Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models

Tamer Dallou, Ahmed Elhossini, Ben Juurlink
Embedded Systems Architecture
Technische Universität Berlin
Einsteinufer 17, 10587 Berlin, Germany
{dallou, ahmed.elhossini, b.juurlink}@tu-berlin.de

Nina Engelhardt
Department of Electrical and Electronic Engineering
The University of Hong Kong
Pokfulam, Hong Kong
nengel@hku.hk

***Abstract*—In the era of multicore systems, it is expected that the number of cores that can be integrated on a single chip will be 3-digit. The key to utilize such a huge computational power is to extract the very fine parallelism in the user program. This is non-trivial for the average programmer, and becomes very hard as the number of potential parallel instances increases. Task-based programming models such as OmpSs are promising, since they handle the detection of dependencies and synchronization for the programmer. However, state-of-the-art research shows that task management is not cheap, and introduces a significant overhead that limits the scalability of OmpSs. Nexus# is a hardware accelerator for the OmpSs runtime system, which dynamically monitors dependencies between tasks. It is fully synthesizable in VHDL, and has a distributed task graph model to achieve the best scalability. Supporting tasks with arbitrary number of parameters and any dependency pattern, Nexus# achieves better performance than Nanos, the official OmpSs runtime system, and scales well for the H264dec benchmark with very fine grained tasks, among other benchmarks from the Starbench suite.**

***Index Terms*—hardware support; task manager; hardware task scheduler; parallel programming; task graph; data flow.**

I. Introduction

In recent years, multi-core/many-core processing is gaining focus as increasing the frequency of a single core is nor more possible due to the power wall. New programming models are required for software development for multi-core systems that can encapsulate the complexity of the underlying system and allow the programmer to express parallelism in his/her application without major modifications. Examples include Google's MapReduce [9], Intel's TBB [16], StarSs [15] and OpenMP [6].

Task based programming models allow exploring parallelism in the application with minimal modifications through the use of various pragmas.The programmer specifies the code blocks that can be potentially executed in parallel by annotating the application with simple pragmas, specifying the memory footprints of these code blocks. These code blocks are called *tasks*. Dependencies between tasks is determined using the *input/output* parameters of each task. The RTS checks all inputs and outputs for tasks submitted for execution and then when all dependencies are resolved, tasks are marked as ready and submitted to one of the free cores for execution. The dependency resolution is shown to be a bottleneck in the RTS of task based programming models such as StarSs [14], and OmpSs [11]. This bottleneck affects the scalability of applications on multi-core systems as no significant speed up can be achieved by increasing the number of processing cores to run parallel tasks. This is mainly because the RTS will spend more time processing dependencies between tasks as more tasks become ready for execution, which is expected when we increase the number of processing cores.

Accelerating the dependency resolution will increase the scalability of applications employing task based programming models to allow better utilization of multi-core systems. Building custom hardware accelerators is one of the solutions that was investigated in literature to reduce the overhead of dependency resolution and increase the overall scalability of parallel applications. The hardware accelerator introduced in [7] named *Nexus++* and analyzed in detail in [11], presented a significant enhancement in terms of scalability for various applications. Nexus++ is used to speed up the dependency resolution in OmpSs's RTS. Nexus++ use a single task graph architecture to resolve dependencies. Tasks processed by Nexus++ can have various dependency patterns and it is possible that tasks can have arbitrary number of *input/output* parameters. Using a single task graph limits the expected scalability that can be obtained by Nexus++. Although significant enhancements were introduced by Nexus++, using multiple task graphs to process tasks in parallel can introduce further enhancements in terms of scalability. On the other hand, dependency between tasks is expressed in OmpSs by several pragmas. Nexus++ had a limited support for these pragmas which limited the performance of some applications employing other pragmas, namely H264dec, which uses the *taskwait on* barrier pragma.

This paper presents Nexus#, a hardware accelerator for task based, data flow programming models in general, and for OmpSs in its current prototype. The main contribution in this paper to parallel programming is introducing a scalable distributed task graph manager, where multiple tasks can be analyzed in parallel. It is implemented as a synthesizable VHDL prototype, aiming at the on-chip integrability with future multicore SoCs as a co-processor. It support the *taskwait on* barrier pragma, among other pragmas, which is expected to enhance the performance of various applications. Several experiments were performed to evaluate the performance of Nexus# and compare it to Nexus++ and other dependency resolution schemes found in the literature.

The remainder of the paper is organized as follows. Section II gives a brief overview of the OmpSs programming model, as well as the related work. Discussion on the design of Nexus++, our baseline task manager, along its execution pipeline is presented in Section III. Then we introduce the new distributed architecture of Nexus# in Section IV. In Section V the simulation environment and the employed benchmarks are described. The evaluation results are presented in Section VI, and conclusions are drawn in Section VII.

II. Background

A. OmpSs

OmpSs is a task-based programming model that enables uncomplicated exploitation of task-level parallelism. OmpSs provides programmers with *pragmas*, annotations added to the sequential code, to identify pieces of code that can potentially run in parallel. The programmer does not need to reason about synchronization between the tasks, as this is done implicitly by the OmpSs runtime system (RTS). Listing 1 shows an example of exploiting parallelism using pragmas.

```
MB_type* X[NB_WIDTH][NB_HEIGHT];
//MB_type: a data str. that rep. MB dependencies.
#pragma omp task input(left, upright) inout(this)
void decode(MB_type* left, MB_type* upright,
            MB_type* this){...}
void main(){
 int i, j;
 init_matrix(X) ;
 for(i=0; i<NB_WIDTH; i++)
  for(j=0; j<NB_HEIGHT; j++)
   decode(X[i][j-1], X[i-1][j+1], X[i][j]);
 #pragma omp taskwait
}
```

Listing 1. OmpSs example of macroblock wavefront decoding in H.264

In this example, the function *decode()* is called inside a nested loop, processing the elements of matrix X. Calculating *decode()* for each element requires the results of the *decode()* call on the left and upper-right cells.This example represents macroblock wavefront decoding in H.264 [18], for one 1920×1088 frame in blocks of 16×16, and it is one of the benchmarks used to evaluate Nexus#.

When a function declaration is annotated with the *omp task* pragma, any calls to the function are turned into task submissions. The in- and output parameters of the task should also be specified in the annotation as shown in Listing 1. This permits the RTS to detect dependencies between tasks and launch them only when all their input data is available. At the end, a *taskwait* pragma makes the thread wait for completion of the submitted tasks.

A source-to-source compiler transforms the annotated function calls to runtime library calls, which generate a task out of each function call, and add it to the task graph. In the example of Listing 1, every time the function *decode()* is called, a task is generated. The call returns immediately, allowing the submission of more tasks concurrently to their execution.

Having identified the tasks and the direction of their parameters, the OmpSs environment builds the task graph at run time, and the task-level parallelism is detected and exploited.

B. Related Work

Parallel programming has always been an active field, even before the breakthrough of multi/many core processors. Nowadays, it receives more importance as the target has shifted from programming supercomputers, to normal consumer devices.

The literature has several programming models that aims at the scalability of user applications. Most of them, however, assume independent tasks and are optimized for a certain application, a certain platform, or both. For example, Carbon [13] assumes independent tasks and uses hardware queues to retrieve tasks with low latency. Al-Kadi et al. [1] proposed a hardware task scheduler optimized for H.264 decoding. It requires, however, that the programmer specifies the dependencies between blocks.

