Tightly-Coupled Hardware Support to Dynamic Parallelism Acceleration in Embedded Shared Memory Clusters

Paolo Burgio*†, Giuseppe Tagliavini*, Francesco Conti*, Andrea Marongiu*‡, Luca Benini*‡
*DEI – Università degli Studi di Bologna – Italy
†Università di Modena – Italy
‡Integrated Systems Laboratory – ETH Zurich – Switzerland
Email: {paolo.burgio, giuseppe.tagliavini, f.conti, a.marongiu, luca.benini}@unibo.it

***Abstract*—Modern designs for embedded systems are increasingly embracing cluster-based architectures, where small sets of cores communicate through tightly-coupled shared memory banks and high-performance interconnections. At the same time, the complexity of modern applications requires new programming abstractions to exploit dynamic and/or irregular parallelism on such platforms. Supporting dynamic parallelism in systems which i) are resource-constrained and ii) run applications with small units of work calls for a runtime environment which has minimal overhead for the scheduling of parallel tasks. In this work, we study the major sources of overhead in the implementation of OpenMP dynamic loops, sections and tasks, and propose a hardware implementation of a generic Scheduling Engine (HWSE) which fits the semantics of the three constructs. The HWSE is designed as a tightly-coupled block to the PEs within a multi-core cluster, communicating through a shared-memory interface. This allows very fast programming and synchronization with the controlling PEs, fundamental to achieving fast dynamic scheduling, and ultimately to enable fine-grained parallelism. We prove the effectiveness of our solutions with real applications and synthetic benchmarks, using a cycle-accurate virtual platform.**

I. Introduction

During the last decade, we witnessed the shift from single to multi- and many-core architectures for embedded systems. One common design paradigm for embedded many-cores consists of multi-*cluster* computation *fabrics*, where each *cluster* contains up to a few tens of simple processing elements (PEs) communicating via low-latency, high-throughput interconnection and shared L1 memory. Examples of cluster-based platforms are nowadays numerous, and include STM STHORM [5], Plurality HAL [19] or even GP-GPUs as NVIDIA Fermi [15] among the others[10] [2].

Cluster-based architectures are capable of tremendous peak Gops/Watt, but efficiently harnessing such a huge computational power is mostly demanded to the software layer. At the same time, embedded applications from the domains targeted by such architectures (e.g., image processing, computer vision etc) are increasing in complexity and often expose high degree of parallelism which is irregular in nature and/or dynamically generated. Consequently, sophisticated programming abstractions and associated tool-chains are necessary for efficiently exploiting cluster-based embedded systems.

Notable examples of computation models where units of work can be dynamically and asynchronously created can be found in the multi-core general-purpose and high-performance computing domains. Cilk Tasks [14], Intel Threading Building Blocks (TBB) [9], or other approaches such as the Apple Grand Central Dispatch [4], Intel Carbon [11] or the current OpenMP specification [1] all provide means to dynamically specify more or less regular types of parallelism in an application. Among these programming models, OpenMP is particularly appealing for the target cluster architecture, for several reasons: i) most programmers are familiar with its intuitive interface based on parallelization directives, to be added to a standard C program; ii) the standard provides several flavours of dynamic parallelism: *dynamic loops*, *sections* and *tasks*; iii) unlike the other mentioned programming models, a number of OpenMP ports for embedded systems exist [7][3][8][13], which provides guidelines and code to get started.

978-3-9815370-2-4/DATE14/ © 2014 EDAA

OpenMP constructs for dynamic parallelism provide a powerful and flexible solution to exploit irregular parallelism in target applications, but their 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. Minimizing runtime overheads is thus a primary challenge to enable fine-grained dynamic parallelism on embedded clusters.

In this work, we study the major sources of overhead in the implementation of OpenMP *dynamic loops*, *sections* and *tasks*, and propose a hardware implementation of a generic Scheduling Engine (HWSE) which fits the semantics of the three constructs. The adaptability of this HW block in the context of different programming models is also discussed. The HWSE is designed as a tightly-coupled block to the PEs within a multi-core cluster, communicating through a shared-memory interface. This allows very fast programming and synchronization with the controlling PEs, fundamental to achieving fast dynamic scheduling, and – ultimately – to enable fine-grained parallelism. We develop RTL models of the cluster and the HWSE, to obtain accurate synthesis results, and SystemC models, which allow us to run complete applications to validate the proposed approach. We compare the results achieved by our HWSE with two freely-available OpenMP runtime implementations, OMPi [3] and GNU LIBGOMP [8].

