TaskGenX: A Hardware-Software Proposal for Accelerating Task Parallelism

Kallia Chronaki*, Marc Casas*, Miquel Moreto*,
Jaume Bosch*, Rosa M. Badia*†

*Barcelona Supercomputing Center,
†Artificial Intelligence Research Institute (IIIA) -
Spanish National Research Council (CSIC)

Abstract. As chip multi-processors (CMPs) are becoming more and more complex, software solutions such as parallel programming models are attracting a lot of attention. Task-based parallel programming models offer an appealing approach to utilize complex CMPs. However, the increasing number of cores on modern CMPs is pushing research towards the use of fine grained parallelism. Task-based programming models need to be able to handle such workloads and offer performance and scalability. Using specialized hardware for boosting performance of task-based programming models is a common practice in the research community. Our paper makes the observation that task creation becomes a bottleneck when we execute fine grained parallel applications with task-based programming models. As the number of cores increases the time spent generating the tasks of the application is becoming more critical to the entire execution. To overcome this issue, we propose TaskGenX. TaskGenX offers a solution for minimizing task creation overheads and relies both on the runtime system and a dedicated hardware. On the runtime system side, TaskGenX decouples the task creation from the other runtime activities. It then transfers this part of the runtime to a specialized hardware. We draw the requirements for this hardware in order to boost execution of highly parallel applications. From our evaluation using 11 parallel workloads on both symmetric and asymmetric systems, we obtain performance improvements up to 15×, averaging to 3.1× over the baseline.

1 Introduction

Since the end of Dennard scaling [13] and the subsequent stagnation of CPU clock frequencies, computer architects and programmers rely on multi-core designs to achieve the desired performance levels. While multi-core architectures constitute a solution to the CPU clock stagnation problem, they bring important challenges both from the hardware and software perspectives. On the hardware side, multi-core architectures require sophisticated mechanisms in terms of coherence protocols, consistency models or deep memory hierarchies. Such requirements complicate the hardware design process. On the software side, multi-core designs significantly complicate the programming burden compared to their single-core predecessors. The different CPUs are exposed to the programmer, who has to make sure to use all of them efficiently, as well as using the memory hierarchy properly by exploiting both temporal and spatial locality. This increasing programming complexity, also known as the Programmability Wall [9], has motivated the advent of sophisticated programming paradigms and runtime system software to support them.

Task-based parallelism [3,6,7,23] has been proposed as a solution to the Programmability Wall and, indeed, the most relevant shared memory programming standards, like OpenMP, support tasking constructs [21]. The task based model requires the programmer to split the code into several sequential pieces, called tasks, as well as explicitly specifying their input and output dependencies. The task-based execution model (or runtime system) consists of a master thread and several worker threads. The master thread goes over the code of the application and creates tasks once it encounters source code annotations identifying them. The runtime system manages the pool of all created tasks and schedules them across the threads once their input dependencies are satisfied. To carry out the task management process, the parallel runtime system creates and maintains a Task Dependency Graph (TDG). In this graph nodes represent tasks and edges are dependencies between them. Once a new task is created, a new node is added to the TDG. The connectivity of this new node is defined by the data dependencies of the task it represents, which are explicitly specified in the application's source code. When the execution of a task finalizes, its corresponding node is removed from the TDG, as well as its data dependencies.

This task-based runtime system constitutes of a software layer that enables parallel programmers to decouple the parallel code from the underlying parallel architecture where it is supposed to run on. As long as the application can be decomposed into tasks, the task-based execution model is able to properly manage it across homogeneous many-core architectures or heterogeneous designs with different core types. A common practice in the high performance domain is to map a single thread per core, which enables the tasks running on that thread to fully use the core capacity. Finally, another important asset of task-based parallelism is the possibility of automatically managing executions on accelerators with different address spaces. Since the input and output dependencies of tasks are specified, the runtime system can automatically offload a task and its dependencies to an accelerator device (E.g. GPU) without the need for specific programmer intervention [8]. Additional optimizations in terms of software prefetching [22] or more efficient coherence protocols [20] can also be enabled by the task-based paradigm.