Google's MapReduce [9], Intel's TBB [16], OpenMP [6], StarPU [4], StarSs [15] and OmpSs [10], all are examples that are aiming at decoupling the programmer from the underling multicore architecture. They differ by the degree of abstraction they provide to the programmer. StarSs [15] and OmpSs [10] are good examples of a high-level parallel programming model, which require the programmer to annotate sections of code that can potentially run in parallel (tasks) with the conditions under which execution is allowed (dependencies). The runtime system then takes care of maintaining dependencies between tasks, and scheduling of ready ones. This comes at a high cost of runtime overhead that limits the scalability to large number of cores [17].

Meenderinck et al. [14] proposed Nexus (the godfather of our design), a hardware accelerator that was limited to the Cell BE processor [5]. It also has some limitation on the number of parameters a task can have, in addition on the number of tasks that can depend on a certain memory segment. Etsion et al. [12] also proposed a hardware task management unit for the StarSs RTS, based on the similarity between task dependency checking and the instruction scheduler of an out-of-order processor. Although a VHDL prototype was presented for it in [20], it was only evaluated using high-level simulations. The hardware implementation, compared to ours, is relatively expensive.

III. Nexus++ Hardware Task Manager

The Nexus++ [8] task manager is a hardware accelerator for runtime systems of task-based programming models such as OmpSs. Task graph management responsibilities that are usually handled by the RTS, are off-loaded to Nexus++. It tracks tasks' input/output information and utilizes simple table lookups to dynamically build the task graph and find out ready tasks to run.

In [7], the VHDL prototype of Nexus++ is presented, which thoroughly describes the design and implementation as well as a trace-driven evaluation testbench. In [11], the integration process is highlighted with the multicore RTS, and the evaluation of Nexus++ with real applications.

A. Nexus++ Processing Pipeline

Nexus++ processes the incoming tasks in a pipelined fashion. It has a simple 3-stage pipeline shown in Figure 1.

The pipeline shown in Figure 1 is an example for processing tasks that have 4 parameters each. The first stage is the *Input Parser* stage, and it handles receiving the new tasks from the host multicore machine. It makes sure that all the parameters of the new task have been received before forwarding it to the next stage. Data communication between the different stages are done using *FIFOs* lists. These lists have status flags, such as *fifo_empty*, *fifo_full*, and *data_valid* flags, which serve as the synchronization signals between the different pipeline stages. The first stage needs two cycles to receive every memory address in the task's input/output list, plus 4 cycles for the header word and synchronization, giving 12 cycles per task. Once the *data_valid* flag of the first *FIFO* list gets activated, the second stage, namely *Insert*, gets triggered. This is the

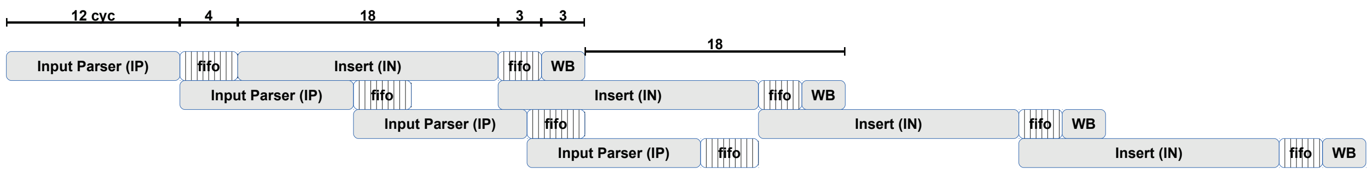

Fig. 1. Nexus++ Pipeline

longest stage in the pipeline, which, as the name indicates, handles the insertion of the new tasks into the task graph, as was thoroughly elaborated in [7]. This stage needs 18 cycles for our 4-parameter task example.

The result of the *Insert* stage decides whether the third stage will be activated or not. The third stage is the *Write Back* stage, and is responsible for sending ready task IDs back to the *Nexus IO* unit, in order to be read by the host multicore machine subsequently. This means that if the inserted task had dependencies on older tasks, it must wait until its dependencies are fulfilled, and therefore cannot be reported by the *Write Back* stage as a ready task. If the inserted task on the other hand had no dependencies, the *Write Back* stage needs 3 cycles to write it back to the *Nexus IO* unit.

The pipeline shown in Figure 1 is, as mentioned before, an example of inserting tasks that have 4 parameters each. In real cases, tasks might have varying number of parameters. Also stalls might happen in one or more stages, for example when the task graph has no more room.

There exist also a second pipeline responsible for handling finished tasks. Handling finished tasks includes kicking off any waiting tasks, and cleaning up Nexus++ tables by deleting the related information of the finished tasks.

Although Nexus++ has one central task manager, it has demonstrated significant improvement over the software RTS using trace-based simulations [11]. Nevertheless, Nexus++ could not improve the scalability of the H264dec benchmark over the software version, since it doesn't support the barrier pragma *taskwait on*. Having lots of buffering in its pipeline, in addition to having relatively long as well as variant pipeline stages leaves some room for further improvement and optimization, as well be discussed in the following section.

IV. Nexus#: Distributed Task Graphs

The first thing to notice in the processing pipeline of Nexus++, Figure 1, is that the unit being processed is a whole task; i.e., the *Insert* stage does not start before having all task parameters being buffered. One idea is to parallelize the *Insert* stage in the pipeline, by replicating the task graphs, and distributing the different memory addresses in a task's input/output list among them.

Looking at the original pipeline again, and having Amdahl's law [2] in mind, we notice that parallelizing only the *Insert* stage will yield a maximum of 2× speedup in ideal cases, since the first and third stages in the pipeline are still serial.

Furthermore, the 2× speedup is an ideal situation, since practically parallelizing the insertion process of the different parameters of a task, implies that a scatter-gather process should take place, since those parameters belongs to the same task, and a final decision must be made whether this task is ready or not. Which introduces an additional overhead that adds to the serial part in Amdahl's equation.

Moreover, the pipeline under consideration is one example of tasks that have 4 parameters each. In real applications, tasks might have varying number of parameters. In fact, we are using a set of benchmarks that have in most cases up to 3 parameters, and in only one case (h264 decoding) 2 to 6 parameters [11], as can be later seen in the experimental setup section. This implies that the maximum task graphs that can be practically used equals the maximum number of parameters a task can have. Which is a scalability hard upper limit. Furthermore, whenever a task with only one parameter is to be inserted, only one task graph will be busy, and the others will be idle.

In order to overcome the above limitations, two design decisions have been made. First: The first stage of the pipeline must be broken down into smaller steps. Secondly: not only parallelizing the insertion process of parameters of a single task, but also those from different tasks. The latter decision is to ensure that applications with very few number of parameters per task can also utilize the different task graphs simultaneously, and thus removing the upper limit on the number of task graphs that can be used, at least for this obvious scenario.

A. Nexus# Design Overview

The block diagram of the proposed distributed task graph system, named Nexus# is depicted in Figure 2. Looking at Figure 2 from top-to-bottom, tasks are still being submitted to the *Nexus IO* unit. It has the same interface as in Nexus++, which is necessary to comply with RTS-Nexus++ communication protocol used in [11].

Since the idea behind Nexus# is to parallelize the insertion process of tasks' parameters using a distributed task graphs, and since the different parameters of a single task might go to different task graphs, a scatter-gather approach must be implemented.

