An Integrated Hardware-Software Approach to Task Graph Management

Nina Engelhardt, Tamer Dallou, Ahmed Elhossini, Ben Juurlink
Embedded Systems Architecture
Technische Universität Berlin
Einsteinufer 17, 10587 Berlin, Germany
{nina.engelhardt, tamer.dallou, ahmed.elhossini, ben.juurlink}@aes.tu-berlin.de

***Abstract*—Task-based parallel programming models with explicit data dependencies, such as OmpSs, are gaining popularity, due to the ease of describing parallel algorithms with complex and irregular dependency patterns. These advantages, however, come at a steep cost of runtime overhead incurred by dynamic dependency resolution. Hardware support for task management has been proposed in previous work as a possible solution. We present VSs, a runtime library for the OmpSs programming model that integrates the Nexus++ hardware task manager, and evaluate the performance of the VSs-Nexus++ system. Experimental results show that applications with fine-grain tasks can achieve speedups of up to 3.4×, while applications optimized for current runtimes attain 1.3×. Providing support for hardware task managers in runtime libraries is therefore a viable approach to improve the performance of OmpSs applications.**

***Keywords*—OmpSs, parallel programming models, task dataflow, hardware task scheduler, runtime library**

I. Introduction

The trend in microprocessor design has shifted from increasing the clock frequency towards integrating an ever increasing number of cores on chip. Nowadays, there exist consumer and mobile devices with 8 cores on chip, and this trend is expected to continue in the near future. To obtain speedups from these new architectures, applications need to use parallel algorithms, which are more challenging to develop than their sequential alternatives. Many programming models have been proposed to ease parallel programming, such as Google's MapReduce [1], Intel's TBB [2], OpenMP [3], StarSs [4] and OmpSs [5]. All parallel programming models share the goal of decoupling the programmer from the underlying multicore machine, but they differ from one another in the degree of abstraction. Higher-level abstractions typically lead to higher overheads as the runtime system needs to do more work to bridge the gap between programming model concepts and the machine's specific architecture.

Task-based programming models are a class of abstractions that 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 organizes execution of all tasks respecting the constraints, avoiding the need for the programmer to reason (and potentially make difficult-to-find mistakes) about the circumstances when the dependencies are fulfilled. StarSs [4] and OmpSs [5] are good examples of such a paradigm. The runtime system of those programming models builds, at runtime, a task graph based on the sequence of function calls and their input/output requirements, and determines which tasks are ready to run. This approach reduces parallel programming complexity, since the task of extracting and managing parallelism is entirely offloaded to the runtime system. The overhead of the runtime system, however, is a particular concern [6]. Overhead can be hidden if the runtime system succeeds to keep the cores busy and does its task graph management responsibilities concurrently. This becomes more difficult with larger numbers of worker cores and/or applications with fine-grain tasks, complex dependency patterns, or both. This has been shown in previous work [7] for the case of applications parallelized using StarSs.

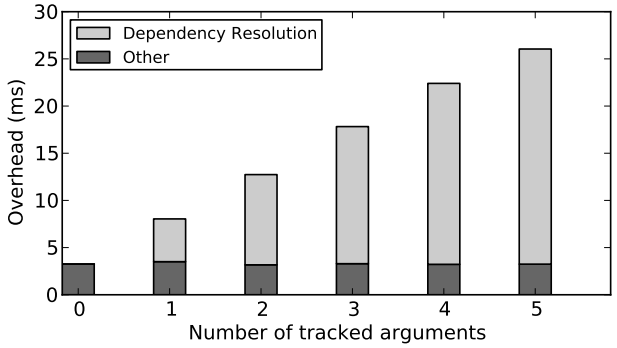

Fig. 1. Dependency resolution and other runtime overhead for tasks with differing numbers of arguments

In the case of OmpSs, the most time-consuming part of the runtime system is dependency resolution. To illustrate, Figure 1 shows the aggregate overhead measured during a run of the *c-ray* benchmark (for a description, see Section V-A) when increasing the number of tracked arguments per task. The proportion dedicated to dependency resolution is highlighted.