Despite their advantages, task-based programming models also induce to computational costs. For example, the process of task creation requires the traversal of several indexed tables to update the status of the parallel run by adding the new dependencies the recently created tasks bring, which produces a certain overhead. Such overhead constitutes a significant burden, especially on architectures with several 10's or 100's of cores where tasks need to be created in a very fast rate to feed all of them. This paper proposes the Task Generation Express (TaskGenX) approach. Our proposal suggests that the software and hardware are designed to eliminate the most important bottlenecks of task-based parallelism without hurting their multiple advantages. This paper focuses on the software part of this proposal and draws the requirements of the hardware design to achieve significant results. In particular, this paper makes the following contributions beyond the state-of-the-art:

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

Listing 1.1: Compiler generated pseudo-code equivalence for task annotation.

```
void createTask(DepList dList, Task t,
                Task parent, Data args) {
  initAndSetupTask(task1, parent, args);
  insertToTDG(dList, task1);
}
```

Listing 1.2: Pseudo-code for task creation.

– A new parallel task-based runtime system that decouples the most costly routines from the other runtime activities and thus enables them to be off-loaded to specific-purpose helper cores.
– A detailed study of the requirements of a specific-purpose helper core able to accelerate the most time consuming runtime system activities.
– A complete evaluation via trace-driven simulation considering 11 parallel OpenMP codes and 25 different system configurations, including homogeneous and heterogeneous systems. Our evaluation demonstrates how TaskGenX achieves average speedups of 3.1× when compared against currently use state-of-the-art approaches.

The rest of this document is organized as follows: Section 2 describes the task-based execution model and its main bottlenecks. Section 3 describes the new task-based runtime system this paper proposes as well as the specialized hardware that accelerates the most time-consuming runtime routines. Section 4 contains the experimental set-up of this paper. Section 5 describes the evaluation of TaskGenX via trace-driven simulation. Finally, Section 6 discusses related work and Section 7 concludes this work.

2 Background and Motivation

2.1 Task-based Programming Models

Task-based parallel programming models [3,6,7,23], 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. One type of these annotations is the task annotations with dependency tracking which OpenMP [1] supports since its 4.0 release [21]. 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 1.1. `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 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 1.1 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 1.2 shows the pseudo-code for the task creation step within the runtime. The `createTask` function is first 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. The TDG is a distributed and dynamic graph structure that the runtime uses to keep the information about the current tasks of the application. The insertion of a task in the TDG 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. Listing 1.3 shows the pseudo-code for the TDG insertion. If for a task the `dList` is empty (line 2), 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 3). Each entry of `dList` contains the actual memory address as well as the access type (read, write or read-write). The runtime keeps a distributed unified dependency tracking structure, the `depMap` 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 8). If the memory address of an entry of the `dList` is not represented in the `depMap`, it is being added as shown in line 9. 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 10-15).

To reduce the lookup into the `depMap` calls, every time the contents of a memory address are modified, the tasks keep track of their *successors* as well as

```
void insertToTDG(DepList dList, Task t) {
 if( dList is empty ) {
   readyQ->push(t);
   return;
 }
 Dependency entry;
 for( d in dList ) {
   entry = depMap[d.address()];
   if(entry==NULL) depMap.add(entry, t);
   if(d.accessType() == "write")
     entry.addLastWriter(t);
   if(d.accessType() == "read") {
     entry.addReader(t);
     entry.lastWriter()->addSuccessor(t);
   }
 }
}
```

Listing 1.3: Pseudo-code for TDG insertion

the number of *predecessors*. The *successors* of a task are all the tasks with inputs depending 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 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 1.2.

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 goes to the *finished* state, the runtime has to perform some actions in order to prepare the successor tasks for execution. These actions are described in Listing 1.4. 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). The runtime activity 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*. At this point the runtime prepares all the appropriate task and dependency tracking data structures as well as inserts the task into the TDG. The second change occurs when a task from being *created* it becomes *ready*; this implies that the input dependencies of this task are satisfied so the runtime schedules and inserts the task into the ready queue. The third change occurs when a running task finishes execution. In this case, following our task states, the task from being *ready* it becomes *finished*; this is followed by the runtime updating the dependency tracking data structures and scheduling possible successor tasks that become ready. For the rest of the paper we will refer to the first state change runtime activity as the task creation overheads (*Create*). For the runtime activity that takes place for the following two state changes (and includes scheduling and dependence analysis) we will use the term runtime overheads (*Runtime*).