One might think at first that whole tasks should be distributed instead to avoid the *gathering* step, but this way dependencies between tasks cannot be tracked, which contradicts the main functional requirement of our hardware co-processor.

B. Input Parsing

Since the *Input Parsing* stage in Nexus++ pipeline (Figure 1) waits for a whole task to arrive before forwarding it to the next stage, this stage is relatively long, and as was described before will be a scalability bottleneck according to Amdahl's law if left as is. Therefore, the *Input Parser* in Nexus# reads new tasks' parameters from the *Nexus IO*, and distributes them immediately among the different task graphs shown in Figure 2. This way, the insertion process of the first parameter of a task can start, even before the second or rest of parameters of the same task have arrived. Furthermore, parameters of different tasks can be inserted in parallel, as long as they do not share the same task graph.

A key to enhanced utilization and scalability of Nexus# is the distribution algorithm. It should have two essential properties; speed and fairness. Speed, since a slow algorithm will bring us back to the long-delay pipeline stage, and fairness, since having many task graphs that have nothing knocking

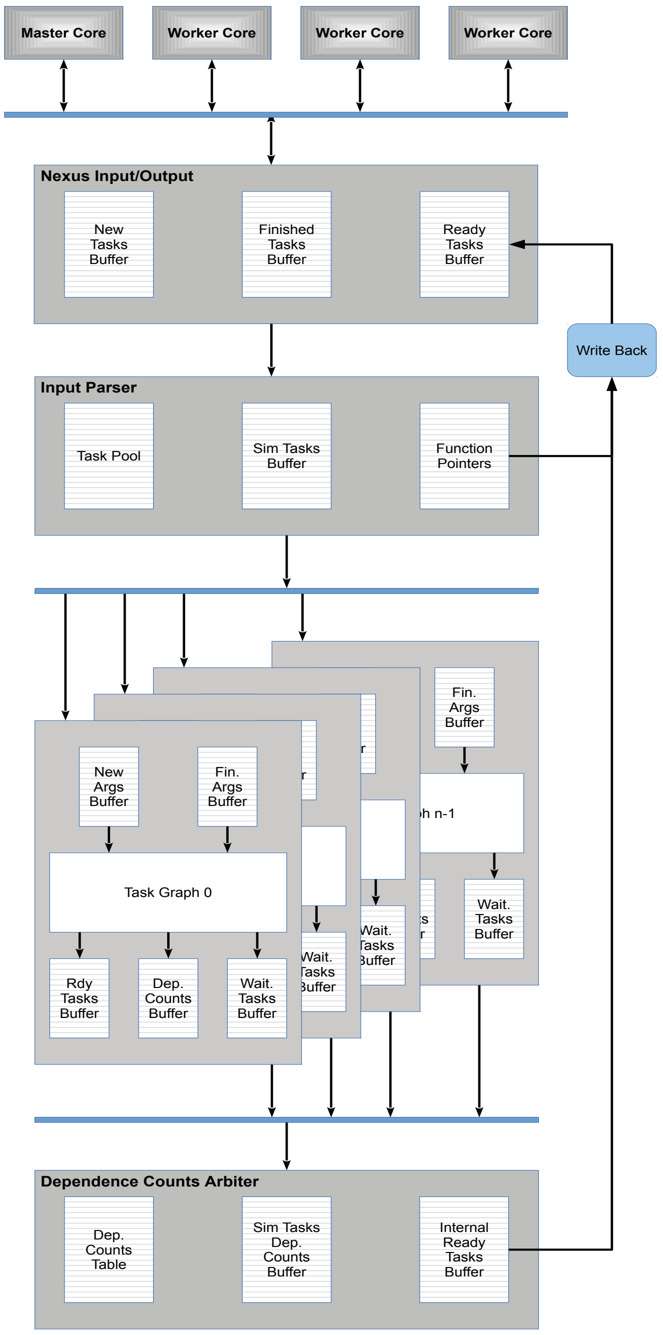

Fig. 2. Nexus# block diagram

their doors makes them useless. The fairness property must also be time-wise. This can be explained by describing the best and worst case scenarios. Let's say that we have n task graphs ($TG_0 \cdots TG_{n-1}$), and m items to be distributed. The best case is when having a round-robin-like distribution. In this case, TG_0 does not get a second item at its input buffer before all other $n-1$ task graphs have also received some inputs. This is to ensure that the different task graphs are busy, while the distribution process goes on. The worst case on the other hand is when the distribution algorithm gives the first m/n items to the first task graphs, the second m/n items to the second task graphs and so on. This scenario implies that the task graphs are working in a serial fashion as shown in

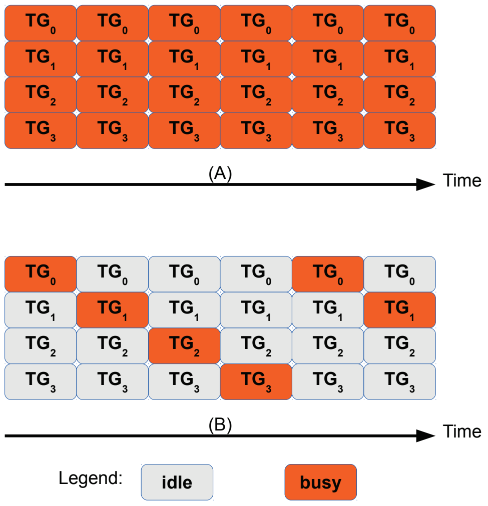

Fig. 3. (A) Best vs. (B) worst case scenarios of exploiting 4 task graphs

Figure 3(B), exactly one after the other, which is equivalent to having one active task graph at a time, in addition to the extra overhead of running the distribution algorithm. Notice that both scenarios have distributed exactly m/n items per task graph eventually.

Given that the data to be distributed are 48-bit memory addresses, and the number of task graphs to choose from is relatively small (5 bits are needed to address 32 task graphs), this problem sounds similar to error detecting codes problems. In our case we should compute the target task graph index as fast as possible (in 1 cycle if possible), therefore, multiple-rounds algorithms, or those which use complex operations like division should be avoided. Furthermore, since our input data are memory addresses, we noticed that for a certain application, the memory addresses it touches differ only in the lower 20 bits. For these reasons, we empirically used the following algorithm to compute the target task graph index:

$$\begin{aligned} TaskGraphID = {} & [addr(19..15) \oplus addr(14..10) \\ & \oplus \ \ addr(09..05) \oplus addr(04..0)] \\ & \text{mod} \ \ num\_task\_graphs; \end{aligned}$$

Our algorithm is based on simple XOR operations of the lowest 20 bits of the input address, divided into 5-bit blocks. It can be computed in one cycle, and has shown experimentally good distribution of the input data among the task graphs, regardless of the used number of task graphs, up to 32 though.

After having distributed all the memory addresses in the new task's input/output list, the *Input Parser* stores the new task in the *Task Pool*. This is important at the end of a task's life cycle; i.e., after running it. At this point, the RTS should report the task as a finished task, and the *Input Parser* will read its input/output list from the *Task Pool*, and distribute them subsequently using the same algorithm, in order to update the task graphs subsequently.

