TaskGenX: A Task-based Runtime System for Fast Task Generation

***Abstract*—Kallia: TODO:**

- **Add justification for 16x faster task creation**
- **Explain dedup performance**
- **Write intro**
- **Write conclusions**

Proposed names instead of RAM (runtime activity manager):

- **RTGen: Rapid Task Generation**
- **TaskGenX: Task Generation eXpress**
- **GenX or xGen: Express (task) generation**
- **QTP: Quick Task Production**

I think I prefer the RTGen but maybe the TaskGenX sounds better and people will remember it...

Task creation is the main bottleneck on parallel applications. In this paper we present how a specialized HW can overcome this issue by executing explicitly task creation. This design gives us high performance improvements for systems of up to 512 cores.

I. Introduction

Kallia: write Intro The future of parallel computing is highly restricted by energy efficiency [20]. Energy efficiency has become the main challenge for future processor designs, motivating prolific research to face the *power wall*.

An interesting approach towards energy efficiency is the use of asymmetric multi-core (AMC) systems. These systems maintain different types of cores that support the same instruction-set architecture. The different core types are designed to target different (performance or power) optimization points [4], [21].

AMCs have been mainly deployed for the mobile market. These mobile SoCs combine low-power simple cores (*little*) with fast out-of-order cores (*big*) to achieve high performance while keeping power dissipation low. Another area where AMCs have been successful is the supercomputing market. The Sunway TaihuLight supercomputer topped the Top500 list in 2016 using AMCs. In this setup, big cores, that offer support for speculation to exploit Instruction-Level Parallelism (ILP), run the master tasks such as the OS and runtime system. Little cores are equipped with wide Single Instruction Multiple Data (SIMD) units and lean pipeline structures for energy efficient execution of compute-intensive code.

Previous experiences have shown that load balancing and scheduling are fundamental challenges that must be addressed to effectively exploit all the resources in these platforms [14], [15], [18], [19], [24], [27]. Mobile applications rely on multi-programmed workloads to balance the load in the system, while supercomputer applications rely on hand-tuned code to extract maximum performance. However, these approaches are not always suitable for general-purpose parallel applications.

In this paper, we evaluate several execution models on an AMC using the PARSEC benchmark suite. This suite includes parallel applications from multiple domains such as finance, computer vision, physics, image processing and video encoding. We quantify the performance loss of executing the applications *as-is* on all cores in the system. These applications were developed on homogeneous platforms and are bound to suffer from load imbalance on parallel regions that statically distribute the work evenly across cores without considering their performance disparity.

To overcome this matter, we consider two possible solutions at the OS and runtime levels to exploit AMCs effectively. The first solution delegates scheduling to the OS. We evaluate the heterogeneity-aware OS scheduler used in existing mobile platforms that assigns threads to different core types based on CPU utilization. This approach does not require modifying the application, but is limited for high-utilization multithreaded applications.

The second solution is to transfer the responsibility to the runtime system so it dynamically schedules work to different core types based on work progress and core availability. We evaluate the impact of using an inherently load-balanced execution model such that of task-based programming models. Recent examples [3], [6], [12], [23], [29]–[31] include clauses to specify inter-task dependencies and remove most barriers which are the major source of load imbalance on AMCs. Another approach of scheduling in the runtime system is to change the existing statically-scheduled work-sharing constructs for the applications implemented in OpenMP to use dynamic scheduling.

This paper quantifies the effectiveness of these solutions at different levels of the software stack with a comprehensive evaluation of representative parallel applications on a real AMC platform: the Odroid-XU3 development board. This platform features an eight-core Samsung Exynos 5422 chip with ARM big.LITTLE architecture with four out-of-order Cortex-A15 and four in-order Cortex-A7 cores.

The rest of this document is organized as follows: Section II describes the evaluated asymmetric multi-core processor, while Section **??** offers information on dynamic schedulers at the OS and runtime system levels. Section IV describes the experimental framework. Section V shows the performance and energy results and associated insights of our experiments. Finally, Section VI discusses related work and Section VII concludes this work.

```
    ...
//task_clause
memalloc(&task, args, size);
createTask(deps, task, parent, taskData);
    ...
```

Listing 1: Compiler generated pseudo-code equivalence for task annotation.

II. Background and Motivation

A. Task-based Programming Models

Task-based parallel programming models [23], [12], [2], are widely used to facilitate the programming of parallel codes for multi-core systems. These programming models offer annotations that the programmer can add to the application's sequential code. By adding these annotations, the programmer decomposes the application into *tasks* and specifies the input and output data dependencies between them. A compiler is responsible to translate the annotations into code by adding calls to the programming model's runtime system. The runtime system consists of software threads and is responsible for the efficient execution of the tasks with respect to the data dependencies as well as the availability of resources.

When the compiler encounters one task annotation in the code, it transforms it to the pseudo-code shown in Listing 1. `Memalloc` is performing the memory allocation for the task and its arguments. Next is a runtime call, which is the createTask, responsible for the linking of the task with the runtime system. At this point a task is considered *created* and below are the three possible states of a task inside the runtime system:

- *Created:* A task is initialized with the appropriate data and function pointers and it is inserted in the Task Dependency Graph (TDG). The insertion of a task in the TDG implies that the data dependencies of the tasks have been identified and the appropriate data structures have been created and properly initialized.
- *Ready:* When all the data dependencies of a created task have been satisfied, the task is ready and it is inserted in the *ready queue* where it waits for execution.
- *Finished:* When a task has finished execution and has not been deleted yet.