Previous work ([7], [8], [9]) introduced hardware support for task graph management, and showed significant improvement in the scalability of different micro benchmarks modeled after H.264 video decoding. In [9], it was proposed to integrate a hardware accelerator, called Nexus++, with the OmpSs runtime system by using a PCIe extension board housing an FPGA. However, as the necessary support from the OmpSs runtime was not provided, Nexus++ was evaluated, similarly to its predecessors in [7], [8], using only synthetic benchmarks.

In this paper, we present VSs, an OmpSs runtime library supporting the Nexus++ hardware manager. We evaluate

Nexus++ and VSs performance on actual applications, both in simulation and in practice. Performance improvements due to Nexus++ reach up to 240% for fine-grain tasks, and 25-30% for coarser granularity. Using an FPGA development board, we developed a prototype PCIe extension card with the Nexus++ hardware design, which can be put into any multicore machine to manage task execution, showing that this approach is functional outside of simulation.

Section II presents some background on the OmpSs programming model as well as related work. In Section III, we describe the Nexus++ hardware manager, and Section IV explains the VSs runtime library and how it is capable of integrating Nexus++. The results of evaluating Nexus++ performance on several benchmarks are presented in Section V. We conclude in Section VI.

II. Background

We present a brief overview of the OmpSs programming model, and of other hardware accelerators in the literature for OmpSs or similar task-based models.

A. OmpSs

OmpSs is a task-based programming model that extends the OpenMP task directive. To execute OmpSs applications, they are compiled with a source-to-source compiler and linked against a runtime library.

1) Programming Model: In OpenMP, annotating a function declaration with `#pragma omp task` means that calls to the function are submitted to be executed asynchronously: the call returns immediately, and the calling context can continue execution. Meanwhile, the function is scheduled to be executed in parallel in a separate context. The calling context can later ascertain completion of all asynchronous function executions using `#pragma taskwait`. It can also wait for completion of only functions operating on a specific data set `p` using `#pragma taskwait on (*p)`.

OmpSs extends the task model to capture data dependencies between tasks. Using the syntax `#pragma omp task in(*a,...) out(*b,...) inout(*c,...)` (with `a`, `b`, `c` pointers to data structures passed as arguments to the function) the programmer can indicate which inputs a task requires, and which outputs it produces. The runtime system then takes care of tracking the dependencies and only launching tasks when all their inputs are ready. This removes work both from the programmer, who no longer has to worry about synchronisation between tasks, and from the main thread of execution, which no longer has to manage the tasks. Work can simply be submitted immediately when it becomes apparent that it should be performed.