C. Data Insertion into Task Graphs

The insertion process starts at each task graph whenever it receives data from the *Input Parser*. Additional buffers must be added before each task graph, namely the *New Args. Buffers*, in order to decouple the *Input Parsing* and *Insertion* processes. The same principle is applied in case that the incoming task is a finished task, in which case the *Input Parser* will read its input/output list from the *Task Pool*, and distribute them subsequently among the *Finished Args. Buffers* shown in Figure 2.

Each one of the task graphs shown in Figure 2 is the same as the one used in Nexus++ [7]. It uses the same set-associative data structure to maintain a *Kick-Off List* for each incoming memory address.

When processing the data in the *New Args. Buffers*, one of two scenarios might occur. First one is when the task has only one parameter that is to be inserted for the first time in the task graph. This means that the processed task has no other parameters at other task graphs, and therefore can be immediately reported as a ready task. This helps to shrink the size of the last *gather* step in our *scatter-gather* approach. This kind of ready tasks are written at the *Rdy Tasks Buffer* shown below every task graph in Figure 2.

The second scenario, is when the new task found dependent, or when it has other parameters to be inserted at other task graphs. In this case, the result is written in the *Dep. Counts Buffer* shown below every task graph in Figure 2, indicating the task's id, and its dependence count: how many *Kick-Off Lists* has it been added to in that task graph only.

The *gather* step then takes place by the *Dependence Counts Arbiter* whenever any of the *Rdy Tasks Buffer* or the *Dep. Counts Buffer* gets written. If any task was reported as ready, the *gather* step in this case will be an arbitration of writing them to the *Internal Ready Tasks Buffer*, in order to be forwarded to the *Nexus IO* unit.

When gathering the results from the *Dep. Counts Buffers* on the other hand, the *gather* step is relatively longer. It should collect the results from the different task graphs, and conclude the final dependence count of each incoming task.

Having a distributed approach, some parameters of a certain task might get processed sooner than others, by other task graphs because of many factors. For example if one task graph stalled and the other not, or if they stalled for different periods of time. It can also be because one task graph got more data to process than the other. In such cases, while waiting for all the parameters of a certain task to be processed by the different task graphs, the temporal dependence count of this task is stored at *Sim(-ultaneous) Tasks Dep. Counts Buffer* shown in Figure 2.

Having all task's parameters processed and was found ready, it will be written on the *Internal Ready Tasks Buffer*. Otherwise, its dependence count will be stored in the global *Dep. Counts Table* shown in Figure 2.

Finally, when processing finished tasks, if there were some tasks waiting in the *Kick-Off List* of a finished task, those waiting tasks will be written in the *Wait. Tasks Buffer* shown below every task graph in Figure 2. The *Dependence Counts Arbiter* after that decrements the dependence counts of those waiting tasks one by one, and decides accordingly whether they are ready to run, or not yet.

In the next section we will describe Nexus#'s pipeline, and how it improves over its predecessor.

D. Nexus# Processing Pipeline

To demonstrate how Nexus#'s pipeline improves over that of Nexus++, we show in Figure 4 the pipeline of inserting tasks that have 4 parameters each, which is the same one used to explain Nexus++ pipeline.

The pipeline of Nexus# has four stages. *I*nput *P*arsing, data *IN*sertion to the task graph, dependence counts *AR*biteration

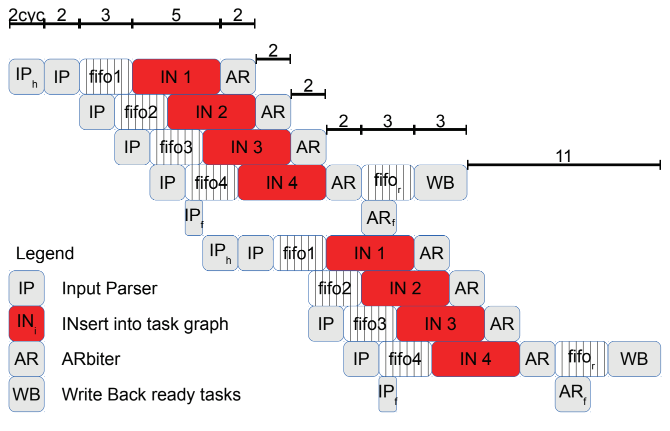

Fig. 4. Nexus# average-case pipeline

as was described in Section IV-C, and finally, the *W*rite ready tasks *B*ack to the *Nexus IO* unit.

The input parsing stage consumes two cycles to receive the header word of the new task (its function pointer and number of parameters), and another two cycles for each parameter (IP_h and IP stages respectively in Figure 4). The communication scheme is based on the PCIe bus [7], therefore, one parameter (48-bit memory address) takes two 32-bit PCIe packets, and thus two cycles.

The *Input Parser* directly distributes every incoming parameter to one of the different task graphs, and in our example, after having distributed the four parameters of the new task, this task's descriptor gets written to the *Task Pool* in one cycle (IP_f stage in Figure 4).

$fifo_{1-4}$ in the pipeline are the *New Args. Buffers* described in Section IV-C. The date written to them needs 3 cycles to appear at their output, which will trigger the next stage in the pipeline: Data Insertion into the task graph. This stage takes 5 cycles per parameter, and once done, the *Dependence Counts Arbiter* collects its result(AR stage in Figure 4).

Once have collected the results of all the 4 parameters of the inserted task in our example, the arbiter checks the readiness of the task. If the task was found ready, its id will be written to the *Internal Ready Tasks Buffer* ($fifo_r$ in Figure 4). The latter fifo takes also 3 cycles to get its data readable at its output port, where the last stage of the pipeline (*Write Back*) takes place. This stage consumes 3 cycles, and simply reads the actual function pointer of the ready task from the *Function Pointers* table shown in Figure 2, and forwards it to the *Nexus IO* unit, to be read later by the RTS.

The difference between the two pipelines can be obviously seen. The Insertion stage in the new pipeline consumed 11 cycles, compared to 18 cycles in the old pipeline. Furthermore, it did not wait for the all the task's parameters to arrive in order to start inserting the first one. Most interesting, the write back stage, where ready tasks gets forwarded to the *Nexus IO* unit, took place every other 18 cycles in the old pipeline for our example, where as this number decreased significantly to 11 cycles in the new pipeline.

Although the pipeline shown in Figure 4 is an example for inserting tasks of 4 parameters each, in real runs, the pipeline might look different. The pipeline shown in Figure 4 assumes that the input buffers of the task graphs are empty, hence the non-perfect parallel insertion of the different parameters. If on the other hand the 4 parameters of our example task were

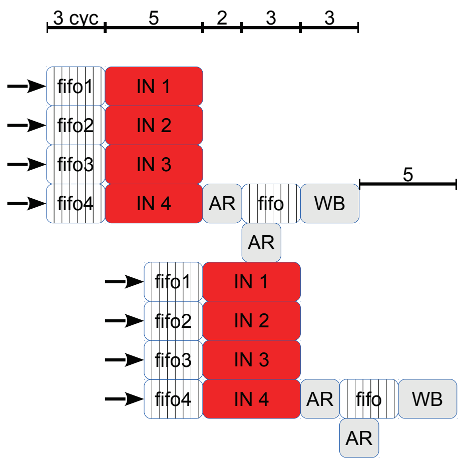

Fig. 5. Nexus# best-case pipeline

already in the buffers, the pipeline in this best-case scenario will behave as can be seen in Figure 5. In this scenario, the *Write Back* stage will take place every other 5 cycles. It is worth mentioning that in this case, the arbiter consumes only two cycles, to collect the results of all the task graphs, and conclude the final dependence count of the corresponding tasks.