The runtime system creates and manages the software threads for the execution of the tasks. Typically one software thread is being bound to each core. One of the threads is the *master thread*, and the rest are the *worker threads*. The master thread starts executing the code of Listing 1 sequentially. The allocation of the task takes place first. What follows is the task creation, that includes the analysis of the dependencies of the created task and the connection to the rest of the existing dependencies. Then, if there are no task dependencies, which means that the task is ready, the task is also inserted in the ready queue and waits for execution.

Listing 2 shows the pseudo-code for the task creation step within the runtime. The `createTask` function is first

```
1 void createTask(Deplist dList, Task t,
2                 Task parent, Data args) {
3   initAndSetupTask(task1, parent, args);
4   insertToTDG(dList, task1);
5 }
6 Dependency depMap[];
7 void insertToTDG(DepList dList, Task t) {
8 if( dList is empty ) {
9   readyQ->push(t);
10    return;
11 }
12 Dependency entry;
13 for( d in dList ) {
14   entry = depMap[d.address()];
15   if(entry==NULL) depMap.add(entry, t);
16   if(d.accessType() == "write")
17     entry.addLastWriter(t);
18   if(d.accessType() == "read") {
19     entry.addReader(t);
20     entry.lastWriter()->addSuccessor(t);
21   }
22 }
23}
```

Listing 2: Pseudo-code for task creation.

initializing the task by copying the corresponding data to the allocated memory as well as connecting the task to its parent task (`initAndSetupTask`). After this step, the task is ready to be inserted in the TDG; this is done by the `insertToTDG` function. This function takes as arguments a list with all the memory addresses that are to be written or read by the task (`dList`), and the task itself. If for a task the `dList` is empty (line 8), this means that there are no memory addresses that need to be tracked during the execution; thus, the task is marked as *ready* by pushing it to the *ready queue* (line 9). Each entry of `dList` contains the actual memory address as well as the access type (read, write or read-write). The runtime keeps a unified dependency tracking structure, the `depMap`(line 6) where it stores all the tracked memory addresses together with their writer and reader tasks. For each item in the `dList` the runtime checks if there is an existing representation inside the `depMap` (line 14). If the memory address of an entry of the `dList` is not represented in the `depMap`, it is being added as shown in line 15. If the address of a `dList` item exists in the `depMap`, this means that a prior task has already referred to this memory location, exhibiting a data dependency. According to the access type of `d`, the readers and the writers of the specific address are updated in the `depMap` (lines 16-21).

To reduce the lookup into the `depMap` calls, every time a memory address is modified, the tasks keep track of their *successors* as well as the number of *predecessors*. The *successors* of a task are all the tasks that their input depends on the output of the current task. The *predecessors* of a task are the tasks whose output is used as input for the current task. When

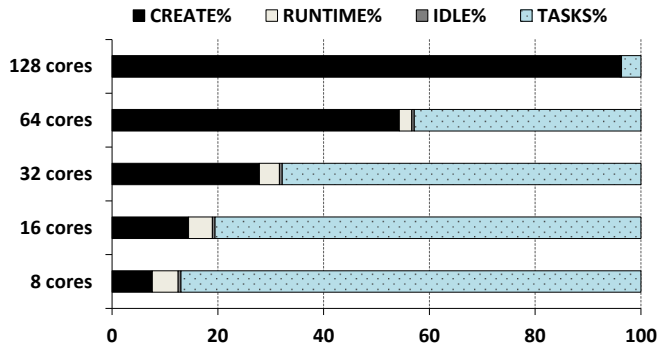


Fig. 1: Master thread activity as we increase the number of cores.

```
1  void task_finish(Task *t) {
2    depMap.removeReaderWriter(t);
4    if(t->successors.empty()) delete t;
5    else {
6      for( succ in t->successors ) {
7        succ.decreasePredecessors();
8        if(succ.numPredecessors == 0)
9          readyQ->push(succ);
10     }
11   }
```

Listing 3: Pseudo-code for task_finish runtime activity.

a `read` access is identified, the task that is being created is added to the list of successors of the last writer task, as shown on line 20 of Listing 2.

As tasks are executed, the dependencies between them and their successors are satisfied. So the successor tasks that are waiting for input, eventually become *ready* and are inserted to the ready queue. When a task becomes *finished*, the runtime has to perform some actions in order to prepare the successor tasks for execution. These actions are described in Listing 3. The runtime first updates the `depMap` to remove the possible references of the task as reader or writer (line 2). Then, if the task does not have any successors, it can safely be deleted (line 3). If the task has successors, the runtime traverses the successor list and for each successor task it decreases its predecessor counter (lines 5-6). If for a successor task its predecessor counter reaches zero, then this task becomes *ready* and it is inserted in the *ready queue* (lines 7-8).