The rest of the paper is structured as follows. We discuss related works in Section II and the target architectural template in Section III. In Section IV we analyze the OpenMP constructs for dynamic parallelism, and in Section V we describe the HWSE and how it supports them. Finally, we

validate our approach and characterize the performance of our implementation in Section VI, then summarize our main findings and discuss future work in Section VII.

II. Related Work

Several programming models from the general-purpose computing domain support dynamic parallelism. Examples are OpenMP [1], Cilk [14], Intel TBB [9], Apple Grand Central Dispatch [4], among the others. Dynamic parallelism is typically implemented by leveraging some sort of centralized or distributed queue system, where tasks are inserted and extracted dynamically, as the worker threads become available at run-time. In some cases [4] [11] [19], the queue management is supported by a dedicated hardware module. Kuacharoen et al. [16] propose an HW task scheduler whose policy can be dynamically reconfigured at runtime, while Pilkington [18] implement a hardware thread scheduler for multi-processor systems with real-time constraints. These solutions are tailored to specific real-time systems, and do not cope with tightly-coupled integration with processing elements, nor with programming models. Paulin et al. [17] propose a *object broker* (ORB) to dispatch tasks in a in a NoC-based multi-tile systems. The approach considers an execution model which adheres to a synchronous client-server semantic, while OpenMP is more flexible and supports also asynchronous and deferred task creations. In addition, ORB is not tightly-coupled to cores, which implies a high cost (50 call cycles for a complete round trip) [17], while our HWSE can efficiently spawn and retrieve units of work in just few clock cycles. Tendulkar et al. [21] propose a lightweight implementation of OpenMP basic services on the SARC architecture [20], exploiting architecture–specific features (such as hardware counters and low-level communication primitives). Their architecture is significantly different from our tightly-coupled clusters, which ultimately leads to different design requirements and implementation solutions. Kwok et al. [12] propose an energy efficient hardware scheduler for an architecture whose tasks are accelerated on an FPGA, thus targeting an heterogeneous (host+FPGA) architecture with specialized HW accelerators, which is very different from our clusters. The Plurality HAL [19] embeds a task scheduler and synchronization unit called Central Synchronizer Unit (CSU), and provide a proprietary C-like language for task definition. Intel Carbon [11] embeds a complex distributed queue system+task dispatcher in a ring-based architecture, providing ISA extensions to interact with the engine. Clearly these products feature ad-hoc optimized designs for a specific platform and associated programming model. As such, they obviously do not share our goal of providing an implementation of a hardware-scheduler which i) is tightly-coupled to the PEs of a shared-memory, multi-core cluster, thus allowing very fine-grained units of computations; ii) generically captures the semantics of common constructs for supporting dynamic parallelism, which makes it suitable to accelerate several programming models. Section V comments on the applicability of the HWSE to some of these programming models.

III. Architecture

In Figure 1 we show a simplified block diagram of the target cluster. It is inspired by platforms such as STM STHORM [5], Plurality HAL [19] and Kalray MPPA [10]. It is composed of (up to) 16 RISC-32 processors connected through a low-latency, high bandwidth interconnect similar to the ones proposed by Plurality LTD [19]. Processors communicate through a fast multi-banked, multi-ported memory, which is configured as a shared, software-managed scratchpad memory (SPM). The number of ports and banks is a multiple K of the number of processors to increase bandwidth. In case there are no bank conflicts, concurrent accesses by multiple cores to the SPM are served simultaneously by the interconnection. Bank conflicts result in a higher latency, due to contention, which is resolved based on round-robin arbitration. The crossing latency of the interconnection is one clock cycle, and word interleaving minimizes the probability of conflicts due to simultaneous accesses to the same bank from multiple readers/writers. As a consequence, conflict-free SPM 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 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 SPM 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. To allow for performance- and energy-efficient transfers, the cluster is equipped with a DMA engine. We consider a simple DMA design, with one slave port from which processors can directly program transfers through memory mapped registers, plus two master ports to move data in/out of the cluster. This DMA is capable of 16-word burst operations. In this work we consider a two-level memory system, with an off-cluster main memory, and we assume a global address space. Synchronization among the processors is achieved through a segment of the local SPM address space featuring *test-and-set (TAS)* semantics.