Listing 1 shows a simplified example of OmpSs programming, extracted from H.264 macroblock wavefront decoding [10]. The function `decode()` is called inside a nested loop, processing the elements of matrix X. To process an element, the task needs the values of the cells west and northeast of the current cell. (Cells that would fall outside the matrix are assumed to be 0.)

```
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

2) Runtime System Implementation: The official OmpSs implementation consists of two parts: a source-to-source compiler, named Mercurium, that transforms pragmas into function calls to a runtime library, and the runtime library itself, named Nanos [5]. To integrate Nexus++, we keep the Mercurium compiler but replace the Nanos runtime library with our own, VSs, that supports exchanging task status information with the Nexus++ task manager over PCIe.

B. Hardware Task Management

Nexus [7] is a basic hardware task manager for StarSs [4], which was integrated in a simulator of the Cell processor. Nexus++ is based on the basic table lockups of Nexus to manage the task graph, but in an overhauled and more efficient design [8], presented as a SystemC prototype. Nexus++ removes the limitations on the number of inputs and outputs a task can have (up to 5 in [7], [11]). Similarly, the number of tasks that can depend on a certain data segment is limited in Nexus, which limits its applicability. Moreover, Nexus++ adds support for double buffering. Nexus++ has also been implemented in VHDL [9]. It provided an improved search algorithm which implements a cache-like multi-way set-associative task graph. Furthermore, the VHDL Nexus++ prototype has presented an interface for integration with task-based runtime systems, which will be employed and evaluated in this paper.

In addition to Nexus++, other hardware scheduling units have been proposed in literature. Most of them, however, assume independent tasks and are optimized for a certain application, a certain platform, or both. For example, Carbon [12] assumes independent tasks and uses hardware queues to retrieve tasks with low latency. An example of a hardware accelerator that targets a certain application domain is the hardware task scheduler optimized for H.264 decoding [13]. It only supports a specific dependency pattern, and any modification to the task structure of the application would lead to incompatibility. Etsion et al. [14] also proposed a hardware task management unit for the StarSs runtime system, 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 [15], it was only evaluated using high-level simulations.

To the best of our knowledge, this is the first time that a hardware task graph manager has been integrated into a runtime and evaluated on actual applications in practice.

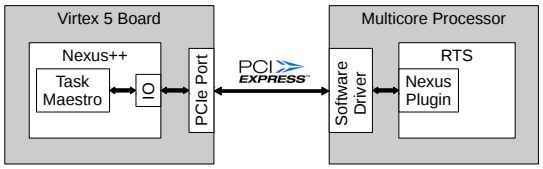

Fig. 2. Nexus++ high level system overview

System Parameter	Value
Nexus++ (w/ PCIe) clock freq.	12.5 MHz
Task Pool size	1024 tasks * 250 bits
No. Parameters per *TD*	8
Task Buffer size	8 tasks * 218 bits
Dependence Table size	8-way * 256 entries * 190 bits
Dummy Kol table size	1024 entries * 96 bits
Kick-Off list size	8 tasks

TABLE I. SYSTEM PARAMETERS

III. NEXUS++

The Nexus++ [9] task manager is a hardware accelerator for dependency resolution, to be used with task-based programming models like OmpSs. Nexus++ tracks tasks input/output information and utilizes simple table lookups to identify ready tasks and schedule them to worker cores. Nexus++ is meant to be integrated with real multicore systems. For this, Nexus++ was implemented in VHDL targeting the Xilinx XUPV5-LX110T FPGA development board. The high level design is depicted in Figure 2. The Nexus++ task manager resides on the FPGA board, and communicates with the runtime system on the multicore system using the PCIe bus.