To summarize, the runtime activity mainly takes place at the task state changes. One state change corresponds to the task creation, so a task from being just allocated it becomes created, and the second change occurs when a task from being ready becomes finished. In these two runtime phases the runtime system interferes and introduces runtime overheads. For the rest of the paper we will refer to the first part as the task creation overheads (CREATE) and for the second part is the dependencies overheads which includes the task dependencies manipulation as well as the scheduling that occurs which is the insertion of a task in the ready queue) in case a task becomes ready (RUNTIME).

B. Motivation

Figure 1 shows the runtime activity of the master thread during the execution of the Cholesky benchmark on 8, 16, 32, 64 and 128 cores. The execution time represented here is the wall clock time during the parallel region of the execution. Each one of the series represents a different runtime overhead from the ones described above. The percentage of time spent on task creation is increasing as we increase the number of cores. This is because the creation overhead is static so the more we reduce the application's execution time by adding resources the more important this step becomes in terms of execution time. On the other hand, the task execution time percentage is decreased as we increase the number of cores because the computational activity is being shared among more resources. The RUNTIME decreases as we increase the number of cores because this activity is also shared among the resources. This is because this part of the runtime takes place once the tasks finish execution and new tasks are being scheduled. So the more the resources, the less the runtime activity per thread, therefore less activity for the master thread. Our motivation for this work is the bottleneck introduced by task creation as shown in Figure 1. Our runtime proposal decouples this piece of the runtime and accelerates it on a specialized hardware resulting in higher performance. Kallia: Elaborate a bit??

III. Runtime Activity Manager

In this paper we propose a semi-centralized runtime system that dynamically separates the most computationally intensive parts of the runtime system and accelerates them on specialized hardware. To develop the Runtime Activity Manager (RAM) we use the OmpSs programming model [12], [2]. The base of our implementation is the Nanos runtime system that is the runtime system of the OmpSs programming model.

Nanos is a distributed runtime system that uses dynamic scheduling. As most task-based programming models, Nanos consists of the master and the worker threads. The master thread is launching the parallel region and creates the tasks that have been defined by the programmer. The scheduler of Nanos consists of a *ready queue* (TASKQ) that is shared for reading and writing among threads and is used to keep the tasks that are ready for execution. All threads have access to the TASKQ and once they become available they try to pop a task from the TASKQ. When a thread finishes a task, it performs all the essential steps described in Section II-A to keep the data dependency structures consistent. Moreover, it pushes the tasks that become ready in the TASKQ.

A. Implementation

RAM assumes the existence of a specialized hardware that accelerates the task creation step. RAM relieves the master and worker threads from this intensive runtime activity by offloading it on the special purpose hardware. Our runtime,

```
1 void SRTloop() {
2  while( true ) {
3    while(RRQ is not empty) {
4      executeRequest( RRQ.pop() );
5  }
6  if( runtime.SRTstop() ) break;
7 return;
8}
```

Listing 4: Pseudo-code for the SRT loop.

apart from the master and the worker threads, introduces the Special Runtime Thread (SRT). When the runtime system starts, it creates the SRT and binds it to the task creation accelerator, keeping its thread id in order to manage the usage of it. During runtime, the master and worker threads look for ready tasks in the task ready queue and execute them along with the runtime. SRT, instead of querying the ready queue for tasks, it looks for runtime activity requests in the runtime ready queue (RRQ) and if there are requests, it executes them.

Figure 2 shows the communication infrastructure between threads within RAM. Our system maintains two queues; the ready task queue (`TASKQ`) and the runtime requests queue (`RRQ`). The TASKQ is used to keep the tasks that are ready for execution. The RRQ is used to keep the pending runtime activity requests. The master and the worker threads can push and pop tasks to and from the TASKQ and they can also add runtime activity to the RRQ. The special runtime thread (SRT) pops runtime requests from the RRQ and executes them on the accelerator.

When the master thread encounters a task clause in the application's code, after allocating the memory needed, it calls the `createTask` as shown in Listing 2 and described in Section II-A. RAM decouples the execution of `createTask` from the master thread. To do so, RAM implements a wrapper function that is invoked instead of `createTask`. In this function, the runtime system checks if the SRT is enabled; if not then the default behaviour takes place, that is, to perform the creation of the task. If the SRT is enabled, a *CREATE* request is generated and inserted in the RRQ. The *CREATE* runtime request includes the appropriate info to execute the code described in Listing 2. That is, the dependence analysis data, the address of the allocated task, its parent and its arguments.

While the master and worker threads are executing tasks, the SRT is looking for *CREATE* requests in the RRQ to execute. Listing 4 shows the code that the SRT is executing until the end of the parallel execution. The special runtime thread continuously checks whether there are requests in the RRQ (line 3). As long as there is a pending creation request, the SRT executes the task submission and inserts the task in the TDG with a call to the `executeRequest` (line 4). When the parallel region is over the runtime system informs that the SRT has to stop execution. This is when the infinite loop of the SRT exits and the execution finishes (line 6).

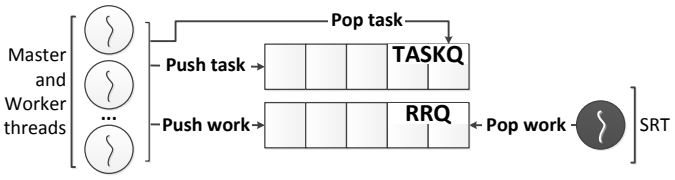


Fig. 2: Communication mechanism between master/workers and SRT threads.

