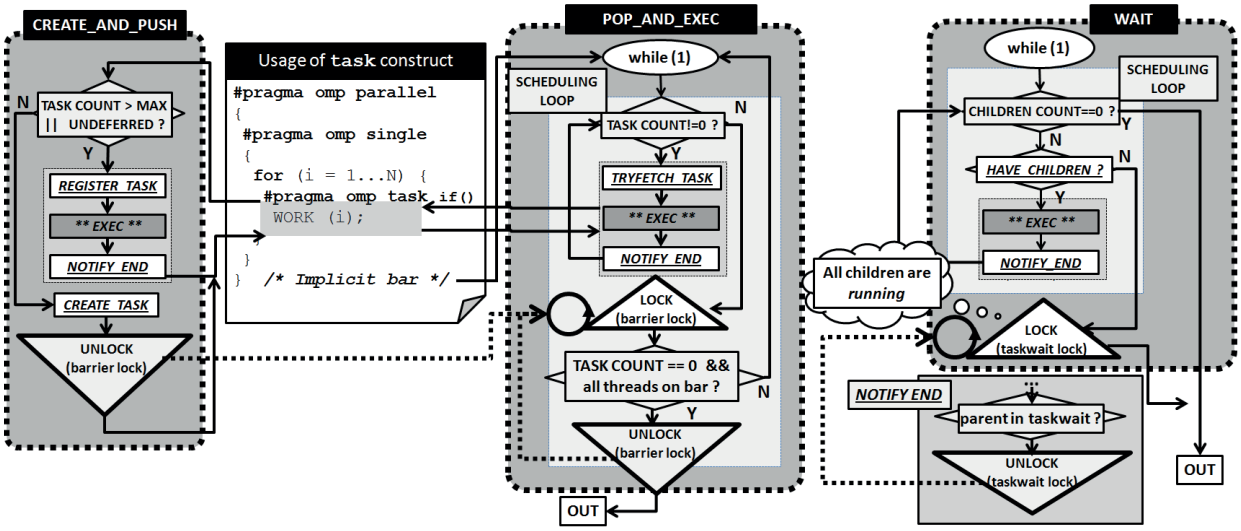

Fig. 4. Design of task scheduling loop

supporting synchronization (**taskwait**). We thus designed a separate **NOTIFY_END** primitive to dispose of the descriptor, which acts as an epilogue to task execution.

Note that since the **TRYFETCH_TASK** primitive does not remove the task *descriptor* from the *work queue*, it is necessary to mark it as *running* to avoid multiple extractions of the same *descriptor*. Thus, the **CREATE_TASK** inserts a *waiting* task in the *work queue* and the **TRYFETCH_TASK** changes its status to *running*. **NOTIFY_END** marks it as *ended*.

To support undeferred tasks (e.g., whose **if** condition is evaluated to *false*) we introduce a **REGISTER_TASK** primitive which inserts a *descriptor* marked as *running*.

Finally, the **HAVE_CHILDREN** primitive allows to determine if a task has children not yet assigned to a thread (i.e., in the *waiting* state). As we will explain in the next section, this is necessary to implement task switching capability in presence of a **taskwait**.

B. Design of the runtime layer

Let us consider the simple example of the **task** construct in the code snippet of Figure 4. The **parallel** directive creates a team of worker threads, then only one of them executes the **single** block. This thread acts as a work producer, since it is the only one encountering the **task** construct. The control flow for the rest of the threads falls through the parallel region to the implied barrier at its end.

The most important part of the implementation of the tasking execution model is Task Scheduling Points (TSP). Parallel threads are allowed to switch from one task to another:

1) at **task** constructs;
2) at implicit and explicit barriers;
3) at the end of the current task;
4) at **taskwait** constructs;

The first point prevents system oversubscription in cases where a thread is required to generate a very high number of tasks (e.g., the **task** directive is nested inside a loop with a huge number of iterations). Placing a TSP on a **task** construct allows the producer thread to switch to executing some of the tasks already in the queue. Task creation is resumed once the queue has been depleted to a certain level.

To keep the implementation of task scheduling as simple as possible we deal with this issue in the following manner. Upon encountering a **task** directive, threads calls the **CREATE_AND_PUSH** runtime function, depicted on the left part of Figure 4. Here, the caller first checks for the number of tasks already in the queue. If this number exceeds a given threshold the thread does not insert the task in the queue, but it immediately executes it instead. Note that this can not be implemented through a simple jump to the task block code. Executing a task without creating a descriptor and connecting it to the others will in fact result in ignoring its existence, which may lead to incorrect functioning of the **taskwait** directive due to bad internal representation of the task hierarchy. Thus we create and insert in the queue a descriptor for a *running* task through the **REGISTER_TASK** primitive. Similarly, we signal task execution termination through a call to **NOTIFY_END**.