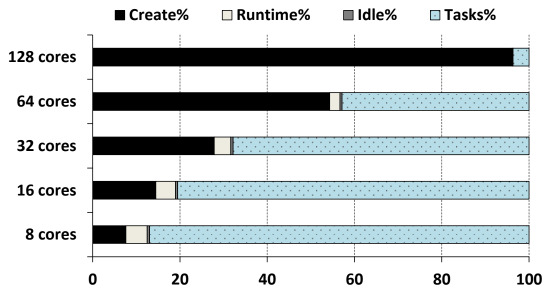


Fig. 1: Master thread activity for Cholesky as we increase the number of cores.

2.2 Motivation

Figure 1 shows the runtime activity of the master thread during the execution of the Cholesky[1] benchmark on 8, 16, 32, 64 and 128 cores[2]. The execution time represented here is the wall clock time during the parallel region of the benchmark. 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 invariant of core count: the more we reduce the application's execution time by adding resources the more important this step becomes in terms of execution time. In contrast, the task execution time percentage is decreased as we increase the number of cores because the computational activity is being shared among more resources. One way to reduce the task creation overhead is by introducing nested parallelism. In this programming technique, every worker thread is able to generate tasks thus the task creation is spread among cores and its overhead is reduced. However, not all applications can be implemented with this parallelization technique and there are very few applications using this scheme. *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.

3 Task Generation Express

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 TaskGenX we use the OpenMP programming model [1], [21]. The base of our implementation is the Nanos++ runtime system responsible for the parallel execution and it is used in this paper as a replacement of the entire OpenMP's default runtime.

[1] Details about the benchmarks used are in Section 4
[2] The experimental set-up is explained in Section 4

```
void task_finish(Task *t) {
  depMap.removeReaderWriter(t);
  if(t->successors.empty()) delete t;
  else {
    for( succ in t->successors ) {
      succ.decreasePredecessors();
      if(succ.numPredecessors == 0)
        readyQ->push(succ);
    }
  }
}
```

Listing 1.4: Pseudo-code for task_finish runtime activity.

Nanos++ [5] 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[3]. 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 2.1 to keep the data dependency structures consistent. Moreover, it pushes the tasks that become ready to the *TaskQ*.

3.1 Implementation

TaskGenX relieves the master and worker threads from the intensive work of task creation by offloading it on the specialized hardware. Our runtime, 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 identifier 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. Instead of querying the ready queue for tasks, the SRT looks for runtime activity requests in the Runtime Requests Queue (*RRQ*) and if there are requests, it executes them.

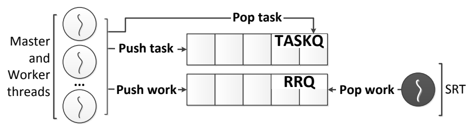
Figure 2a shows the communication infrastructure between threads within TaskGenX. Our system maintains two queues; the Ready Task Queue (*TaskQ*) and the Runtime Requests Queue (*RRQ*). The *TaskQ* is used to keep the tasks that are ready for execution. The *RRQ* is used to keep the pending runtime activity 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 1.2 and described in Section 2.1. TaskGenX decouples the execution of `createTask`

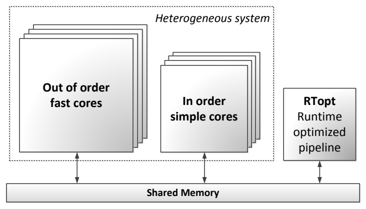
[3] Nanos++ also supports nested parallelism so any of the worker threads can potentially create tasks. However the majority of the existing parallel applications are not implemented using nested parallelism.

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

Listing 1.5: Pseudo-code for the SRT loop.



(a) Communication mechanism between master/workers and SRT threads.



(b) SoC architecture including three types of cores: out of order, in-order and RTopt.

Fig. 2

from the master thread. To do so, TaskGenX 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 1.2. 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 1.5 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). If there is a pending creation request, the SRT calls the `executeRequest` (line 4), which extracts the appropriate task creation data from the creation request and performs the task creation by calling the `createTask` described in Listing 1.2. When the parallel region is over, the runtime system informs the SRT to stop execution. This is when the SRT exits and the execution finishes (line 6).

3.2 Hardware Requirements

As described in the previous section, TaskGenX assumes the existence of a specialized hardware that accelerates the task creation step. The goal of this paper is not to propose a detailed micro-architecture of the specialized hardware; instead we sketch the high-level hardware requirements for the TaskGenX set-up, in the hope to be an insightful and useful influence for hardware designers. The SRT is bound to the task creation accelerator and executes the requests in the RRQ. Previous works have proposed custom accelerators for the runtime activity [12, 15, 18, 19, 26, 27]. These proposals significantly accelerate (up to three orders of magnitude) different bottlenecks of the runtime system[4]. These special purpose designs can only execute runtime system activity.