IV. Analysis of OpenMP dynamic parallelism cost

In this section we analyze OpenMP constructs for dynamic parallelism, aiming at i) deriving a minimal set of primitives that capture their semantics and ii) characterizing the cost of each primitive, to discover best candidates for HW acceleration. The reference OpenMP implementation considered in this work is the GNU GCC runtime library (`libgomp`). This will be used as a baseline also for our evaluation section.

OpenMP features three different constructs to support dynamic parallelism: *sections*, *dynamic loops* and *tasking*. Figure 2 shows three code snippets providing examples of use for each construct:

1) **Sections** (Figure 2a). Different portions of code are annotated to statically decompose a program into coarse-grained tasks (here, `task_A` and `task_B`) deployed onto parallel threads;
2) **Dynamic loops** (Figure 2b). Tasks are dynamically created out of *chunks* of loop iterations and executed by parallel threads.
3) **Tasking** (Figure 2c). OpenMP *Tasks* have been introduced since specifications version 3.0. Compared to *sections*, OpenMP *tasks* enable more sophisticated forms of dynamic, irregular and asynchronous parallelism.

Figure 3 shows how the code in Figure 2 is transformed by the GCC compiler. The compiler resorts to the runtime system to retrieve the next available chunk of loop iterations for dynamic loops (`GOMP_dynamic_next()`) or the next available section (`GOMP_section_next()`). Both functions implement a simple FIFO queue, to which parallel threads access in a mutually exclusive manner to

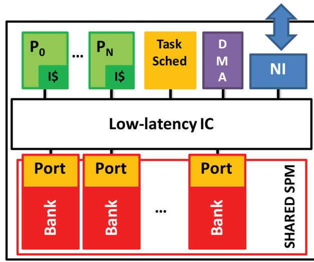

Fig. 1. On-chip shared memory cluster template

```
/* a) */
#pragma omp sections
{
  #pragma omp section
  { task_A (); }
  #pragma omp section
  { task_B (); }

} /* End of workshare:
     (implicit) Synch */
```

```
/* b) */
/* 4 iter => 1 task */
#pragma omp for schedule \
  (dynamic, 4)
for (i=0; i<64; i++)
{
  work_iter (i);
} /* End of workshare:
     (implicit) Synch */
```

```
/* c) */
#pragma omp single nowait
{
  for (i<64)
  { /* Task creation */
    #pragma omp task
    { task_A (i); }
  }

  /* Task-based synch */
  #pragma omp taskwait

  #pragma omp task
  { task_B (i); }

} /* (Implicit) thrd synch:
     execute all tasks */
```

Fig. 2. Different construct for dynamic parallelism: a) *sections*, b) *dynamic loops*, c) *tasks*

```
/* a) */
/* N_SECTIONS: 2 */
GOMP_sections_start (2);

while(ID = GOMP_sections_next()) {
  switch(ID)
  {
    case 1: task_A (); break;
    case 2: task_B (); break;
    case 0: /* END */  break;
  }
}
GOMP_sections_end ();
```

```
/* c) */
if(GOMP_single_start ())
{
  for(i<64) /* Pass FN, DATA */
    GOMP_task (&task_A, { &i });

  GOMP_taskwait ();
  GOMP_task (&task_B, NULL);
}
GOMP_single_end ();
/* (Implicit) thread synch:
    execute all tasks */
```

```
/* b) */
/* START: 0, END: 64, INCR: +1, CHUNK: 4 */
GOMP_dynamic_loop_start (0, 64, +1, 4);
while (GOMP_dynamic_loop_next (&ISTART, &IEND))
{
  for (i=ISTART; i<IEND; i++)
    work_iter (i);
}
GOMP_loop_end ();
```