IV. Experimental Methodology

A. Applications

Table I shows the evaluated applications, the input sizes used, and their characteristics. All applications are implemented using the OmpSs programming model [12]. We obtain Cholesky and QR from the BAR repository [5] and we use the implementations of the rest of the benchmarks from the PARSECSs suite [8]. More information about these applications can be found in [8] and [9]. We choose the input sizes of the applications so that they create enough tasks to feed up to 512 cores. The number of tasks for each application and input as well as the average per-task CPU cycles can be found on Table I.

B. Hardware requirements

Our work makes the assumption of a specialized hardware. To evaluate our approach we study the requirements of this accelerator in order to make sense of our runtime. Based on the analysis done in the work of Etsion et.al [13] there has to be a certain *task decode rate* so that there is optimal utilization of a multi-core system. This rule can be applied in the case of RAM for the *task creation rate*, i.e the frequency with which the runtime system greates tasks. If the *task creation rate* is higher than the *task execution rate* then for a highly parallel application the resources will always have tasks to execute and they will not starve. To maintain the *task creation rate* high enough we can accelerate the task creation cost. Equation 1 shows the maximum optimal task creation cost, $C_{opt}(x)$ for x number of cores.

$$C_{opt}(x) = avg.tasksize/x \tag{1}$$

Using this equation we compute the $C_{opt}(x)$ for each application according to their average task CPU cycles from Table I for $x = 512$ cores. C_{gp} is the cost of task creation when it is performed on a general purpose core, namely the CREATE column as shown on Table I. To have optimal results for each application on systems up to 512 cores, C_{gp} needs to be reduced to $C_{opt}(512)$. Thus the specialized hardware accelerator needs to perform task creation with a ratio $r = C_{gp}/C_{opt}(512)$ times faster than the general purpose core.

We compute r for each application and we observe that for the applications with a large number of per-task CPU cycles and relatively small CREATE cycles (QR512, Dedup, Ferret, Fluidanimate) r is already very close to optimal without the need of a faster hardware accelerator. For the rest of the

TABLE I: Evaluated benchmarks and relevant characteristics

Application	Problem size	#Tasks	Avg task CPU cycles (thousands)	Per task overheads (CPU cycles)			Measured perf. ratio	Parallel model
				CREATE	ALL	DEPS + SCHED		
Cholesky factorization	32K 256Bsize 32K 128Bsize	357 762 2829058	753 110	15221 17992	73286 58820	58065 40828	3.5	deps
QR factorization	16K 512Bsize 16K 128Bsize	11 442 707 265	518 570 3 558	17595 21642	63008 60777	45413 39135	6.8	deps
Blackscholes	native	488 202	348	29141	85438	56297	2.3	data-parallel
Bodytrack	native	329 123	383	9 505	18979	9474	4.2	pipeline
Canneal	native	3 072 002	67	25781	50094	24313	2.0	unstructured
Dedup	native	20 248	1 532	1294	9647	8353	2.7	pipeline
Ferret	native×2	84 002	29 088	38913	98457	59544	3.6	pipeline
Fluidanimate	native	128 502	16 734	30210	94079	64079	3.3	data-parallel
Streamcluster	native	3 184 654	161	6892	13693	6801	3.5	data-parallel

applications, a more powerful hardware is needed. For these applications r ranges from 3× to 197×. In our evaluation we use the value of 16× which is a relatively small value within this range that we consider realistic to implement in hardware.

C. Simulation

To evaluate RAM we make use of the TaskSim simulator [25]. TaskSim is a trace driven simulator, that supports the specification of homogeneous or heterogeneous systems with many cores. By default, TaskSim allows the specification of the amount of cores and supports up to two core types in the case of heterogeneous asymmetric systems. This is done by specifying the number of cores of each type and their difference in performance between the different types (performance ratio) in the TaskSim configuration file.

Our evaluation consists of experiments on both symmetric and asymmetric platforms with the number of cores varying from 8 to 512. In the case of asymmetric systems we simulate the behaviour of an ARM big.LITTLE architecture [16]. To set the correct performance ratio between big and little cores we measure each application on a real ARM big.LITTLE platform and we compare its performance when they run on a little and on a big core. Table I shows the measured performance ratio for each case. The average performance ratio if we take into account our 11 workloads is 3.8. Thus in the specification of the asymmetric systems we use as performance ratio between big and little cores the value 4.

We modify TaskSim so that it features one extra hardware accelerator (per multi-core) responsible for the fast task creation. Apart from the task execution time, our modified simulator tracks the execution time of the runtime overheads. These overheads include: (a) task creation and (b) dependencies resolution and scheduling. When this accelerator executes a task creation job, it finishes it 16× faster than a general purpose core due to the analysis done in Section IV-B. Contrarily, if the accelerator is assigned a task to execute, it executes it 4× slower than a general purpose core.