As an alternative, in our envisioned architecture we propose to have a general purpose core that has been optimized to run the runtime system activity more efficiently. The runtime optimized (RTopt) core can be combined with both homogeneous or heterogeneous systems and accelerate the runtime activity. Figure 2b shows the envisioned architecture when RTopt is combined with an asymmetric heterogeneous system. This architecture has three core types that consist of simple in-order cores, fast out-of-order cores and an RTopt core for the SRT. RTopt can optimize its architecture, having a different cache hierarchy, pipeline configuration and specialized hardware structures to hold and process the SRT. As a result, the RTopt executes the SRT much faster than the other cores. The RTopt can also execute tasks, but will achieve limited performance compared to the other cores as its hardware structures have been optimized for a specific software (the SRT).

To evaluate our approach we study the requirements of the RTopt in order to provide enough performance for TaskGenX. Based on the analysis by Etsion et al. [15], there is a certain *task decode rate* that leads to optimal utilization of the multi-core system. This rule can be applied in the case of TaskGenX for the *task creation rate*, i.e. the frequency of task generation of the runtime system. 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 remain idle. To achieve a high *task creation rate*, we can accelerate the task creation cost. Equation 1 shows the maximum optimal task creation cost, $C_{opt}(x)$ in order to keep x cores busy, without starving due to task creation.

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

If C_{gp} is the cost of task creation when it is performed on a general purpose core, then the RTopt has to achieve a speedup of $r = C_{gp}/C_{opt}(x)$ to achieve full utilization of the system. Section 4.2 performs an analysis based on these requirements for the evaluated applications. As we will see in Section 4.2, a modest and implementable value of $r = 16\times$ is enough to significantly accelerate execution on a 512-core system.

Finally, if TaskGenX executes on a regular processor without the RTopt core, the SRT is bound to a regular core without any further modification. In this scenario, applications will not significantly benefit from having a separate SRT.

4 Experimental Methodology

4.1 Applications

Table 1 shows the evaluated applications, the input sizes used, and their characteristics. All applications are implemented using the OpenMP programming model [21]. We obtain Cholesky and QR from the BAR repository [4] and we use the implementations of the rest of the benchmarks from the PARSECSs suite [10]. More information about these applications can be found in [10] and [11]. As the number of cores in SoCs is increasing, so does the need of available task

[4] Section 6 further describes these proposals.

Table 1: Evaluated benchmarks and relevant characteristics

Application	Problem size	#Tasks	Avg task CPU cycles (thousands)	Per task overheads (CPU cycles)			Measured perf. ratio	r	Parallel model
				Create	All	Deps + Sched			
Cholesky factorization	32K 256	357 762	753	15221	73286	58065	3.5	10.34	dependencies
	32K 128	2829058	110	17992	58820	40828		83.74	
QR factorization	16K 512	11 442	518 570	17595	63008	45413	6.8	0.01	dependencies
	16K 128	707 265	3 558	21642	60777	39135		3.11	
Blackscholes	native	488 202	348	29141	85438	56297	2.3	42.87	data-parallel
Bodytrack	native	329 123	383	9 505	18979	9474	4.2	12.70	pipeline
Canneal	native	3 072 002	67	25781	50094	24313	2.0	197.01	unstructured
Dedup	native	20 248	1 532	1294	9647	8353	2.7	0.43	pipeline
Ferret	native×2	84 002	29 088	38913	98457	59544	3.6	0.68	pipeline
Fluidanimate	native	128 502	16 734	30210	94079	64079	3.3	0.91	data-parallel
Streamcluster	native	3 184 654	161	6892	13693	6801	3.5	21.91	data-parallel

parallelism [25]. We choose the input sizes of the applications so that they create enough fine-grained tasks to feed up to 512 cores. The number of tasks per application and input as well as the average per-task CPU cycles can be found on Table 1.