Fig. 3. GCC-transformed dynamic parallelism constructs: a) *sections*, b) *dynamic loops*, c) *tasks*

```
/* INIT */
int GOMP_loop_dynamic_start(int start, int end,
                            int incr, int chunk) {
  gomp_work_share_t ws = /* Create WS */;
  /* Init WS fields */
  ws.chunk = chunk;
  ws.end = ((stride > 0 && start > end)
    || (stride < 0 && start < end)) ? start : end;
  ws.stride = stride;
  ws.next = start;
  return INIT_OK;
}

/* END */
void GOMP_loop_end() {
  /* Let thread move to the next WS */
  current_WS[thread_ID]++;
}
```

Fig. 4. GOMP code snippet for loop **INIT** and **END**

```
/* FETCH */
int GOMP_loop_dynamic_next (gomp_work_share_t *ws,
                            int * pstart, int * pend) {
  /* 'ws' holds the status on thread's current WorkShare */
  int start, end, chunk, left;

  LOCK(); /* Atomically access to the WS */
  start = ws->next;
  if (start == ws->end)
    return WS_ENDED; /* No more work in the WS! */

  /* Extract work (a chunk of iterations) from the WS */
  chunk = ws->chunk_size * ws->stride;
  left = ws->end - start;
  /* Adjust boundaries if we exceed # loop iterations*/
  if (ws->stride < 0) {
    if (chunk < left) chunk = left;
  } else {
    if (chunk > left) chunk = left;
  }
  end = start + chunk;
  ws->next = end; /* *pstart/end are passed to application */
  *pstart = start; *pend = end;
  UNLOCK(); /* Release the WS */

  return WS_HAVE_WORK;
}
```

Fig. 5. GOMP code snippet for loop **FETCH**

update a shared counter. *Sections* and *dynamic loops* rely on a *work-share* data structure, which describes the parallel work to be done (e.g., number of iterations, chunk size, global lower and upper bounds of a loop, etc.). The code snippet in Figure 4 shows how the *work-share* data structure is initialized in the `GOMP_loop_dynamic_start()` function, and how the current thread is pointed to the next *work-share* when the loop (or section) is over in the `GOMP_loop_end()` function. These operations can be captured by two generic **INIT** and **END** primitives. Figure 5 shows how the `GOMP_loop_dynamic_next()` function updates the *work-share* during *loop* (or *sections*) execution. OpenMP *sections* can be seen as a specialized case of *loops* where *chunk* = *stride* = 1. A generic **FETCH** primitive can be used to generalize the work-share update operation. A more in-depth analysis is required for OpenMP tasks, as follows. Figure 6 shows execution time breakdown for the Task **INIT** primitive. The major contributors are the critical region to update the FIFO queue, and the memory allocation for the OpenMP Task descriptor. FIFO semantics can easily be supported in HW, and since management of pre-allocated memory bins is also a very generic operation, used in every runtime system, we select also this functionality for HW acceleration. The **FETCH** primitive for *tasks* can thus be enriched with this functionalities. A *task* in the work queue can be *executing*, *unexecuted* or *ended*, thus a mechanism for tracing its status must be put in place. To this aim we enrich the semantics of the **INIT** and **END** primitives for *tasks*.

Table I summarizes the functionality of the selected primitives for HW acceleration. Their implementation is discussed in Section V.

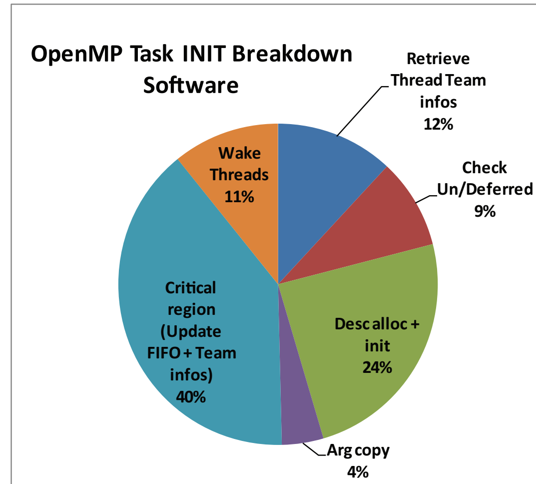

Fig. 6. Timing overhead of task **INIT** performed in software, breakdown of the different components

V. The Hardware Scheduling Engine

In this section we describe the *Hardware Scheduling Engine (HWSE)*, a module to accelerate in HW the primitives introduced in Section IV.