V. Evaluation

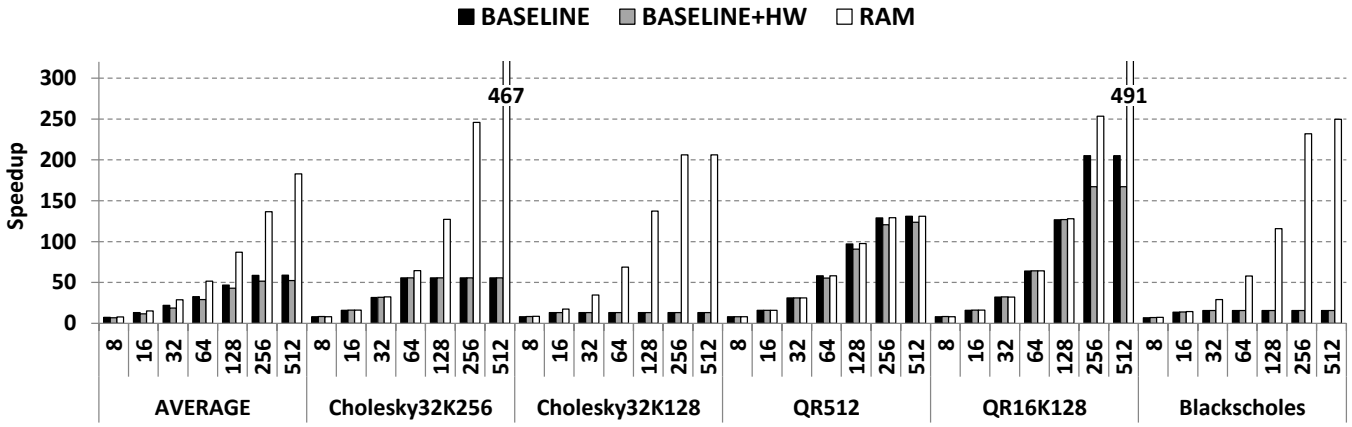
A. Homogeneous systems

Figures 3a and 3b show the speedup over one core of three different scenarios: the baseline runtime system (BASELINE), which is the default runtime without using any external hardware support, the baseline runtime system that uses the external hardware (BASELINE+HW), and RAM that by default uses the external hardware when running on up to 512 cores. In the case of BASELINE+HW the specialized hardware acts as a very slow general purpose core that is additional to the number of cores shown on the x axis. If this core executes a task creation job, it executes it 16× faster, but as it is specialized for this, we assume that when a task is executed on this core it is executed 4× slower than in a general purpose core. The runtime system in this case does not include our modifications that automatically decouple the task creation step for each task. This comparison is used only to show that the baseline runtime is not capable of utilizing the accelerator and in most of the cases performance degrades due to the scheduling with increased heterogeneity of the system.

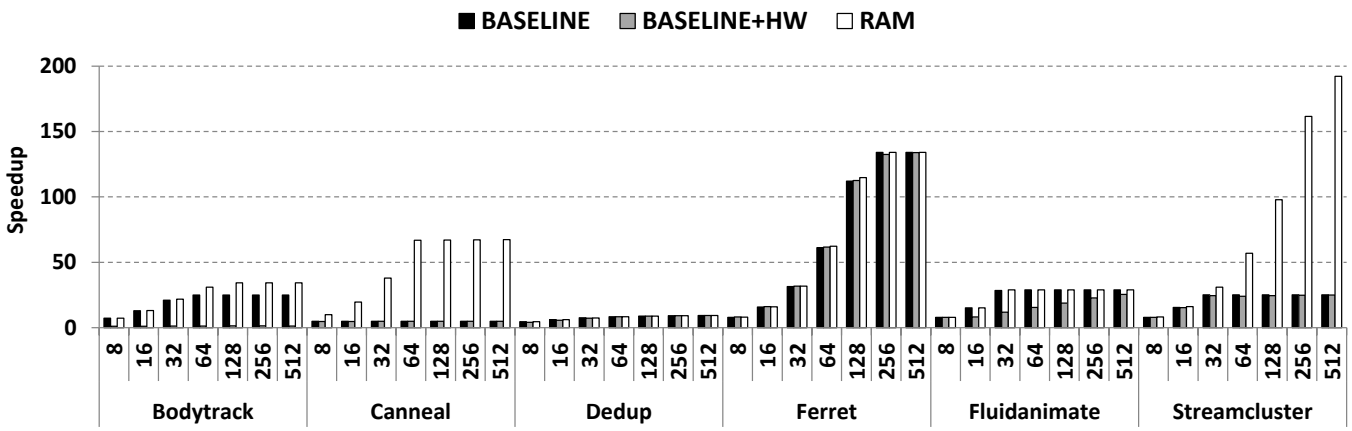
Focusing on the average results first, we can observe that RAM constantly improves the baseline and the improvement is increasing as we increase the number of cores, reaching up to 6× better performance on 512 cores. This is because as we increase the number of cores, the task creation overhead becomes more critical part of the execution time and affects performance even more. So, this becomes the main bottleneck due to which the performance of many applications saturates. RAM overcomes it by automatically detecting and moving task creation on the specialized hardware.

Moving in more detail, we can see that all the applications have a saturation point in speedup. For example Cholesky32K256 saturates on 64 cores, while QR512 on 256 cores. In most cases this saturation in performance comes due to the task creation that is executed sequentially for an important percentage of the execution time (as shown in Figure 1). This is solved with RAM as it efficiently decouples the task creation code and accelerates it leading in higher speedups.