4.2 Simulation

To evaluate TaskGenX we make use of the TaskSim simulator [16, 24]. TaskSim is a trace driven simulator, that supports the specification of homogeneous or heterogeneous systems with many cores. The tracing overhead of the simulator is less than 10% and the simulation is accurate as long as there is no contention in the shared memory resources on a real system [16]. 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 [17]. To set the correct performance ratio between big and little cores, we measure the sequential execution time of each application on a real ARM big.LITTLE platform when running on a little and on a big core. We use the Hardkernel Odroid XU3 board that includes a Samsung Exynos 5422 chip with ARM big.LITTLE. The big cores run at 1.6GHz and the little cores at 800MHz. Table 1 shows the measured performance ratio for each case. The average performance ratio among our 11 workloads is 3.8. Thus in the specification of the asymmetric systems we use as performance ratio the value 4.

To simulate our approaches using TaskSim we first run each application/input in the TaskSim trace generation mode. This mode enables the online tracking of task duration and synchronization overheads and stores them in a trace file. To perform the simulation, TaskSim uses the information stored in the trace file and executes the application by providing this information to the runtime system. For our experiments we generate three trace files for each application/input combination on a Genuine Intel 16-core machine running at 2.60GHz.

We modify TaskSim so that it features one extra hardware accelerator (per multi-core) responsible for the fast task creation (the RTopt). Apart from the task duration time, our modified simulator tracks the duration of the runtime overheads. These overheads include: (a) task creation, (b) dependencies resolution, and (c) scheduling. The RTopt core is optimized to execute task creation faster than the general purpose cores; to determine how much faster a task creation job is executed we use the analysis performed in Section 3.2.

Using Equation 1, we compute the $C_{opt}(x)$ for each application according to their average task CPU cycles from Table 1 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 shown on Table 1. 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 a general purpose core.

We compute r for each application shown on Table 1. 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 very close to zero, meaning that the task creation cost (C_{gp}) is already small enough for optimal task creation without the need of a faster hardware accelerator. For the rest of the applications, a more powerful hardware is needed. For these applications r ranges from $3\times$ to $197\times$. Comparing r to the measured performance ratio of each application we can see that in most cases accelerating the task creation on a big core would not be sufficient for achieving higher task creation rate. In our experimental evaluation we accelerate task creation in the RTopt and we use the ratio of $16\times$ which is a relatively small value within this range that we consider realistic to implement in hardware. The results obtained show the average results among three different traces for each application-input.

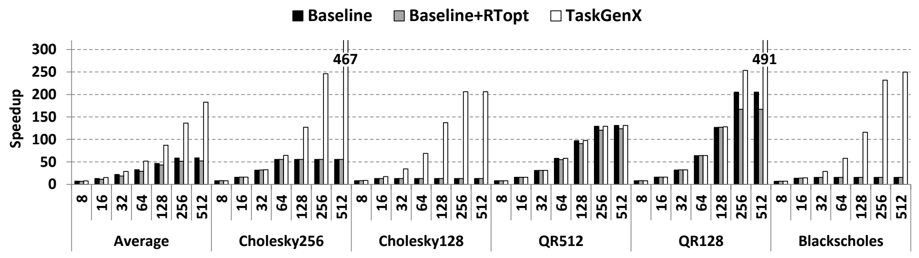
5 Evaluation

5.1 Homogeneous Multicore Systems

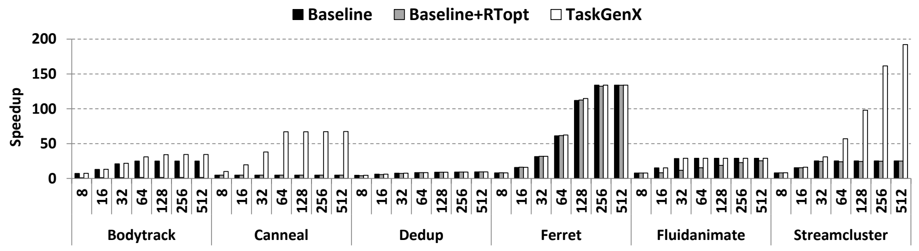
Figures 3a and 3b show the speedup over one core of three different scenarios:

- *Baseline*: the Nanos++ runtime system, which is the default runtime without using any external hardware support
- *Baseline+RTopt*: the Nanos++ runtime system that uses the external hardware as if it is a general purpose core
- *TaskGenX*: our proposed runtime system that takes advantage of the optimized hardware

We evaluate these approaches with the TaskSim simulator for systems of 8 up to 512 cores. In the case of Baseline+RTopt the specialized hardware acts as a 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 $16\times$ faster, but as it is specialized for this, we assume that when a task is executed on this core it is executed $4\times$ 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. The comparison against the Baseline+RTopt is used only to show that the baseline runtime is not capable of effectively utilizing the accelerator. In most of the cases having this additional hardware without the appropriate runtime support results in slowdown as the tasks are being executed slower on the special hardware.