Type	INIT	FETCH	END	SYNC
Sections	**W** task_descr for each section update thread status	**R** task_descr (section_descr)	update thread status	-
Dynamic loops	**W** loop infos: start, end, .. update thread status	**R** chunk_istart, chunk_iend	update thread status	-
Tasking	**W** task_descr	**R** task_desc update task status	update task status	explicit

TABLE I. DESCRIPTION OF THE DIFFERENT PRIMITIVES FOR EACH OF THE THREE DYNAMIC PARALLELISM CONSTRUCTS

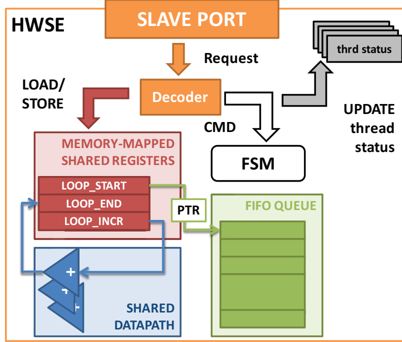

Fig. 7. Scheme of the HWSE

#tasks	**Area** (kgates) *32*	*64*	*128*	**Power** (mW) *32*	*64*	*128*
Decoder FSM	5.42	5.42	5.38	2.42	4.80	9.57
Datapath	3.00	2.81	2.78	1.35	1.27	1.27
Task queue	4.70	9.39	18.69	2.47	2.47	2.46
Total	13.12	17.62	26.85	6.24	8.54	13.30
% of cluster	1.49	1.99	3.01	1.23	1.67	2.59

TABLE II. HWSE MODULE

A. HW Module Implementation and integration in the cluster

The internal core structure of the HWSE (shown in Figure 7) consists of a control finite-state machine that receives the various **INIT**, **FETCH** and **END** primitives and responds accordingly. We design a central *core* datapath implementing these primitives, plus additional logic to specialize their behavior for the construct at hand (*loops*, *sections*, *tasks*, memory allocation). Before using it, the HWSE must be configured to enable the desired construct. This can be done via memory-mapped configuration registers, which are appropriately set within SW routines that we provide (`hwse_init_*`, see Section IV). The **INIT** primitive for *dynamic loops* simply consists of writing lower bound, upper bound and stride into the LOOP_START, LOOP_END, LOOP_INCR registers. The same happens for *sections* (a special case of loops with *chunk* = 1). Loop boundaries (or the next available section) are computed by a submodule implementing the **FETCH** primitive. A simple circular buffer of 32, 64 or 128 elements implements the FIFO queue; the control FSM is responsible for storing and extracting elements from the queue. Invoking the **END** primitive results in updating a thread-specific register which stores its current *work-share*.

To implement the memory allocator functionality we reuse entirely the logic for loop scheduling. In the **INIT** primitive the base address for the memory heap, its global size and the size of a memory bin (containing the specific work/task descriptor) are stored respectively in the LOOP_ START, LOOP_END and LOOP INCR registers. Requests for a new memory bin are serviced through the loop iterations scheduler, until there are available bins. Then, memory bins are extracted from the FIFO queue (`alloc`) in the **FETCH** primitive, and inserted back therein (`free`) in the **END** primitive.

Task support deserves further discussion. The **INIT** primitive supports the creation of a task (function `GOMP_task()` in Figure 3c)) by inserting the address of a newly created task descriptor in the FIFO queue. Similarly, the **FETCH** primitive dequeues a task descriptor address from the queue. The **END** primitive for *tasks* was not accelerated in hardware, and the reason will be explained in next section.

We integrated the HWSE in the target cluster, tightly coupling it to cores through the high-speed interconnection. The FSM can be controlled by the cluster by means of a memory-mapped interface; registers are memory-mapped, and special addresses trigger the different primitives. We implemented a RTL (SystemVerilog) model of the HWSE, and synthesized it using the STMicroelectronics 28 nm bulk low-threshold libraries as a target, with a clock frequency of 400 MHz. Table II summarizes the results regarding area (in gates) and power (in mW), and the impact on the cluster area and power (in %), which we gathered similarly. Depending on the queue size, the HWSE adds $\approx$ 1%-3% to the area and power of the original cluster design. Much of the area occupation and power consumption of the HWSE is in the FIFO task queue and thus depends of its depth (the maximum number of tasks supported). In absence of contention only 2 clock cycles (for crossing the IC) + 1 (the delay added by the module) are necessary to execute any primitive.