There are also scenarios where the insertion stage takes longer periods of time, if for example the task graph stalled due to not fining an empty slot for in a certain line in the set-associative structure. The task graph must then wait until one task finishes, which its parameters share the same line. The good thing about such a scenario is that this gives time to the *Input Parsing* stage to fill up the input buffer of the stalled task graph, increasing the chance that all task graphs work in parallel as in the best-case scenario shown in Figure 5.

The *Dependence Count Arbiter* handles a relatively large amount of computation, which might eventually make it a bottleneck. To avoid this, we designed it in a way to iterate between the three buffers at the end of each task graph in a prioritized fashion. The highest priority goes to reading the *Ready Tasks Buffer*, since they are ready tasks and only need to be forwarded to the next pipeline stage. Second priority is for reading the *Waiting Tasks Buffers*, since they have potential ready tasks. While serving one of the previous two scenarios, this gives time for the different task graphs to finish what they do, namely inserting new items to the task graph. Therefore, this increases the chance of reaching the best-case scenario pipeline shown in Figure 5. To accomplish this, the lowest priority in the *Dependence Count Arbiter* is for reading the *Dep. Counts Buffers*.

In the whole design process, we made sure that Nexus# is deadlock-free, by well-dividing it into different blocks, with fifo lists used as the communication medium to ensure decoupling, and testing it thoroughly.

E. Nexus# Synthesis

Nexus# synthesizablity was tested targeting the Xilinx ZYNQ-7 ZC706 FPGA board. It has a relatively larger FPGA compared to the Virtex-5 board used in the evaluation of Nexus++ [7]. In Nexus#, we wanted to evaluate the distributed task graphs paradigm, which did not fit on the Virtex-5 FPGA board, and thus the switch to the ZYNQ-7 ZC706 FPGA board. Table I shows an overview of the target FPGA utilization, using different design configurations.

Configuration	Registers	LUTs	Block RAMs	Max.(Test) Freq.(MHz)	Total Util.
ZC 706 (Totals)	437200	218600	545		
Nexus++	1%	7%	14%	114.44 (100.00)	7%
Nexus# 1 TG	1%	8%	13%	112.63 (100.00)	7%
Nexus# 2 TGs	2%	15%	25%	112.63 (100.00)	15%
Nexus# 4 TGs	3%	29%	47%	85.26 (83.33)	29%
Nexus# 6 TGs	4%	44%	69%	55.66 (55.56)	44%
Nexus# 8 TGs	4%	58%	91%	43.53 (41.66)	58%

TABLE I
DEVICE UTILIZATION USING DIFFERENT DESIGN CONFIGURATIONS ON THE ZC706 FPGA BOARD

Our baseline is the Nexus++ [7] design. Although it was evaluated using a different FPGA board, we have re-synthesized it again using the ZC706 FPGA board to make comparable with the other configurations.

The three main criteria shown in the table are the registers, look-up tables(LUTs), and block RAMs. The latter reflects the data structures used in the design, mainly the tables in the task graphs for example, while the first two reflect the computational part of the design, i.e., the state machines.

Having only one task graph in the configuration of Nexus# is most analogous to Neuxs++. This can be seen in Table I, as both have very close utilization values.

By increasing the number of task graphs in Nexus#, one can notice how this is reflected in Table I: the number of block RAMs almost doubles due to using multiple task graphs, and the number LUTs also doubles because of the extra work the *Input Parser* and the *Dependence Counts Arbiter* blocks have to manage every time the number of task graphs doubles.

Hardware-wise comparison with [20], [19] shows that their design consumes 29,138 registers and 110,729 LUTs respectively, which is comparable to the resources needed by our 8 task graphs design (19,350/127,290 registers/LUTs respectively), and 6× more than the resources needed by the 1 task graph configuration. Moreover, using a micro benchmark built after [19] that includes inserting 5 independent tasks, each with two parameters, Nexus# (with one task graph) consumes 78 cycles compared to 172 cycles consumed in [19]. Their design runs at a higher frequency though (150 MHz).

V. Performance Evaluation

To evaluate the performance of Nexus#, we ran several trace-based simulations using various benchmarks. The purpose of our experiments is to measure the enhancements of the scalability that can be obtained by the new distributed task graph, and compare it to the related architectures.

A. Benchmarks

We use four benchmarks from the Starbench benchmark suite [3]: *c-ray* (ray tracing), *h264dec* (H.264 video decoding), *rot-cc* (image rotation and color conversion) and *streamcluster* (k-median clustering). To these benchmarks, we add *sparselu* (sparse LU matrix factorization benchmark used by the OmpSs developers).

c-ray and *rot-cc* have simple dependency patterns, with tasks working on each line of the input image independently. For *c-ray*, there is only one task per line, which means that all tasks are independent. For *rot-cc* there are two tasks per line, one for rotation and one for color conversion, with the second depending on the first. All pairs are independent from each

	# tasks	total work (ms)	avg task size (μs)	# deps
c-ray	1200	7381	6151	1
rot-cc	16262	8150	501	1
sparselu	54814	38128	696	1-3
streamcluster	652776	237908	364	1-3
h264dec-1x1-10f	139961	640	4.6	2-6
h264dec-2x2-10f	35921	550	15.3	2-6
h264dec-4x4-10f	9333	519	55.6	2-6
h264dec-8x8-10f	2686	510	189.9	2-6

TABLE II
BENCHMARKS' DURATIONS OBTAINED FROM TRACES COLLECTED ON A XEON E7-4870 MACHINE

Matrix dimension	# Tasks	Average task weight	
		FLOPs	μs
250	31,374	167	0.084
500	125,249	334	0.167
1000	500,499	667	0.334
3000	4,501,499	2012	1.006

TABLE III
GAUSSIAN ELIMINATION TASKS FOR DIFFERENT MATRIX SIZES

other. *c-ray* is a best case for this type of runtime, as it has long tasks and ample parallelism, thus most runtime overhead can overlap with task execution.

streamcluster is a streaming data analysis kernel with fork-join-style parallelism. It consists of a chain of groups of about 400 tasks followed by a `taskwait`.

sparselu and *h264dec* have more complex dependency patterns. *sparselu* is a sparse matrix LU factorization kernel from the developers of OmpSs. It scales well, as the granularity is designed to match Nanos overheads. The H.264 decoder, on the other hand, can be configured to run with variable granularity by setting the number of macroblocks that are processed by one task. At the extreme, a new task is created for each macroblock. This fine-grain parallelism is especially challenging to manage. We test 4 variation of the *h264dec* benchmark varying the number of macroblocks that are mapped to one task. h264dec-1x1-10f indicates that only 1 macroblock is mapped to one task, h264dec-2x2-10f mappes 4 macroblocks to one task, and so on. All variation has 10 full HD frames (hence the 10f) of a video stream (pedestrian_area.h264) as input.

To validate the dummy tasks/entries approach, the task graph of Gaussian elimination with partial pivoting [18] is used. In this benchmark, the number of tasks that depend on a certain memory segment depends on the size of the input matrix as depicted in the dependency pattern of Figure 6, assuming an $n \times n$ matrix. Table III gives an overview about number and granularity of Gaussian tasks for different matrix sizes.