RAM is effective as it either improves performance or it performs as fast as the baseline (there are no slowdowns). The applications that do not benefit (QR512, Ferret, Fluidanimate) are the ones with the highest average per task CPU cycles as shown on Table I. Even if these applications consist of many tasks, the task creation overhead is considered negligible compared to the task cost, so accelerating it does not help



(a) Speedup of CATS, dHEFT and BF on 8 cores compared to the ideal



(b) Cholesky 8×8

Fig. 3: test

much. This can be verified by the results shown for QR16K128 workload. In this case, we use the same input size as QR512 (which is 16K) but we modify the block size, which results in more and smaller tasks. This not only increases the speedup of the baseline, but also shows even higher speedup when running with RAM reaching very close to the ideal speedup and improving the baseline by 2.3×. Modifying the block size for Cholesky, shows the same effect in terms of RAM over baseline improvement. However, for this application, using the bigger block size of 256 is more efficient as a whole. Nevertheless, RAM improves the cases that performance saturates and reaches up to 8.5× improvement for the 256 block-size, and up to 16× for the 128 block-size.

Blackscholes and Canneal, are applications with very high task creation overheads compared to the task size as shown on Table I. This makes them very sensitive to performance degradation due to task creation, thus the application saturates even with limited core counts of 8 or 16 cores. These are the ideal cases for using RAM as such bottlenecks are eliminated and performance is improved by 16× and 14× respectively. However, it is interesting to observe that for Canneal for which the task creation lasts a bit less than half of the task execution time, accelerating it by 16 times is not enough and soon performance saturates at 64 cores. In this case, a more powerful hardware would improve things even more.

Streamcluster has also relatively high task creation overhead compared to the average task cost so improvements are increased as the number of cores is increasing. RAM reaches up to 7.6× improvement in this case.

The performance of Bodytrack saturates on 64 cores for the baseline. However, it does not approach the ideal speedup as its pipelined parallelization technique introduces significant task dependencies that limit parallelism. RAM still improves the baseline by up to 37%. This improvement is low compared to other benchmarks, firstly because of the nature of the application and secondly because Bodytrack introduces nested parallelism, meaning that in this case, task creation is spread among cores and is not becoming a sequential overhead as happens in most of the cases. Thus, in this case it is not correct to look at the CREATE overhead value as this is be parallelized among all cores for the 329 123 tasks of the application.

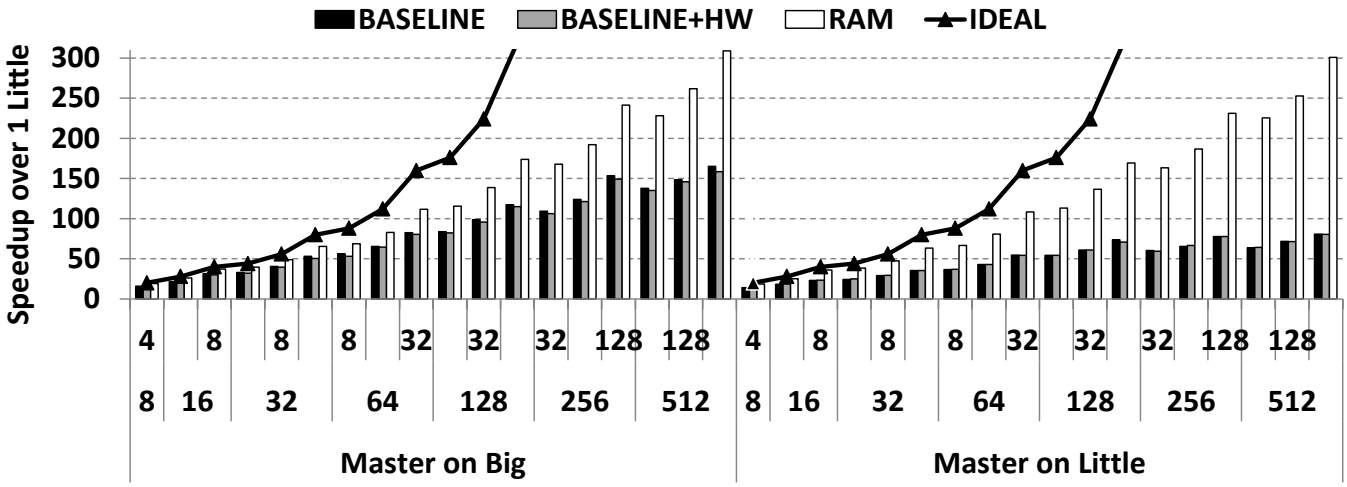


Fig. 4: Average speedup on heterogeneous simulated systems

B. Heterogeneous systems

At this part of the evaluation our system supports two types of general purpose processors, simulating an asymmetric ARM big.LITTLE multi-core processor. In our simulations, we assume that the big cores are four times faster than the little cores of the system. This assumption is based on prior works [10] that have shown that for most applications the performance ratio ranges from 3.5× to 4.5×. In this set-up there are two different ways of executing a task-based application. The first way is to start the application's execution on a big core of the system and the second way is to start the execution on a little core of the system. If we use a big core to launch the application, then this implies that the master thread of the runtime system (the thread that performs the task creation when running with the baseline) runs on a fast core, thus tasks are created faster than when using a slow core as a starting point. We evaluate both approaches and compare the results of the baseline runtime and RAM.