B. Programming Interface and integration in the libgomp

To conveniently program the HWSE we have developed a SW API, which abstracts the low-level process of register configuration. Table III summarizes the functions provided by this API and their description. As already discussed previously, our HWSE implementation is based on the analysis and optimization of the GCC OpenMP runtime library: `libgomp`. Table III also lists the corresponding functions in the `libgomp` library for supporting *dynamic loops*, *sections* and *tasks*. Starting from this implementation, we replaced the schedulers for *sections*, *dynamic loops* and *tasks* with calls to our HWSE API. For the former two constructs the operation was straightforward, due to the one to one correspondence between the HW and SW primitives. Tasks, on the contrary, have much more sophisticated semantics than a simple FIFO queue, which required more work for the integration. Task-level synchronization implies that any thread encountering a `taskwait` construct must wait on the completion of child tasks of the current task (see specifications [1]). This implies that parent-children information among tasks must also be stored, other than a FIFO representation. `libgomp` does so by using a tree data structure. We opt for a more lightweight implementation based on atomic counters [3][9], handled in

software rather than in the HWSE, to maintain the generality of our primitives implementation.

Task descriptors contain information on shared data, thus can become very large. Thus we do not store the descriptors themselves in the HWSE FIFO, but only their address. Descriptors are stored in the shared L1 SPM, so once their address is extracted the SW can quickly access the information therein. Since the atomic counter to support *taskwait* is part of the *task* descriptor, the **END** primitive for tasking (which simply updates it) was not implemented in hardware.

C. Applicability of the HWSE to different programming models

Table IV gives an overview of the most relevant task-based programming models. As shown most of them are implicitly asynchronous, and in some cases also assume a fork-join execution model, as OpenMP. The semantics of **INIT/FETCH** primitives perfectly matches the behavior of a work queue supporting asynchronous execution. Synchronicity (and synchronization) can be implemented with the support of the **FETCH** primitive, e.g., wrapping it in a software loop until there are tasks to execute. Complex dependencies between tasks (e.g., parent-children relationship) can be expressed enriching the descriptors of the work to execute. Indeed, our primitives (and the corresponding HWSE implementation) agnostically handle the memory address of a language-specific data structure describing a single task, that therefore can include information – such as references to other task structures – to be managed by a higher software level. The work descriptor can be enriched also to specify a set of tasks, e.g., to support data parallelism similarly to what happens in Intel TBB [9], where high-level data parallel constructs are built on top of a task scheduling library. Finally, all the programming models shown in Table IV abstract memory allocation to software. The HWSE proposed in this work can be configured as a pre-allocated memory manager to support and accelerate memory allocation and free primitives.

VI. Experiments

We prototyped the proposed cluster using a SystemC Virtual Platform [6] which models the HWSE integrated in the cluster platform described in Section III, with main architectural parameters as summarizes in Table VI. With this setup, we validate our approach both with synthetic benchmarks, and applications from image processing domains.

ARM v6 cores	16	# L1 SPM banks	32 (K=2)
L1 SPM size	256 KB	# L1 SPM latency	$\geq$ 2 cycles
L3 size	256 MB	L3 latency	$\geq$ 59 cycles
$I\$_i$ size	1 KB	$I\$_i$ line	4 words
t_{hit}	= 1 cycle	t_{miss}	$\geq$ 59 cycles