This same solution is adopted when an undeferred task is explicitly generated by the user through the **if(FALSE)** clause. In all the other cases, a call to **CREATE_AND_PUSH** will result in regular creation of a team descriptor and insertion in the queue (**CREATE_TASK**). After that, the producer thread signals the presence of work in the queue by releasing a *barrier_lock* on which consumer threads wait.

This brings us to the second TSP. As explained before, threads not executing the **single** block are trapped on the barrier implied at the end of the region. This is implemented through a call to the **POP_AND_EXEC** function (central part of Figure 4). Here, threads first check for the presence of tasks in the queue. If there are tasks available the encountering thread initiates an execution sequence. First, the task descriptor is extracted from the queue with the **TRYFETCH_TASK** primitive. Then, the associated task code is executed. Finally, notification of task completion is signaled through the **NOTIFY_END** primitive. If the queue is empty, the encountering thread busy waits on the *barrier_lock* (note that this lock is initialized as *busy* at system startup). When the lock is released by a producer pushing a task in the queue, the current thread checks for the presence of tasks in the queue and for the number of threads waiting on the lock (annotated in a counter). If all threads are on the lock and there are no tasks in the queue, this indicates that the end of the parallel region has been reached. Otherwise, there may still be work left to do, so the thread jumps back to the scheduling loop.

Note that upon task termination we execute again an iteration of the scheduling loop, thus implementing the third TSP.

Finally, a TSP is also implied at a **taskwait** construct. However, in this specific case the *Task Scheduling Constraint* only allows to switch execution to a task that was directly created by the current one to prevent deadlocks. We implement this semantics in the **WAIT** runtime function. Each task keeps track of its children. The **HAVE_CHILDREN** primitive allows to fetch the descriptor of a child task in the *waiting* state. If a valid task descriptor is returned, the thread can be rescheduled

on that task. Otherwise, all the children are in the *running* state and the thread will have to stay idle waiting for their completion. In this case, the last terminating child notifies the parent through the **`NOTIFY_END`** primitive.

C. Implementation details

Task descriptors are interconnected within two co-existing data structures; a *queue* and a *tree*. The *queue* contains the descriptors of all the tasks in the *waiting* state for the current parallel region and it is implemented as a doubly-linked list. Similarly, to build the *tree* representation, each descriptor handles a reference to a doubly-linked list of children, i.e., the set of tasks that it has previously created, being either in the *waiting* or *running* state. Each descriptor also traces the *parent* task. Upon task creation, the corresponding descriptor is inserted into the *work queue* by updating the connections in the *queue* and in the *tree* data structures.

Consistency between the two representations is enforced by making their updates atomic through a *work_queue_lock*. Each of the insertion/removal primitives protects the critical sections that update the descriptor with this lock. We have manually optimized the assembly for the primitives to minimize the duration of critical sections.

To ensure fast access to task descriptors, we store them in L1 memory. However, the number of tasks co-existing in the system can become very high, thus we need a mechanism to avoid memory oversubscription. We implemented a custom allocator which uses a fixed number (1024) of statically reserved bins in a region of the TCDM. The **`task_malloc`** and **`task_free`** retrieve and dispose memory for a new descriptor using a LIFO list of free bins. Concurrent write accesses to the list are protected by a lock (*task_malloc_lock*). There are four main types of locks in our tasking support framework. Besides *task_malloc_lock* and *work_queue_lock*, the *barrier_lock* is used inside the task scheduling loop for idle threads to wait for available tasks and the *taskwait_lock* is used by a task to wait for termination of its children.

Note that there is one *work_queue_lock* and one *barrier_lock* for each parallel region in the system, while there is a *taskwait_lock* for every task in the system. In fact, the number of locks used at any time can be high. As a consequence, the single-ported test-and-set (TAS) memory may easily become a bottleneck if multiple threads are concurrently performing any form of synchronization.