As shown in Figure 2, data communication occurs between Nexus++ and the runtime system (via the *Nexus Plugin*). When the master thread creates new tasks, the runtime system (RTS) submits them to Nexus++. Nexus++ sends ready task IDs to the runtime system, and whenever a worker core finishes a task, the runtime system notifies Nexus++ of that.

The FPGA board can be plugged into the host multicore machine and the task graph management responsibilities can be offloaded to Nexus++. In [9], the VHDL prototype of Nexus++ is presented, which thoroughly describes the design and implementation as well as a trace-driven evaluation testbench.

A. Functional Overview

Figure 3 shows the block diagram of our task manager. It is mainly composed of two units: the Nexus input/output unit (*Nexus IO*) which handles communication with the host runtime system, and the task management unit known as the *Task Maestro*, which manages the task graph at runtime and issues tasks when they are ready.

The *Nexus IO* unit manages communication between the host runtime system and the *Task Maestro* using three FIFOs: The *New Tasks FIFO* receives the incoming new tasks from the runtime system, and buffers them for the *Task Maestro*. The *Task Maestro* notifies the runtime system of ready tasks by writing them to the *Ready Tasks FIFO*. Whenever a task is finished, the runtime system writes its ID back to the *Finished Tasks FIFO*.

The *Task Maestro* reads the new tasks and stores them in (1) a temporary task storage table called the *Task Buffer*, which is accessed when inserting new tasks to the task graph, and (2) in the *Task Pool*, where they reside until the end of their life cycles, and are accessed when processing finished tasks. If a task has more inputs/outputs than can be stored in one *Task Pool* entry, then the *Task Maestro* allocates other entries in the *Task Pool* for this task. This mechanism of adding dummy tasks [9] ensures that there is no static upper bound on the number of inputs/outputs of the tasks handled by Nexus++.

The *Task Maestro* constructs the task graph and calculates the *Dependence Count* for the task in progress. This is done by comparing every input/output of the new task against all inputs/outputs of all previously submitted tasks. The resulting *Dependence Count*, if larger than 0 (i.e. if there are unfulfilled dependencies), is stored in the *Dependence Counts* table shown in Figure 3. Otherwise, the task is ready to run and the *Task Maestro* writes its function pointer along with its *Task Pool* index to the *Ready Tasks FIFO* of the *Nexus IO* unit.

The dependency information is stored in the *Dependence Table* shown in Figure 3. It is designed to ensure fast lookups, by implementing a simple hashing mechanism. This way, each memory address can be mapped to one entry in the hash table. In [9], it was shown that in order to reduce lookup time when a hash collision occurs, the hash table is replicated in a set-associative structure. This way, an input/output lookup requires only one visit to the *Dependence Table*, rather than multiples of that when implementing a linked-list structure inside a larger hash table, as was the case in [8].

For each entry in the hash tables, a *Kick-Off List* is maintained containing the task IDs that are waiting for this memory address to be released. If for a certain memory address the number of tasks that can be stored in the *Kick-Off List* is exceeded, a new *Kick-Off List* will be allocated in a separate table, the *Dummy Kick off Lists* table shown in Figure 3.

Whenever a task is finished, the runtime system communicates its ID to the *Nexus IO*, which writes the incoming data to the *Finished Tasks FIFO*. The *Task Maestro* reads said FIFO, looks up the finished task info from the *Task Pool* and updates the task graph. Finally, the *Task Maestro* deletes the finished task entry from the *Task Pool*.

B. Design Space Exploration