(a)



(b)

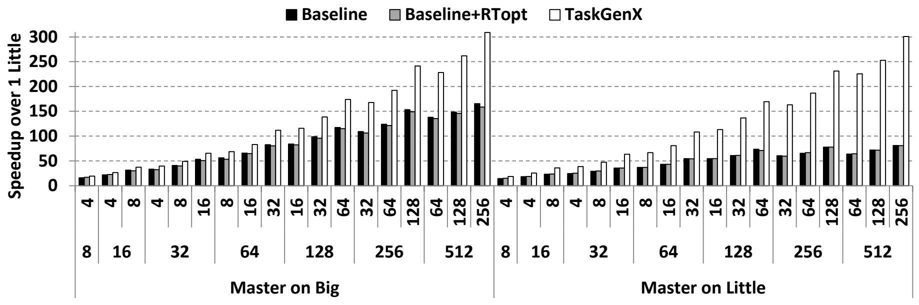
Fig. 3: Speedup of TaskGenX compared to the speedup of Baseline and Baseline+RTopt for each application for systems with 8 up to 512 cores. The average results of (a) show the average among all workloads shown on (a) and (b)

Focusing on the average results first, we can observe that TaskGenX constantly improves the baseline and the improvement is increasing as the number of cores is increased, reaching up to 3.1× improved 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. TaskGenX overcomes it by automatically detecting and moving task creation on the specialized hardware.

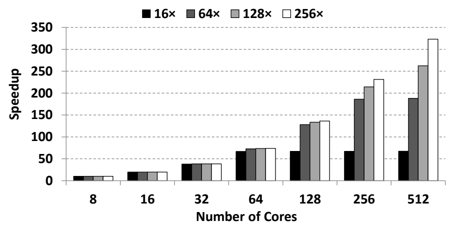
Moving in more detail, we can see that for all applications the baseline has a saturation point in speedup. For example Cholesky256 saturates on 64 cores, while QR512 on 256 cores. In most cases this saturation in performance comes due to the sequential task creation that is taking place for an important percentage of the execution time (as shown in Figure 1). TaskGenX solves this as it efficiently decouples the task creation code and accelerates it leading to higher speedups.

TaskGenX 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 1. Dedup also does not benefit as the per task creation cycles are very low compared to its average task size. 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 much.

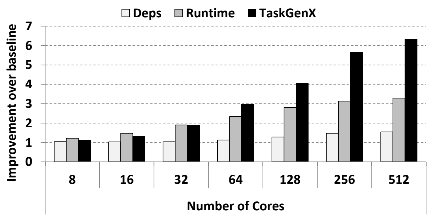
This can be verified by the results shown for QR128 workload. In this case, we use the same input size as QR512 (which is 16K) but we modify the block size,



(a) Average speedup among all 11 workloads on heterogeneous simulated systems. The numbers at the bottom of x axis show the total number of cores and the numbers above them show the number of big cores. Results are separated depending on the type of core that executes the master thread: a big or little core.



(b) Canneal performance as we modify r



(c) Average improvement over baseline. X-axis shows the number of cores. For each case an RTopt core is used additionally to the number of cores.

Fig. 4

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 TaskGenX 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 TaskGenX over baseline improvement. However, for this application, using the bigger block size of 256 is more efficient as a whole. Nevertheless, TaskGenX 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 1. This makes them very sensitive to performance degradation due to task creation. As a result their performance saturates even with limited core counts of 8 or 16 cores. These are the ideal cases for using TaskGenX as such bottlenecks are eliminated and performance is improved by 15.9× and 13.9× respectively. However, 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. Figure 4b shows how the performance of Canneal is affected when modifying the task creation performance ratio, r between the specialized hardware and general purpose. Using a hardware that performs task creation close to 256× faster than the general purpose core leads to higher improvements.

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. TaskGenX 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. TaskGenX 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. With nested parallelism task creation is being spread among cores so it is not becoming a sequential overhead as happens in most of the cases. Thus, in this case task creation is not as critical to achieve better results.