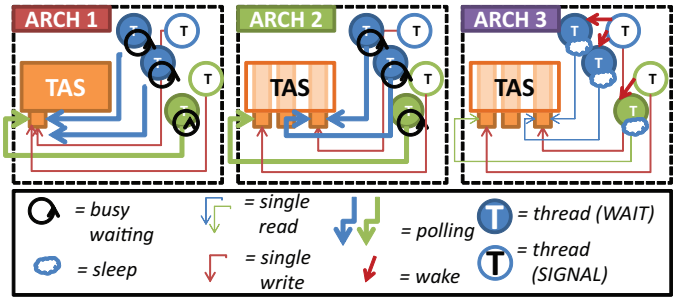

Fig. 6. Different architectural configurations (*ARCH*) of the TAS memory

To address this issue we consider the following architectural variants (shown in Figure 6). We refer to the implementation with single-ported, single-banked TAS memory as **ARCH 1** and consider it as a baseline for the other solutions. Any *wait* operation in this architecture is always implemented with busy-waiting (see leftmost part of the figure). As the number of locks increases, the concurrent traffic overloads the TAS port. However, in many cases the conflict is created by contention for the memory port, not for a lock. This issue can be mitigated by considering an architectural modification to increment the number of ports and banks of the TAS memory. In this variant (referred to as **ARCH 2**) the TAS segment has 16 banks/ports and thus, similar to the data TCDM segment, can serve concurrent accesses to different locks in parallel.

In both **ARCH 1** and **ARCH 2** all of the *wait* operations are implemented with busy-waiting on the lock until the corresponding *signal* operation FREEs it. While **ARCH 2** solves the issue of sequentialization of accesses to distinct locks, it does not remove the polling activity of multiple cores, which creates congestion. While *work_queue_lock* and *task_malloc_lock* are used to implement critical sections protecting atomic queue updates, *barrier_lock* and *taskwait_lock* implement a different synchronization pattern, where one thread (or more) is waiting for another one (or more) to notify verification of a specific event. Thus, while in the first case a busy-waiting implementation is to be preferred (short duration of the critical section), in the second case we could rather consider an alternative idle/wake mechanism where threads that find a busy lock enter a sleep state and will be awaken after the lock has been set to FREE. We refer to this architectural variant as **ARCH 3**.

V. Experiments

To validate our design we performed an extensive set of experiments using a SystemC-based virtual platform modeling the tightly-coupled cluster described in Section III [14]. Table I summarizes the main architectural parameters, a typical setup for the considered platform template (see [2]).

TABLE I

ARM v6 cores	16	TCDM banks	16
I$ size	1 KB	TCDM latency	$\geq$ 2 cycles
I$ line	4 words	TCDM size	256 KB
t_{hit}	= 1 cycle	L3 latency	$\geq$ 60 cycles
t_{miss}	$\geq$ 59 cycles	L3 size	256 MB

We implemented the tasking support on top of a runtime [15] optimized for the target platform. We present three types of experiments:

1) cost characterization of the main tasking constructs;
2) parallelization speedup for varying task granularity and comparison with other tasking implementations;
3) parallelization speedup for two real programs: the Strassen matrix multiplication benchmark and the FAST corner detection application.

A. Tasking cost characterization

We measured the cost of the OpenMP tasking services presented in Section IV (top layer of Figure 5, in black). We create 16 threads, one per processor, then force one single thread to produce 256 tasks. The tasks are composed of ALU instructions only, to exclude memory effects from the measurement (each task consists of 500 ALU operations).

Figure 7 shows the results for these measurements for each of the three architectural variants discussed in the previous section. There is one additional bar per plot, labeled IDEAL, which shows the cost for executing the corresponding runtime operation on a single core, while the rest of the cores is idling (thus no interference of any kind takes place). These experiments are run under architectural variant **ARCH 1**.

A first observation is that the cost for tasking in the IDEAL case is between 70 and 130 clock cycles. The optimized assembly routines allow a low cost for these services.

Figure 7 (a) shows the cost for creating a task with the **`task`** directive. Contributions include time for creating the descriptor (*task_malloc*) and initializing it (*desc init*), *work_queue_lock* acquisition (*lock*) and release (*unlock*), plus update of the internal work queue data structures (*update wq*). Results for **ARCH 1** show that the duration of the *lock* and the *unlock* phases are greatly impacted by high contention for the single-ported TAS memory, as well as task descriptor creation and initialization (in which some locks are accessed). When moving to **ARCH 2** most of this effect disappears