A VHDL test bench was implemented to simulate a configurable multicore system. Among the configurable parameters are the number of cores, core clock frequency, Nexus++ table sizes, etc. The test bench simulates parts of the runtime system and the *Nexus Plugin*. It submits new tasks to Nexus++, receives ready tasks information from it, schedules ready tasks to worker cores and simulates their execution, and finally notifies Nexus++ of finished tasks.

Tasks are based on experimental traces which include task input/output information and task execution times. Thus, task execution is simply modeled by waiting for a certain time. The traces were generated after running the benchmarks as described in Section V-A. The list of configurable parameters and their experimental values are shown in Table I. The sizes of the different tables and lists in the *Task Maestro* were empirically determined, enlightened by previous work ([8],

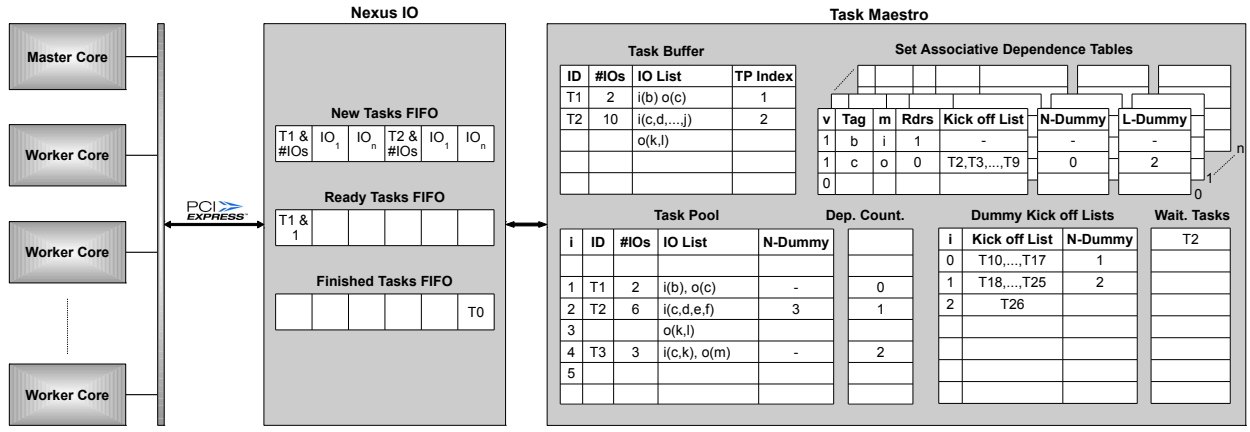

Fig. 3. Nexus++ block diagram in a multicore system

[9]), and by noticing the scalability behaviour when simulating the runs of the different benchmarks described in Section V-A.

IV. VSs Runtime

To integrate Nexus++ into OmpSs, we implemented a runtime library that provides the same API as the Nanos runtime, named VSs.

A. Structure

Our implementation is based on the proto-runtime whose basic principles are described in [16]. This proto-runtime provides the basic management of tasks: creation, switching and destruction. It also provides callback interfaces for implementing the programming model's synchronization directives. This means that to implement the OmpSs functionality, only the following logic had to be added:

Dependency resolution: At the beginning and end of each task, the runtime checks which dependencies are fulfilled. Each data structure has an associated queue that saves pending access requests in the order they were made. New tasks are added to the back of the queue, while finishing tasks free the task(s) at the front of the queue to execute. Multiple tasks that only read a data structure (argument of type `in`) may be launched simultaneously, but only one task (type `out` or `inout`) may have write access to a given data structure at any time. Tasks that have gained access to all their arguments are put into the pool of ready tasks.

Taskwait: Taskwait requests suspend the thread until all child tasks have finished execution. As suspension is already taken care of by the proto-runtime, only a check for the number of remaining child tasks at the end of each child task is necessary. When this number reaches 0, the thread is unsuspended.

*Taskwait on (*p):* Taskwait on suspends the thread, the same as taskwait. The continuation of the thread is added to the access request queue of `p`, as a special type of task that is neither reader nor writer and thus does not block subsequent writers. When the last write access finishes, the thread reaches the front of the queue and is resumed.

Scheduling policy: The policy for selecting a task from the pool of ready tasks is also customizable. In this case a work-first policy was chosen that prefers running existing tasks before the continuation of the master thread, which might create new tasks.

The proto-runtime also contains instrumentation that allows measuring overhead and its decomposition into the various steps of the runtime's core scheduling loop.

B. Comparison with Nanos

The choice of developing a separate implementation of the runtime library means we first have to confirm that Nanos and VSs are equivalent in terms of performance.

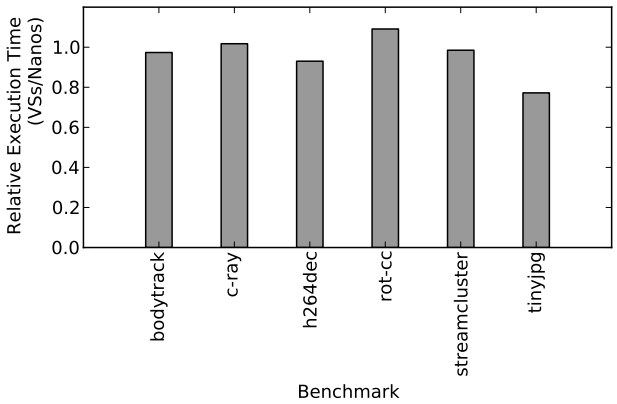

Fig. 4. Execution time comparison between Nanos and VSs, for various benchmarks on a 4-core machine

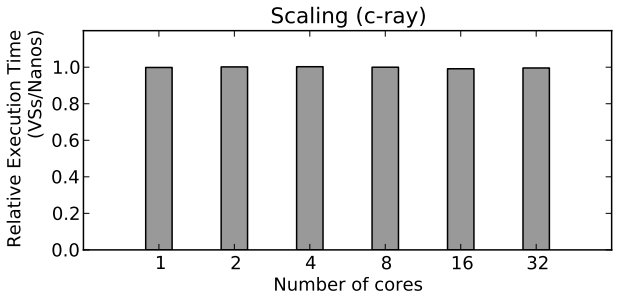

Fig. 5. Scaling behaviour comparison between Nanos and VSs, for the *c-ray* benchmark