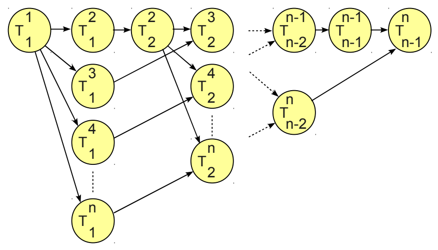

Fig. 6. Dependency pattern for the Gaussian elimination benchmark. T_i^j: i, j row and column numbers respectively

B. Experimental Setup

From the execution of each benchmark on a 40-core Xeon E7-4870 machine running at 2.40GHz, we collected traces that include the task descriptors (which specify the inter-task dependencies) and the execution time of each task. The test bench simulates the RTS. It submits new tasks to Neuxs#, receives ready task information from it, schedules ready tasks to worker cores and simulates their execution, and finally notifies Nexus# of finished tasks. Using the information from the traces, we performed two sets of simulations:

No Overhead: This simulates the execution of an application without any overhead, to determine the lower bound for the execution time of the benchmarks. In this simulation, the simulation time does not advance while dependencies are resolved. Only the execution time of the tasks is taken into account. This allows us to determine when the lack of available parallelism in the application is the limiting factor.

Nexus# only: In this simulation, we additionally account for the dependency resolution overhead incurred by the Nexus# core. If an application scales much worse in this simulation than in *No Overhead*, it indicates a bottleneck inside the design. In this simulation free worker cores start executing tasks directly after they are reported as ready by Nexus#. No communication or other non-dependency resolution overhead is accounted for.

These simulations are compared to the actual runs of the benchmarks on the same machine that the traces were collected on, compiled using the Mercurium compiler version 1.3.5.8 and linked to the accompanying Nanos runtime library. We also compare them to the results obtained when using Nexus++ as the task manager.

VI. EVALUATION RESULTS

The first test we carried out to evaluate Nexus# was the scalability test: we simulated the H264dec benchmarks with changing the number of task graphs (TGs) used in Nexus#, in order to get the optimal configuration. We chose the H264dec benchmark because we can group several macroblocks to be decoded by one task, and hence varying the task size. The finer the tasks are, the more challenging it is for any task graph manager, since the worker cores will finish executing their fine tasks quickly and demand the scheduler for new ones more often. This way, we can see the impact of task size on the performance Nexus#. Grouping several macroblocks together is not a trivial task, and requires the programmers to explicitly specify which macroblocks can be grouped together in order to preserve dependencies. The goal of Nexus# is to alleviate the programmer from doing this, by being able to manage the finest tasks without the need to apply the grouping technique (1 macroblock per task).

The results of the different scalability tests are depicted in Figure 7. The upper graphs (a) in Figure 7 shows the results of running Nexus# at 100MHz, regardless the number of task graphs used. This is to give a fair scalability test and relating it to only the number of task graphs used. Figure 7(b) on the other hand, depicts the results of running Nexus# at variable frequencies depending on the number of task graphs, as per Table I. This gives the relaistic performance of Nexus# using a certain number of task graphs, and helps to determine the optimal configuration of Nexus#.

No grouping of macroblocks exist in the left-most graphs in Figure 7, meaning that this experiment has the finest tasks.

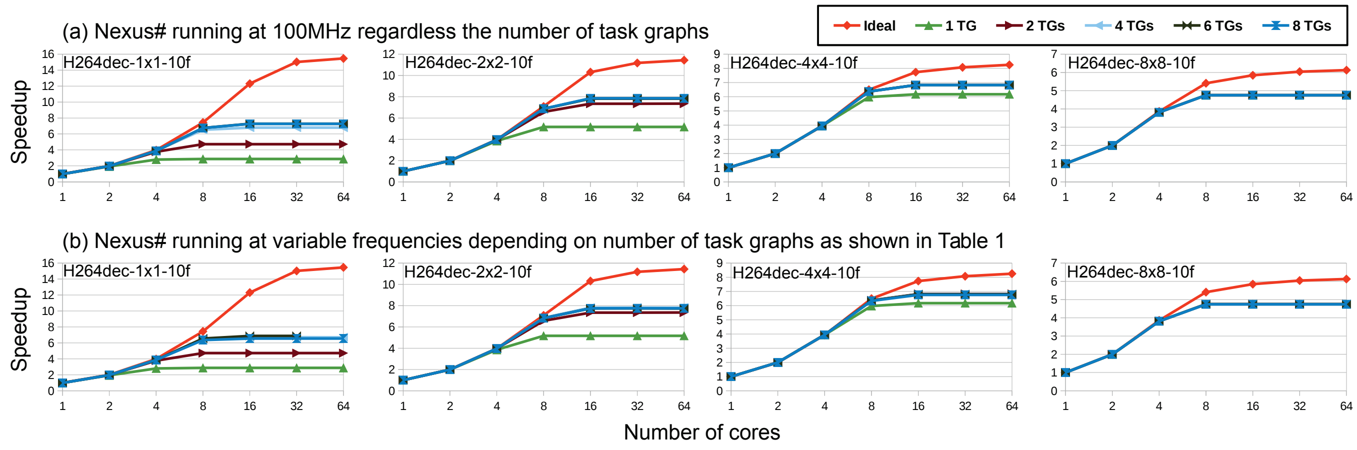

Fig. 7. Scalability of Nexus# running different configurations of the H264dec benchmark

The other graphs include grouping of macroblocks: from left-to-right order, 2×2, 4×4, and 8×8 macroblocks per task respectively. Looking back at Table II, one can see the effect of grouping on task size, which went from 4.6 μs on average in case of no grouping, to about 190 μs in case of grouping 8×8 macroblocks per task. Having the same input, the total number of tasks also changed drastically between the different configurations.

The red line (most-upper in all graphs) in Figure 7 is the ideal scalability curve, where only the execution times of tasks have been simulated, without adding the time needed to resolve dependencies as was described in Section V-B:*No Overhead*.

It can be seen that the larger the task size is, the easier it becomes to Nexus# to handle them, since it could get closer to the ideal scalability curve, even using small number of task graphs. Most interestingly is the hardest experiment shown in the left-most graphs in Figure 7, where one can see that Nexus# scales up to 7×, using 6 task graphs. The differences between using 4, 6, and 8 task graphs are very minimal, but we chose to use 6 task graphs for our later evaluation, since this configuration achieves the best scalability results.

Changing the number of task graphs in Nexus# impacts the maximum operating frequency as was shown in Table I, since the design has more structures as the number of task graphs grows. This imposes more work on the *Dependence Counts Arbiter*; the unit responsible for gathering results from the different task graphs.

The results of the two experiments shown in Figure 7(a), with Nexus# running at 100MHz and (b), with it running at the test frequency shown in Table I, they both confirm our observation that using 6 task graphs achieves the best scalability results. Although the operating frequency has been reduced significantly in experiment (b) of Figure 7 using 6 and 8 task graphs to 55.56 MHz and 41.66 MHz respectively, their performance results were slightly smaller than their higher speed siblings in Figure 7(a). The following experiments were done using 6 task graphs running at 55.56 MHz. First, we evaluate the benchmarks listed in Table II.