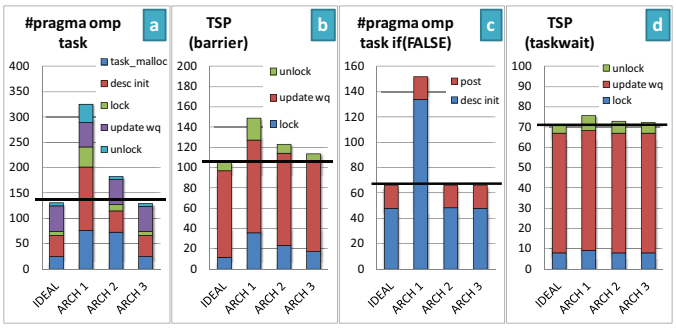

Fig. 7. Breakdown of the time spent in high-level OpenMP services

as expected. **ARCH 3** further improves the results, since the polling traffic for threads waiting on the *barrier_lock* is removed, thus reaching the IDEAL performance.

Figures 7 (b), (c) and (d) report similar cost results respectively for i) a Task Scheduling Point occurring on implicit and explicit *barriers* ii) creating an undeferred task (annotated with a **`if(FALSE)`** clause) iii) a Task Scheduling Point occurring on a **`taskwait`**. Similar conclusions hold for the benefits of **ARCH 2** and **ARCH 3** over **ARCH 1**. Note that for undeferred tasks we need not acquire a lock since the descriptor is in a local variable, which also removes the need for **`task_malloc/free`**. The *post* cost refers to switching execution back to the calling context.

B. Task granularity impact on speedup

Figure 8(a) shows how different task granularities affect speedup for each of the three architectures. For this characterization we consider a synthetic benchmark consisting of a loop with a parameterizable number of iterations (*GR*) and whose body contains one dummy ALU (MOV) instruction.
We consider the same setup of the previous experiment, with 16 threads, where only one is responsible for the creation of 256 tasks while the remaining 15 can immediately start to execute them (the producer thread can also join task execution after creating them all). We perform experiments for task granularities (*GR*) varying in the range between 1 and 15K. To obtain the speedup we compare the total execution time for the 256 tasks on a single thread with the parallel execution time. The theoretical maximum speedup ($16\times$) is depicted by the **UPPER** curve, while the **LOWER** curve shows a lower bound to the speedup (i.e., below this value we have slowdown).

The figure shows that the **LOWER** value is reached for values of GR $\approx$80, while the **UPPER** bound is asymptotically reached for granularities of $\approx$5000. Note that the actual maximum speedup is lower than $16\times$, because one processor acts as a task producer and does not take part to the actual parallel execution. This limits the maximum speedup achieved in slightly more than $15\times$. In this region we do not appreciate significant difference among the architectures. For finer tasks, however, the performance of different architectures differentiate. Figure 8 (b) "zooms in" the finer task region, plotting the relative speedups referred to the **ARCH 3** for a given granularity. The numbers on top report the absolute values for the speedup of **ARCH 3**. A considerable speedup (20 to 30%) is achieved when switching from **ARCH 1** to **ARCH 2**, at any granularity. Switching to **ARCH 3** further improves performance, up to +30% speedup (for GR$\approx$ 10).

In summary, these experiments identify 80 ALU instructions as the minimum task granularity to achieve any speedup in our implementation and 5000 to reach the upper bound. Note that tasks in real applications are likely to have more than 80 instructions, including memory accesses, here not modeled.

We also compared our tasking support with GCC-OpenMP [11] and OMPi [9]. Experiments were performed on a Intel i7 quad-core machine @3.4GHz, featuring HyperThreading technology. Since the target machine only has four cores, for fair comparison we repeat the experiment on an instance of our cluster with the same number of PUs. In addition we also compare HyperThreading (8 threads) with our runtime running on 8 PUs. The results for this experiment are shown in Figure 8(c). Solid lines refer to the experiment with four cores, dashed lines refer to the experiment with 8 cores. The results show that **ARCH 3** asymptotically reaches the maximum speedup ($4\times$) for GR$\approx$5000, outperforming both GOMP and OMPi which reach their peak values for GR$\approx$100000. Similar conclusion applies when 8 threads are considered. Note that, since our synthetic tasks are made of ALU instructions, HyperThreading only achieves $6\times$. Results prove that we achieve peak speedup for tasks $\approx 20\times$ finer-grained than both GOMP and OMPi.

C. Real Benchmarks

In this section we analyze our runtime on two real programs: *Strassen* matrix multiplication and FAST corner detection.