On Figure 4 we plot the average speedup over one little core obtained among all 11 workloads. The chart shows two categories of results on the x axis, separating the cases of the master thread's execution. The numbers of the bottom of x axis show the total number of cores and the numbers above show the number of big cores. The bars represent the average speedup when running with the baseline runtime or with RAM and the line shows the ideal speedup for of each configuration. The ideal speedup is the speedup that we would obtain if we were running an application in parallel assuming zero runtime overheads and no dependencies between tasks, technically unachievable for the real applications of our evaluation. Equation 2 shows how the ideal speedup is computed for our simulated system where the big cores are four times faster than the little cores.

$$ideal\_speedup(big, little) = big \times 4 + little \quad (2)$$

The results show that moving the master thread from a big to a little core degrades performance of the baseline. This is because the task creation becomes even slower so the rest of the cores spend more idle time waiting for the tasks to become ready. RAM improves performance in both cases. Specifically when master runs on big, the average improvement of RAM reaches 2×. When the master thread runs on a little core, RAM improves performance by up to 7×. This is mainly due to the slowdown caused by the migration of master thread on a little core. Using RAM on asymmetric systems achieves approximately similar performance regardless of the type of core that the master thread is running. This makes our proposal more portable for asymmetric systems as the programmer does not have to be concerned about the type of core that the master thread migrates.

C. Comparison to other approaches

As we saw earlier, RAM improves the baseline scheduler by up to 6.3× for 512 cores. In this section we compare RAM with other approaches. To do so, we consider the works of Carbon [22], Task Superscalar [13], Picos++ [28] and Nexus# [11]. We group these works based on the part of the runtime activity they are offloading from the CPU. Carbon and Task Superscalar, are runtime-driven meaning that they both accelerate all the runtime and scheduling parts. The insertion of a task in the TDG, the dependence analysis as well as the scheduling, namely the ready queue manipulation are transferred to the special hardware with these approaches. These overheads are represented on Table I under ALL. The second group of related works that we compare is the dependencies driven, which includes approaches like Picos++ and Nexus#. These works aim to accelerate only the dependence analysis part of the runtime as well as the scheduling that occurs when a dependency is satisfied. For example, when a task finishes execution, and it has produced input for another task, the dependency tracking mechanism is updating the appropriate counters of the reader task and if the task becomes ready, the task is inserted in the ready queue. The insertion into the ready queue is the scheduling that occurs with the dependence

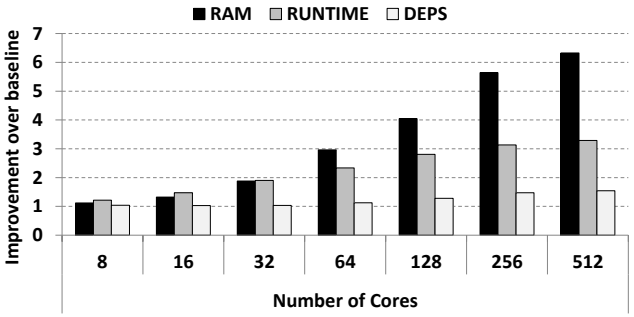


Fig. 5: Average improvement over baseline

analysis.

Figure 5 shows the average improvement in performance for each core count over the performance of the baseline scheduler on the same core count. The RUNTIME represents the runtime driven approaches and the DEPS represents the dependencies driven approaches as described above. Accelerating the scheduling with RUNTIME-driven is as efficient as RAM for a limited number of cores, up to 32. This is because they both accelerate the scheduling that is involved during task to TDG insertion which is an important part of task creation. DEPS driven approaches on the other hand are not as efficient since the task creation step takes place on the master thread. Increasing the number of cores, we observe that the improvement of RUNTIME-driven over the baseline is reduced and stabilized close to 3× while RAM continues to speedup the execution. We attribute this to the fact that serializing the scheduling operations becomes a bottleneck when increasing the number of cores. Serializing the scheduling overheads of up to 32 cores, is efficient but with an increased number of cores it is better to perform scheduling in a distributed manner, just as RAM allows.

DEPS driven works go through the same issue of the serialization of the dependency tracking and the scheduling that occurs at the dependence analysis stage. The reason for the limited performance of DEPS compared to RUNTIME is that DEPS does not accelerate any part of the task creation. Improvement over the baseline is still significant as performance with DEPS is improved by up to 1.54×.

RAM is the most efficient software-hardware co-design approach when it comes to highly parallel applications. On average, it improves the baseline up to 6.3× for homogeneous systems and up to 6.6× for heterogeneous systems. Compared to other state of the art approaches, RAM is more effective on a large number of cores showing higher performance by 54% over RUNTIME driven approaches and by 70% over DEPS driven approaches.