Figure 8 shows the performance evaluation of Nexus# using 6 task graphs, and compares it to other task graph managers: 1- our baseline OmpSs runtime system called Nanos, and 2- Nexus++. Furthermore, the ideal scalability curve for each benchmark is also added to the different graphs in order to see the big picture. All speedup results are calculated against the single core execution time of the ideal curve, which very close (although faster) to the sequential version of the benchmark. The results of Nanos are only up to 32 cores, which is limited by the hard number of cores our test machine has. In Figure 8(a), the *c-ray* benchmark represent an easy case for all the task managers; it has relatively large tasks (6 msec on average), and has only independent tasks. All task graph managers performed well and were close to the ideal scalability curve and scored 31.5× speedup on 32 cores. Nexus# continued its excellent performance and scored 194× on 256 cores, compared to 60.4× achieved by Nexus++.

The *rot-cc* benchmark has smaller tasks, with pairwise independent tasks, which is a harder case for the task graph managers than *c-ray*, but still relatively easy. Both the hardware task managers (Nexus++ and Nexus#) scored 32× speedup on 32 cores, which is better than the software task manager (Nanos) which scored only 24×. Nexus# and Nexus++ continued scaling to 256 cores, both achieving 254× speedup.

The *sparselu* benchmark has more complex dependencies between its tasks, and again, the hardware task managers performed better than Nanos (31× vs. 24.5× speedup on 32 cores respectively). Nexus# achieved up to 94.4× speedup on 256 cores, which is slightly better than the 84.9× speedup achieved by Nexus++.

The *streamcluster* benchmark is a more difficult one for the task graphs. Nanos achieved only 4.9× on 32 cores, whereas Nexus++ and Nexus# achieved 7.9× and 30.1× respectively. Nexus# continued scaling achieving about 39× speedup on >=64 cores.

The performance of the *H264dec* benchmarks are depicted in Figure 8(b). Our main focus is on the left-hand side graph, which does not requires the programmer to do any grouping of macroblocks per task. In this graph, we can see that Nanos performs pretty bad and cannot achieve any speedup. Nexus# on the other hand achieved up to 6.9× on >= 16 cores. Nexus++ does not support the "task-wait-on" OmpSs pragma [11], and achieved only 2.2× speedup on >=4 cores.

Grouping several macroblocks per task increases the average task size, and makes the management of the task graph much easier. Nanos in particular achieves its best performance when 8×8 macroblocks are grouped in one task, and scored 3.9× on 8 cores. Its performance drops down when using larger number of cores though. Nexus# achieved slightly better speedup, and sustained its performance for larger number of cores.

From all the graphs in Figure 8, it can be clearly seen that

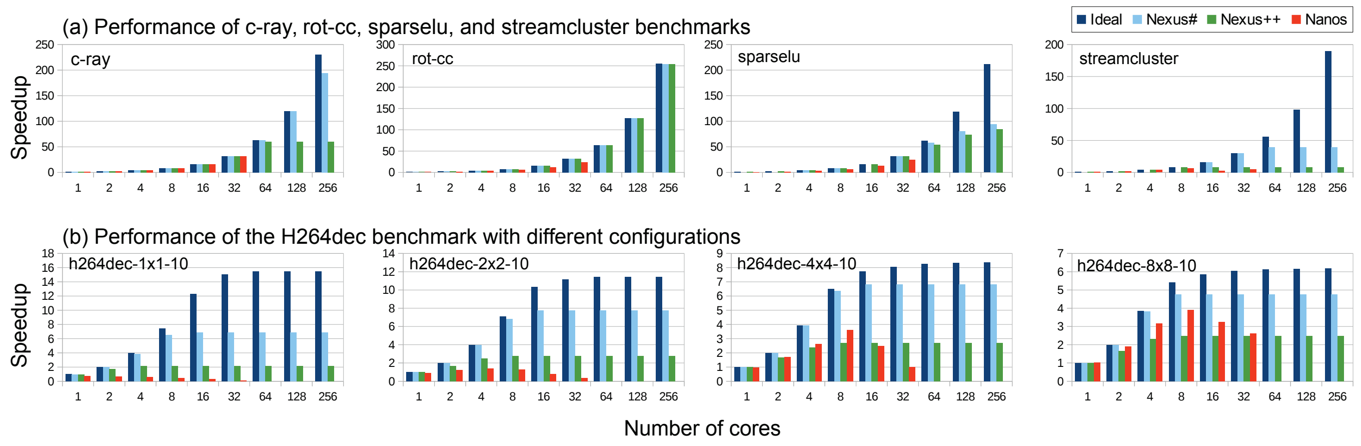
Fig. 8. Performance of Nexus# running different benchmarks, in comparison to other task managers

Benchmark	Nanos Max.	Nexus++ Max.	Nexus# Max.
c-ray	31.4×	60.4×	194.0×
rot-cc	24.5×	254×	254×
sparselu	24.5×	84.9×	94.4×
streamcluster	4.9×	7.9×	39.6×
h264dec-1x1-10f	0.7×	2.2×	6.9×
h264dec-2x2-10f	1.4×	2.7×	7.7×
h264dec-4x4-10f	3.6×	2.7×	6.8×
h264dec-8x8-10f	3.9×	2.5×	4.7×

TABLE IV
MAXIMUM SCALABILITY USING THE DIFFERENT TASK GRAPH MANAGERS

Nexus# has the upper hand over the other two task graph managers. Most interestingly are the cases where tasks are very fine grained. We think that such tasks are the key to utilizing the ever-increasing computing power being embedded on the state-of-the-art and future SoCs. In such cases, the strength of Nexus# is most clear and needed. Table IV summarizes the maximum achievable speedup for the different benchmarks using Nanos, Nexus++, and Nexus#. Since we based our evaluation results on the single-core ideal simulations, it is also worth mentioning that the sequential version for most of the benchmarks has a very close (slower) execution time as our baseline. The only exception is the H264dec benchmark, where the sequential execution is almost twice as fast as the single-core ideal simulation. This shows the potential overhead of porting an application to the task execution model. Hence, the maximum (real) speedup achieved in the case of H264dec benchmarks is about 3×.

Figure 9 shows the speedup achieved by running the Gaussian elimination problem (Figure 6) on different multicore systems for different matrices of sizes ranging from 250×250 to 3000×3000. The Gaussian elimination benchmark is a micro benchmark that is not trace-based as the previous benchmarks. This benchmark in particular is a worst-case scenario for Nexus# as the example described in Figure 3(B).

Running the application on a 250×250 matrix for example, starts by having one ready task (T_1), and 249 dependent tasks. Those are direct dependencies, meaning that all the 249 tasks have the same memory address as input, that is to be produced by the first ready task (T_1). This indicates that regardless the number of task graphs used in Nexus#, only one will be used to insert the first 250 tasks, and another one for the next wave of tasks, and so on. For this reason, increasing the number of task graphs used for this benchmark will have a negative impact on performance, mainly because the clock frequency deriving Nexus# decreases as the number of task graphs increases. Furthermore, the tasks generated in this benchmark has only up to 2 parameters per task. As a result, we chose to evaluate this benchmark using only 2 task graphs.