5.2 Heterogeneous Multicore Systems

At this part of the evaluation our system supports two types of general purpose processors, simulating an asymmetric multi-core processor. The asymmetric system is influenced by the ARM big.LITTLE architecture [17] that consists of big and little cores. In our simulations, we consider that the big cores are four times faster than the little cores of the system. This is based on the average measured performance ratio, shown on Table 1, among the 11 workloads used in this evaluation.

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 load 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 TaskGenX.

Figure 4a plots the average speedup over one little core obtained among all 11 workloads for the Baseline, Baseline+RTopt and TaskGenX. The chart shows two categories of results on the x axis, separating the cases of the master thread's execution. The numbers at the bottom of x axis show the total number of cores and the numbers above show the number of big cores.

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. TaskGenX improves performance in both cases. Specifically when master runs on big, the average improvement of TaskGenX reaches 86%. When the master thread runs on a little core, TaskGenX improves performance by up to 3.7×. This is mainly due to the slowdown caused by the migration of master thread on a little core. Using TaskGenX 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.

5.3 Comparison to Other Approaches

As we saw earlier, TaskGenX improves the baseline scheduler by up to 6.3× for 512 cores. In this section we compare TaskGenX with other approaches. To do so, we consider the proposals of Carbon [19], Task Superscalar [15], Picos++ [27] and Nexus# [12]. We group these proposals 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 task creation, dependence analysis as well as the scheduling, namely the ready queue manipulation, are transferred to the RTopt with these approaches. These overheads are represented on Table 1 under ALL. For the evaluation of these approaches one RTopt is used optimized to accelerate all the runtime activities. The second group of related designs that we compare against is the dependencies-driven, which includes approaches like Picos++ and Nexus#. These approaches aim to accelerate only the dependence analysis part of the runtime as well as the scheduling that occurs when a dependency is satisfied. The RTopt in this case is optimized to accelerate these activities. 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 analysis. These overheads are represented on Table 1 under *Deps+Sched*.

Figure 5 shows the average improvement in performance for each core count over the performance of the baseline scheduler on the same core count. *Runtime* represents the runtime driven approaches and the *Deps* represents the dependencies driven approaches as described above. X-axis shows the number of general purpose cores; for every core count one additional RTopt core is used.

Accelerating the scheduling with *Runtime*-driven is as efficient as TaskGenX for a limited number of cores, up to 32. This is because they both accelerate task creation which is an important bottleneck. *Deps*-driven approaches on the other hand are not as efficient since in this case the task creation step takes place on the master thread.

Increasing the number of cores, we observe that the improvement of the *Runtime*-driven over the baseline is reduced and stabilized close to 3.2× while TaskGenX continues to speedup the execution. Transferring all parts of the runtime to RTopt with the *Runtime*-driven approaches, leads to the serialization of the runtime. Therefore, all scheduling operations (such as enqueue, dequeue of tasks, dependence analysis etc) that typically occur in parallel during runtime are executed sequentially on the RTopt. Even if RTopt executes these operations faster than a general purpose core, serializing them potentially creates a bottleneck as we increase the number of cores. TaskGenX does not transfer other runtime activities than the task creation, so it allows scheduling and dependence analysis operations to be performed in a distributed manner.

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 1.5×.

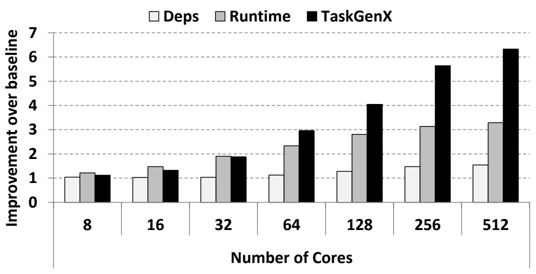


Fig. 5: Average improvement over baseline. X-axis shows the number of cores. For each case an RTopt core is used additionally to the number of cores.

TaskGenX is the most efficient software-hardware co-design approach when it comes to highly parallel applications. On average, it improves the baseline by up to 3.1× for homogeneous systems and up to 3.7× for heterogeneous systems. Compared to other state of the art approaches, TaskGenX 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.

6 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 [1], OmpSs [14], StarPU [2] and Swan [28]. 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 offers 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 adding hardware support to boost performance of task-based runtime systems. Even if the works focus more on the hardware part of the design, their contributions are very relative to our study as we can distinguish which parts of the hardware is more beneficial to be accelerated.