VI. Related Work

Our approach is a new task-based runtime system design that enables the acceleration of task creation to overcome important bottlenecks in performance. Task-based runtime systems have intensively been studied. State of the art task-based runtime systems include the OpenMP [7], OmpSs [12], StarPU [1] and Swan [30]. All these models support tasks and maintain a TDG specifying the inter-task dependencies. This means that the runtime system is responsible for the task creation, the dependence analysis as well as the scheduling of the tasks. However, none of these runtime systems offer automatic offloading of task creation. The fact that task-based programming models are so widely spread makes approaches like ours very important and also gives importance to works that focus on how to boost performance of such programming models with hardware support.

Many works in the research community focus on adding hardware support to boost performance of task-based runtime systems by reducing the runtime overheads. Even if the works focus more on the hardware part of the design, their contributions are very related to our study as we can distinguish which parts of the hardware is more beneficial to be accelerated.

Carbon [22] accelerates the scheduling of tasks by implementing hardware ready queues. Carbon maintains one hardware queue per core and accelerates all possible scheduling overheads by using these queues. Nexus# [11] is also a distributed hardware accelerator capable of executing the *in*, *out*, *inout*, *taskwait* and *taskwait on* pragmas, namely the task dependencies. Unlike Carbon and Nexus, RAM accelerates only task creation. Moreover, ADM [26] is another distributed approach that proposes hardware support for the inter-thread communication to avoid the going through the memory hierarchy. This aims to provide a more flexible design as the scheduling policy can be freely implemented in software. These designs require the implementation of one hardware component for each core of an SoC. Our proposal assumes a centralized hardware unit that is capable of operating without the need to change the SoC.

Task superscalar [13] and Picos++ [28] use a single hardware component to accelerate parts of the runtime system. In the case of Task superscalar, all the parts of the runtime system are transferred to the accelerator. Picos++ [28] is a hardware-software co-design that supports nested tasks. This design enables the acceleration of the inter-task dependencies on a special hardware. Swarm [17] performs speculative task execution. Instead of accelerating parts of the runtime system, Swarm uses hardware support to accelerate speculation. This is different than our design that decouples only task creation.

To summarize, the difference between our work and the prior studies are mainly two:

- Their implementation requires changes in hardware of the SoC, meaning that they need an expensive design where each core of the chip has an extra component, while in our case we need only one hardware unit.
- None of the previous works is aiming in accelerating task creation overheads.

VII. Conclusions

This paper presented the first runtime system that decouples task creation and accelerates it on a special hardware. From our study we obtain interesting conclusions such as:

- The bigger the tasks the less important the task creation overheads
- RAM makes current programming models more portable when running on asymmetric systems as the programmer does not have to worry for the type of core that launches the application
- Accelerating task creation is actually the most critical point to achieve high performance in many core systems as from other approaches we saw that they do not scale so well compared to RAM
- Current task-based programming models suffer from task creation

REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2), 2011.

[2] E. Ayguadé, R. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. Pérez, J. Planas, and E. Quintana-Ortí. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *International Journal of Parallel Programming*, 38(5-6), 2010.

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA*, pages 506–517, 2005.

[5] Barcelona Supercomputing Center. Barcelona Application Repository. Available online on April 18th, 2014.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC*, pages 66:1–66:11, 2012.

[7] O. A. R. Board. OpenMP Specification. 4.5, 2015.

[8] D. Chasapis, M. Casas, M. Moreto, R. Vidal, E. Ayguade, J. Labarta, and M. Valero. PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite. *Trans. Archit. Code Optim.*, 2015.

[9] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *ICS*, pages 329–338, 2015.

[10] K. Chronaki, A. Rico, M. Casas, M. Moretó, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Task scheduling techniques for asymmetric multi-core systems. volume 28, pages 2074–2087, July 2017.

[11] T. Dallou, N. Engelhardt, A. Elhossini, and B. Juurlink. Nexus#: A distributed hardware task manager for task-based programming models. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1129–1138, May 2015.

[12] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21, 2011.

[13] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100, Dec 2010.

[14] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing Power Efficiency with Asymmetric Multicore Systems. *Communications of the ACM*, 52(12), 2009.

[15] P. Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*, 2011.

[16] B. Jeff. big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling. *ARM White Paper*, 2013.

[17] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, Dec 2015.

[18] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223–234, 2012.

[19] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *ISCA*, pages 154–165, 2013.

[20] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, and Others. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, University of Notre Dame, CSE Dept., 2008.

[21] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, pages 64–75, 2004.

[22] 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*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[23] OpenMP architecture review board: Application program interface, 2013.

[24] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In *SC*, 2013.

[25] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels. *ACM Trans. Archit. Code Optim.*, 8(4), 2012.

[26] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, New York, NY, USA, 2010. ACM.

[27] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264, 2009.

[28] X. Tan, J. Bosch, M. Vidal, C. Álvarez, D. Jiménez-González, E. Ayguadé, and M. Valero. General purpose task-dependence management hardware for task-based dataflow programming models. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 244–253, May 2017.

[29] H. Vandierendonck, K. Chronaki, and D. S. Nikolopoulos. Deterministic scale-free pipeline parallelism with hyperqueues. In *SC*, pages 32:1–32:12, 2013.

[30] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *PACT*, pages 1–11, 2011.

[31] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "Codelet" program execution model for exascale machines: Position paper. In *EXADAPT*, pages 64–69, 2011.