1) Strassen: The Strassen algorithm for matrix multiplication exposes high degrees of parallelism and consequently allows parallelization with tasks of different granularities. The algorithm is naturally structured in three stages: in Stage 1 ten sums are computed. Similarly, Stage 2 consists of seven multiplications, and Stage 3 consists of four sets of sums and subtractions. Each of these operations can be mapped entirely onto a coarse-grained task (our COARSE scheme). In alternative, the loops from which their computation takes place can be parallelized to obtain finer tasks (our FINE scheme).

For the COARSE scheme, Stage 3 was further split in two tasks, separated by a **`taskwait`** construct to enforce a data dependence. Thus, the maximum speedups achievable are equal to the number of tasks in each stage (respectively $10\times$, $7\times$ and $4\times$). For the FINE scheme, each task processes two rows of the sub-matrix in every sum/multiplication. Increasing the size of the matrices affects both the number of tasks (the number of rows increases) and the size of the single task (the number of columns increases). Ideally, this scheme extracts the maximum parallelism, with a theoretical speedup of $16\times$.

We consider square matrices $N \times N$, with $N \in \{8, 16, 32, 64, 128\}$. We do not consider larger matrices for two reasons: i) they would not fit into the TCDM and we do not want delays due to data transfers from/to the main memory to affect our observations; and ii) the considered matrix sizes are, in our opinion, representative of *fine-grained* tasks from the class of applications/systems we are targeting.

Figure 9 shows how the speedups for the two tasking strategies scale in the different architectures, as we increase the size of matrices and for each stage.

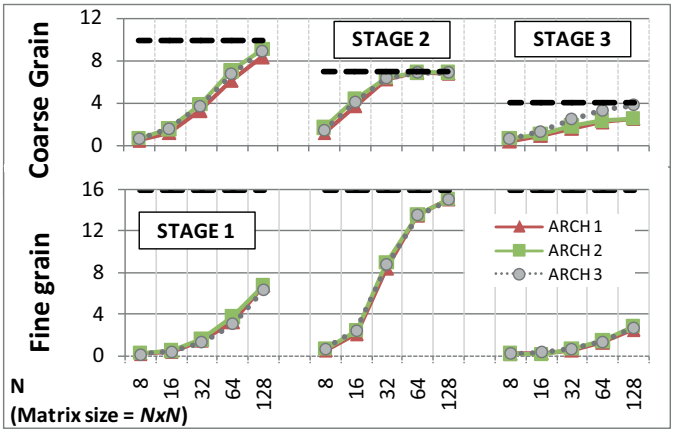

Fig. 9. Speedup of Strassen Algorithm

Ideal speedups are also reported in dotted black lines. Obviously the speedup increases with matrix size, since the overhead of tasking becomes less significant as the the actual workload grows. This is the reason why in Stage 2 (more

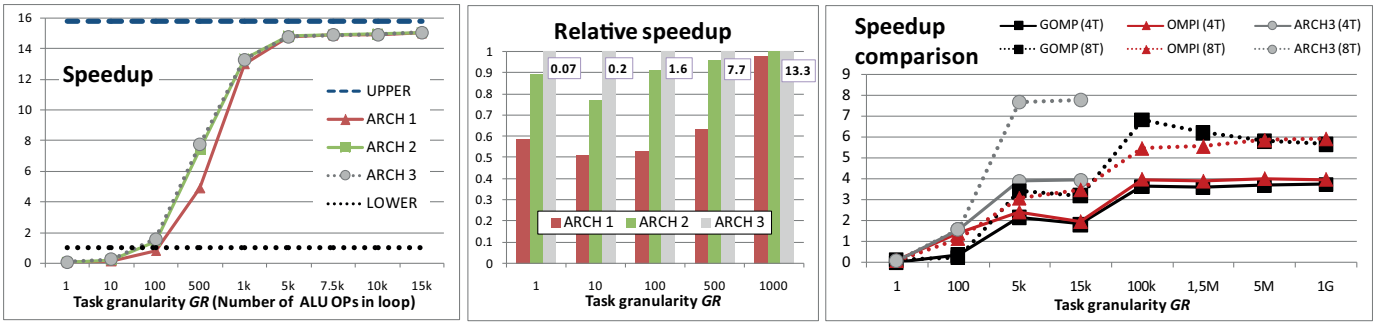

Fig. 8. Parallelization speedup for increasing task granularity (a,b) and against existing OpenMP runtimes (c)

computation) we reach near-ideal speedup for small matrices (32x32 for the COARSE scheme, 64x64 for the FINE scheme). However, the FINE strategy does not always allocate enough work to amortize the overheads, and this is the reason why Stages 1 and 3 do not scale beyond 7× and 4×, respectively.