TABLE VI. Architectural parameters

A. Synthetic benchmarks

The first experiment to measure the performance improvement brought by our HWSE compared to the software schedulers consists of three synthetic workloads. To test accelerated *sections* we spawn 24 sections each consisting of 100 NOPs (to prevent side effects due to memory contention). For *dynamic loops* we create a loop of 64 iterations each containing 100 NOPs, while for the *tasking* we spawn 18 tasks each containing a loop of 5000 iterations of 100 NOPs. All the processors are involved in the computation. Table VII summarizes the speedup brought by the HWSE for each of the three primitives, over the pure SW version. Accelerated *sections* provide the best speedups, significantly higher than *dynamic loops* even if the two constructs share almost identical semantics. The reason for this difference is that each **INIT** and **FETCH** event for the *sections* implies a single write (read) in the HWSE, while *loops* require multiple consecutive writes (reads). Consequently the HWSE must be locked to prevent non-mutually exclusive updates from distinct threads. This operation is done in software, and implies the difference in performance that we observe. Similarly, the HWSE can only accelerate a portion of the sophisticated *task* scheduler, leaving a relevant portion of the code to be executed in software. For this reason we observe a more modest 41% speedup for the **INIT** and 21% for the **FETCH**. The **END** primitive, as already explained, was not accelerated.

Type	INIT	FETCH	END
Sections	16×	78×	181×
Dynamic loops	1.07×	6×	14×
Tasks	1.41×	1.21×	-

TABLE VII. Sections and loop speedup compared to the pure SW version

B. Comparison with software schedulers

We compared our HWSE to two freely available OpenMP runtimes, namely `libgomp` [8] and OMPi [3]. Both runtime systems have been ported on the target cluster platform. For this comparison we consider 5 image processing applications: JPEG decoding, Color Tracking, Strassen matrix multiplication, FAST corner detection, Viola-Jones face detection. For each of them we propose, where possible, two alternative implementations: one which uses *tasks* and one which uses *dynamic loops* or *sections*. In both cases we generate work units as fine-grained as possible [7] [11] [21], to verify the effectiveness of our HWSE.

Table V shows the performance improvement for each application, when the HWSE is compared to the software schedulers in `libgomp` and OMPi. For the Strassen matrix multiplication we provide the speedup for each of the three main phases of the algorithm. We see almost no performance gain for stage 2, because the work units are very coarse grained, which tends to minimize the impact of the software runtime overheads A similar situation takes place for the task-based version of the face detection. Besides these two cases, on average the HWSE achieves $\approx 32\%$ speedup versus `libgomp`, and $\approx 76\%$ speedup versus OMPi.

VII. Conclusions

Modern embedded systems are embracing many-core cluster-based designs. To efficiently exploit the potential of such machines, new powerful abstractions are necessary, that support irregular and dynamic parallelism. These abstractions require a runtime support whose overhead can be significant, hindering performance and restricting the tasks that can be efficiently spawned to the ones at coarse granularities. In this work we analyze the majour sources of overhead incurring when supporting one of the most expressive programming models for dynamic parallelism – OpenMP – on shared-memory many-cores clusters. We formalize a set of primitives for genericallyupporting all of them, and identify key performance bottlenecks. We implemented these primitives in a Hardware Scheduling Engine HWSE and compare against

Type		HWSE APIs	libgomp API
Sections	INIT	`hwse_sections_init_count(n_sections)`	`GOMP_sections_start(n_sections)`
	FETCH	`hwse_sections_fetch_ID()`	`GOMP_sections_next()`
Dynamic loops	INIT	`hwse_loops_init_loop(start, end, incr * chunk)`	`GOMP_dynamic_loop_start(start, end, incr, chunk)`
	FETCH	`hwse_loops_fetch_iters(&istart, &iend)`	`GOMP_dynamic_loop_next(&istart, &iend)`
Tasking	INIT	`hwse_task_init_desc_addr(desc_addr)`	`GOMP_task(FN_PTR, DATA_PTR, ...)`
	FETCH	`hwse_task_fetch_desc_addr()`	Task Scheduling Point. See OpenMP tasking specs.[1] for details.

TABLE III. HWSE APIs. SECTION IV DESCRIBES THEIR PARAMETERS IN DETAILS.