Figure 9 shows the results of the Gaussian elimination benchmark using Nexus++, Nexus# with one task graph (1TG), Nexus# with two task graphs (2TG) respectively, all running at 100 MHz. The baseline here is the single-core execution time using Nexus++. Each worker core is assumed to be able to do 2 GFLOPS, which means that the average computation time (in μs) of the Gaussian tasks = 2000/#FLOPs, as can be seen in Table III above. It can be seen that Nexus# (2TG) has a slight improvement over Nexus++. About 19% in case of the very fine grain tasks in matrix-250, and as the matrix size (and hence larger number of tasks of larger granularity) increases, Nexus# has about 10% performance improvement over Nexus++. This benchmark, as was described before, is to show that our hardware task managers don't have a static limit on the number of tasks that can wait for a certain memory address.

According to Vandierendonck et al. [17], the runtime overhead of their proposed software task graph manager can go as low as 400 cycles (0.2 μs on their test machine) per task, their experiment assumed inserting 1-parameter tasks to an empty task graph, which is an ideal case. Therefore, we still think that hardware acceleration is vital.

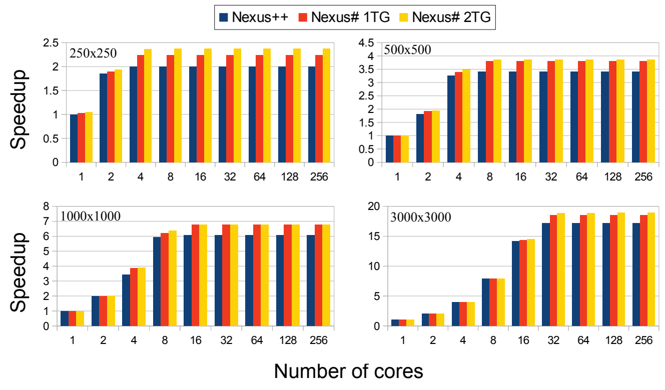
Fig. 9. Performance of Nexus# running Gaussian elimination benchmark for different matrix sizes

Designing a hardware accelerator for the RTS in a multicore system implies that a communication between the two to take place in real time. Our hardware task manager is meant to be integrated in future many-core chips, which in the era of high integrity and homogeneity, we believe that an FPGA will also be integrated on the same chip. This will provide a low-latency communication channel between the computing cores and the user design on the FPGA.

Compared to the state-of-the-art clock frequencies that drive microprocessors found in HPC machines or even consumer products, Nexus# runs at a relatively very low frequency (50 - 100 MHz). Although power-consumption analysis is part of the future work, Nexus# has a potential to manage the task graphs in a wide range of multicore machines, without being the power-drain hot spot. Depending on the use case, the number of needed number of task graphs can be accordingly configured. In cases with limited number of cores and/or limited power, as in smart phones and other consumer devices, it can even be turned off (as dark silicon) if the number of ready tasks exceeds a certain threshold.

Since multiple applications use different memory spaces inherently, Nexus# can manage them at the same time. The door is wide open ahead of Nexus# to be integrated in real multicore/manycore SoCs, as we think that task-based, data flow execution model is a key to utilizing the computational power of such systems. For example, OpenMP 4.0 has adapted the tasking model, and is a suitable target to analyze and see the impact of integrating Nexus# with it. Also important is to analyze the effect of data communication and RTS's non-dependency-related tasks on the scalability figures. Therefore, the next step will be to integrate Nexus# with the ARM-core on the target Xilinx ZYNQ platform to run real benchmarks on CPU+FPGA, in a tightly-coupled fashion.

VII. Conclusions

We presented Nexus#, a VHDL prototype of a hardware task manager for the OmpSs runtime system. Supporting the in, out, inout, taskwait, and taskwait on pragmas, Nexus# is suitable for wide range of applications, including H264 decoding. It employs a distributed task graph management strategy, which enables the parallel insertion of the input/output memory addresses of the incoming tasks. Besides implementing a low-latency task graph look-up mechanism using set-associative cache-like structures, Nexus# also uses a fast distribution function that directs seamlessly the incoming memory addresses to the proper task graph.

Generating data and runtime traces for multiple benchmarks in the Starbench suite, and embedding them in a Modelsim testbench, the experimental results show that Nexus# achieved significant speedups for all the benchmarks on a 256-core pseudo-machine. Results also prove that Nexus# perform better than Nanos, the official OmpSs runtime, as well as Nexus++, our central task graph manager, by orders of magnitude, for benchmarks that have very fine grain tasks and/or complex dependency patterns. We have also shown that a benchmark modeled after Gaussian elimination, where the number of tasks that depend on a certain task is not constant, ran successfully and efficiently with an achieved speedup of $19\times$ for a 3000×3000 matrix using 64 cores.

Nexus# is fully configurable, and depending on the use case, the number of task graphs can be changed. Although targeting OmpSs applications, Nexus#'s low-latency retrieval task graphs can be used with other programming models. We well focus in the future on tightly-coupling Nexus# with real CPUs, as well as on power analysis, and the possibility of dynamically turning (parts of) it on and off (as a dark silicon), in order to integrate it with low-power mobile systems.

References

[1] G. Al-Kadi and A. S. Terechko. A Hardware Task Scheduler for Embedded Video Processing. In *Proc. 4th Int. Conf. on High Performance Embedded Architectures and Compilers*, 2009.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] M. Andersch, C. C. Chi, and B. Juurlink. Programming parallel embedded and consumer applications in openmp superscalar. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 281–282, New York, NY, USA, 2012. ACM.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, Feb. 2011.

[5] B. S. C. BSC. Cell superscalar. http://www.bsc.es/plantillaG.php?cat_id=179, 2009.

[6] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE Computational Sci. Eng.*, 1998.

[7] T. Dallou, A. Elhossini, and B. Juurlink. FPGA-Based Prototype of Nexus++ Task Manager. In *6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) 2013, Co-located with SC 2013*, 2013.

[8] T. Dallou and B. Juurlink. Hardware-Based Task Dependency Resolution for the StarSs Programming Model. In *41st Int. Conference on Parallel Processing Workshops (ICPPW), SRMPDS*, pages 367 –374, 2012.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symp. on Operating Systems Design & Implementation*, 2004.

[10] 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(2):173–193, 2011.

[11] N. Engelhardt, T. Dallou, A. Elhossini, and B. Juurlink. An Integrated Hardware-Software Approach to Task Graph Management. In *16th IEEE International Conference on High Performance and Communications HPCC-2014*, pages 392–399, 2014.

[12] 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. *IEEE/ACM Int. Symposium on Microarchitecture*, 0, 2010.

[13] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. 34th Annual Int. Symp. on Computer Architecture*, 2007.

[14] C. Meenderinck and B. Juurlink. A Case for Hardware Task Management Support for the StarSs Programming Model. In *Proc. 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, 2010. Sp. Session on Multicore Systems: Des. and Apps.

[15] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *International Journal of High Performance Computing Applications*, 2009.

[16] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 1st edition, 2007.

[17] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. Analysis of Dependence Tracking Algorithms for Task Dataflow Execution. *ACM Transactions on Architecture and Code Optimization*, 10(4):61:1–61:24, Dec. 2013.

[18] M. Veldhorst. Gaussian Elimination with Partial Pivoting on an MIMD Computer. *Journal of Parallel and Distributed Computing*, 1989.

[19] F. Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia. Analysis of the task superscalar architecture hardware design. *Procedia Computer Science*, 18(0):339 – 348, 2013. 2013 International Conference on Computational Science.

[20] F. Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia. FPGA-Based Prototype of the Task Superscalar Architecture. In *7th HiPEAC Workshop on Reconfigurable Computing (WRC 2013)*, 2013.