As shown in the charts, considering different architectures does not significantly affect performance for Stages 1 and 2. This is due to the regular nature of the parallel workload, which does not require synchronization. Stage 3, on the contrary, uses a `taskwait` construct to separate the two sub-stages, thus showing the benefits of **ARCH 3**.

2) FAST: The FAST [16] algorithm compares the intensity value of each point p of the input image with all the sixteen points on the circle of radius 3 and center p. p is classified as a corner if there exists a set of contiguous pixels within the circle that are all brighter (minimum) or darker (maximum) than p (with a tolerance threshold). The number of contiguous pixels and the threshold value are both algorithm parameters; typical values are respectively 9 and 20.

Given an *N*M* input image, the algorithm generates an output vector whose size is *N*M*3*, containing the coordinates of the corner points and a score. The latter is used in a subsequent non-maxima suppression stage, which merges multiple pixels belonging to the same corner. Finally, a keypoint detection pass detects relevant features.

The core kernel performs most of the computation and it exhibit data-parallelism at the pixel level. By estimating the number of instructions in the main loop body we can easily determine a minimum number of iterations to achieve near-ideal speedup by checking the chart in Figure 8. We eventually design our tasks to process an entire image row to achieve this goal. We perform experiments increasing the size $N \times N$ of input images, with $N \in \{64, 128, 256, 512\}$, thus the granularity of tasks doubles with the input size. However, due to the limited size of the TCDM it is not possible to store the whole dataset therein. We thus split the image into *stripes*, and process them one after the other. We adopt a double buffering technique to overlap computation and DMA transfers from the global memory. Table II shows the speedup of the parallelized algorithm compared to the sequential version for different image sizes. A considerable speedup is achieved even for small images (11× for a 64x64 image, with each task only processing 64 pixels) and the speedup reaches 91% of the theoretical 16× for $N \geq 256$.

TABLE II

Input dim	64x64	128x128	256x256	512x512	Ideal
Speedup	11,01	13,54	14,19	14,60	16

VI. Conclusion

Embedded systems are embracing the many-core paradigm. Cluster-based designs are a promising solution to achieve performance and scalability while meeting power budgets. However, powerful programming abstractions are required to use these machines effectively. In particular, modern embedded applications call for paradigms to express dynamic and irregular patterns of parallelism. The OpenMP tasking model provides a convenient conceptual framework to exploit this kind of parallelism. However, the fine-grained nature of parallel tasks in embedded workloads calls for a design of the runtime support which minimizes the cost of its implementation. We identified the key memory bottlenecks implied in the tasking execution model and proposed an implementation aimed at minimizing their impact, considering architectural variants of a generic shared-memory cluster. Experiments performed with both synthetic and real benchmarks demonstrate that our lightweight support enables fine-grained tasking, thus being suitable for the targeted class of embedded systems. From an extensive characterization of the overheads of our tasking support we derived guidelines to quickly size tasks in applications, to achieve near-ideal speedups. In the future we will consider new metrics in the experiments, namely memory utilization and number of banking conflicts.

Acknowledgment

This work was supported by projects FP7 VIRTICAL (288574) and JTI SMECY (ARTEMIS-2009-1-100230), funded by the European Community.

References

[1] Plurality Ltd. The HyperCore Processor Whitepaper. http://www.warthman.com/projects-Plurality_Architecture_Shared-Memory_Synchronizer-Scheduler_Load-Balancing_Task-Oriented-Programming_Parallel-Cores.htm. April 2010.

[2] 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.

[3] NVIDIA. FERMI Series Whitepaper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. 2009.

[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.3.1. http://www.openmp.org/mp-documents/OpenMP3.1.pdf. July 2011.

[8] AMD, Inc. Fusion Series Whitepaper. http://www.amd.com/us/Documents/48423_fusion_whitepaper_WEB.pdf March 2010.

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

[10] 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.

[11] FSF - The GNU Project. GOMP - An OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp/. 17 September 2011.

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

[13] A. Rahimi et al. A fully-synthesizable single-cycle interconnection network for shared-L1 processor clusters. In *DATE 2011*.

[14] D. Bortolotti et al. Exploring Instruction caching strategies for tightly-coupled shared-memory clusters, *SoC*, 2011.

[15] A. Marongiu et al. Fast and Lightweight Support for Nested Parallelism on Cluster-Based Embedded Many-Cores In *DATE 2012*

[16] E. Rosten et al. Faster and better: A machine learning approach to corner detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 32:105–119, 2010.