Name	Explicit a/synch exec.	Task synch.	Task Parall.	Data Parall.	Inter-task dep.	Mem. alloc
Intel TBB [9]	Fork/Join	Explicit Join	Tasks	Lib built on top of Task scheduler	Group `spawn_and_wait`	Implicit in Class Inherit.
OpenMP [1]	Implicitly asynch Explicit synch	Implicit at TSP Explicit (`taskwait`)	*Tasks* *Sections*	*Dynamic* *Loops*	Parent-child `taskwait`	Transparent
Cilk [14]	Fork/Join	Explicit Join	*Co-operative* Tasks	`cilk_for`	-	Transparent
Apple GCD [4]	Synch/Asynch (queue-based)	Explicit (Q WAIT)	Tasks	-	-	Implicit in Q ALLOC
Plurality CSU [19]	Asynch	Token-based	*Regular* Tasks	*Duplicable* Tasks	Tokens	Transparent

TABLE IV. MOST RELEVANT PROGRAMMING MODELS SUPPORTING DYNAMIC PARALLELISM

Type	JPEG	Color tracking	Strassen	FAST	Face detection	Average
	GCC-OpenMP [8]					
Loops	**27%**	**6%**	*S1*: **80%** *S2*: **28%** *S3*: **80%**	**53%**	**20%**	**42%**
Tasks	**26%**	**27%**	*S1*: **26%** *S2*: **0.5%** *S3*: **22%**	**48%**	**3%**	**21.8%**
	OMPI [3]					
Loops	**48%**	**27%**	*S1*: **83%** *S2*: **83%** *S3*: **82%**	**83%**	**85%**	**70.1%**
Tasks	**80%**	**97%**	*S1*: **96%** *S2*: **44%** *S3*: **97%**	**95%**	**90%**	**85.6%**

TABLE V. HWSE PERFORMANCE IMPROVEMENT AGAINST LIBGOMP AND OMPI SW SCHEDULERS.

existing runtimes ported on the target cluster, resulting in up to 97% performance improvement. We also synthesized the HWSE RTL model to gather precise area and power consumption. The HWSE adds approximately 3% to the area and power of the original cluster design.

ACKNOWLEDGEMENTS

This work was supported by projects FP7 VIRTICAL (288574), P–SOCRATES (611016) and ERC-AdG MultiTherman (291125), funded by the European Community.

REFERENCES

[1] OpenMP Application Program Interface v3.1, 2011.

[2] Adapteva, Inc. Epiphany-IV 64-core 28nm Microprocessor. [Online] http://www.adapteva.com/products/silicon-devices/e64g401/, 2013.

[3] S. Agathos, P. Hadjidoukas, and V. Dimakopoulos. Design and Implementation of OpenMP Tasks in the OMPi Compiler. In *Informatics (PCI), 2011 15th Panhellenic Conference on*, pages 265–269, 2011.

[4] Apple, Inc. Grand Central Dispatch. http://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html.

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

[6] D. Bortolotti, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini. VirtualSoC: a Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip. In *013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*, pages 2182–2187. IEEE, May 2013.

[7] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini. Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1504–1509, 2013.

[8] FSF - The GNU Project. GOMP - An OpenMP implementation for GCC.

[9] Intel Corporation. Threading Building Blocks. [Online] http://threadingbuildingblocks.org/, 2006.

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

[11] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News*, 35:162–173, June 2007.

[12] T.-O. Kwok and Y.-K. Kwok. Practical design of a computation and energy efficient hardware task scheduler in embedded reconfigurable computing systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 4 pp.–, 2006.

[13] A. Marongiu, P. Burgio, and L. Benini. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In *DATE*, pages 105–110, 2012.

[14] Massachusets Institute of Technology. The Cilk Project. [Online] http://supertech.csail.mit.edu/cilk/, 1998.

[15] NVIDIA. Next Generation CUDA Compute Architecture: Fermi - WhitePaper. [Online] http://www.nvidia.fr/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010.

[16] P. Kuacharoen and M. A. Shalan and V. J. Mooney III. A Configurable Hardware Scheduler for Real-Time Systems. In *in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101. CSREA Press, 2003.

[17] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, and G. Nicolescu. Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(7):667–680, 2006.

[18] C. Pilkington. *Thread execution scheduler for multi-processing system and method (US Patent 7802255)*, 2010.

[19] Plurality Ltd. The HyperCore Processor.

[20] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The SARC Architecture. *Micro, IEEE*, 30(5):16–29, 2010.

[21] P. Tendulkar, V. Papaefstathiou, G. Nikiforos, S. Kavadias, D. Nikolopoulos, and M. Katevenis. Fine-grain OpenMP runtime support with explicit communication hardware primitives. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–4, 2011.