Carbon [19] 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# [12] 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, TaskGenX accelerates only task creation. Moreover, ADM [25] is another distributed approach that proposes hardware support for the inter-thread communication to avoid 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 a 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 [15] and Picos++ [27] 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++ [27] 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 [18] 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.

Our work diverges to prior studies for two main reasons:

– The implementation of prior studies requires changes in hardware of the SoC. This means that they need an expensive design where each core of the chip has an extra component. Our proposal offers a much cheaper solution by requiring only a single specialized core that, according to our experiments, can manage the task creation for 512-core SoCs.
– None of the previous works is aiming at accelerating exclusively task creation overheads. According to our study task creation becomes the main bottleneck as we increase the number of cores and our study is the first that takes this into account.

7 Conclusions

This paper presented TaskGenX, the first software-hardware co-design that decouples task creation and accelerates it on a runtime optimized hardware. Contrarily to previous works, our paper makes the observation that task creation is a significant bottleneck in parallel runtimes. Based on this we implemented TaskGenX on top of the OpenMP programming model. On the hardware side, our paper sets the requirements for the RTopt in order to achieve optimal results and proposes an asymmetric architecture that combines it with general purpose cores.

Based on this analysis we evaluate the performance of 11 real workloads using our approach with TaskSim simulator. Accelerating task creation, TaskGenX achieves up to 15.8× improvement (Cholesky128) over the baseline for homogeneous systems and up to 16× (Blackscholes) on asymmetric systems when the application is launched on a little core. Using TaskGenX on asymmetric systems offers a portable solution, as the task creation is not affected by the type of core that the master thread is bound to.

We further showed that for some cases like Canneal where task creation needs to be accelerated as much as 197× in order to steadily provide enough created tasks for execution. However, even by using a realistic and implementable hardware approach that offers 16× speedup of task creation, achieves satisfactory results as it improves the baseline up to 14×.

Comparing TaskGenX against other approaches such as Carbon, Nexus, Picos++ or TaskSuperscalar that manage to transfer different parts of the runtime to the RTopt proves that TaskGenX is the most minimalistic and effective approach. Even if TaskGenX transfers the least possible runtime activity to the RTopt hardware it achieves better results. This implies that TaskGenX requires a less complicated hardware accelerator, as it is specialized for only a small part of the runtime, unlike the other approaches that need specialization for task creation, dependency tracking and scheduling.

We expect that combining TaskGenX with an asymmetry-aware task scheduler will achieve even better results, as asymmetry introduces load imbalance.

References

1. OpenMP architecture review board. OpenMP Specification. 4.5, 2015.
2. 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.
3. 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.
4. Barcelona Supercomputing Center. https://pm.bsc.es/projects/bar BSC Application Repository. Available online on April 18th, 2014.
5. Barcelona Supercomputing Center. Nanos++.
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. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, 1995.
8. J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *IPDPS*, pages 557–568, 2012.
9. B. Chapman. *The Multicore Programming Challenge*, pages 3–3. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
10. 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.
11. 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.
12. T. Dallou, N. Engelhardt, A. Elhossini, and B. Juurlink. Nexus#: A distributed hardware task manager for task-based programming models. In *IPDPS*, pages 1129–1138, 2015.
13. R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. In *Solid-State Circuits, IEEE Journal of*, volume 9, 1974.
14. 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.
15. 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 *MICRO*, pages 89–100, 2010.
16. T. Grass, C. Allande, A. Armejach, A. Rico, E. Ayguadé, J. Labarta, M. Valero, M. Casas, and M. Moreto. Musa: A multi-level simulation approach for next-generation hpc machines. In *SC16*, pages 526–537, Nov 2016.
17. B. Jeff. big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling. *ARM White Paper*, 2013.
18. M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *MICRO*, pages 228–241, 2015.
19. S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA*, pages 162–173, 2007.
20. M. Manivannan and P. Stenström. Runtime-guided cache coherence optimizations in multi-core architectures. In *IPDPS*, 2014.
21. OpenMP architecture review board: Application programming interface 4.5. 2015.

22. V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 325–334, New York, NY, USA, 2013. ACM.
23. J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.
24. 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.
25. 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, 2010.
26. M. Själander, A. Terechko, and M. Duranton. A look-ahead task management unit for embedded multi-core architectures. In *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 149–157, 2008.
27. 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 *IPDPS*, pages 244–253, 2017.
28. H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *PACT*, pages 1–11, 2011.