We ran two sets of experiments to test the relative performance of the two runtimes. The first, mirroring the conditions in which the FPGA implementation will later be evaluated, was

to run several applications from the Starbench benchmark suite for OmpSs [17] on a machine with an Intel Core i5-2500K 4-core CPU. The applications were compiled using the same flags, with optimization level -O3, using Mercurium 1.3.5.8 and gcc 4.6.3, and linked either against the Nanos runtime or our own replacement VSs runtime.

Figure 4 shows the relative performance of the VSs-linked binary over the Nanos-linked binary for each application. Values below 1 indicate that VSs is faster, values greater than 1 that Nanos is faster. Execution time using either runtime is nearly identical in all cases. Differences are mostly in favour of VSs, except for *rot-cc*. Substituting the Nanos library with our own runtime should therefore not have any significant impact on performance in subsequent experiments beyond allowing us to more easily interface with and instrument the runtime.

To look for differences that may not appear in this setting, we also examined the scaling behaviour of the runtime in a second experiment. On a 40-core Xeon E7-4870 machine running at 2.40GHz, we ran the *c-ray* benchmark, again linked once against Nanos and once against VSs, making different numbers of cores available to the application.

Figure 5 shows the relative performance of the Nanos- and VSs-linked runs of *c-ray* for 1 to 32 cores. Execution times are virtually identical for both versions. Substituting VSs for Nanos in our experiments to evaluate Nexus++ scaling performance should thus not affect the results negatively.

C. Nexus++ Integration

To exploit the Nexus++ hardware accelerator, the VSs runtime library had to be modified to accomodate a *Nexus Plugin* that communicates the relevant information about tasks to the extension board.

Communication is implemented via memory-mapped I/O over PCIe. A Linux driver module was built that provides a device file handle for the board and allows the application to map it into its memory space for direct access. The file read and write operations were also implemented but are not used by the runtime because system calls were found to have significantly larger overheads, doubling communication time. Writing to and reading from mapped memory is performed by the CPU using 32-bit or 64-bit `mov` instructions as appropriate. Both instructions take the same amount of time to perform: writing one word over PCIe takes approximately 250ns in our setup, and reading one word takes 400ns. Hence it is preferable to move 64 bits of data at once if they are available. (It is also possible to use the larger `mov` instructions from the SIMD instruction set extensions, but they do not provide further performance boosts: moving 128 bits takes twice as long as moving 64 bits.)

In the PCIe memory space reserved for Nexus++, the NexusIO unit provides as interface three FIFO queues: When a new task is created, its descriptor has to be written to the *New Tasks FIFO* by the runtime; available tasks can be read from the *Ready Tasks FIFO*; and when a task finishes, the runtime should notify Nexus++ by writing to the *Finished Tasks FIFO*. Each FIFO has an associated status word also mapped to memory. The runtime system has to read this status word to ensure that there is space in the FIFO before writing to either the *New Tasks* or the *Finished Tasks FIFO*, and to ensure that there is data available before reading the *Ready Tasks FIFO*. The status word also reports the current queue occupancy. This allows an optimization to the flow control mechanism also evaluated in Section V-C, where the runtime keeps a conservative estimate of the queue status and completes a number of operations before issuing another read of the status word.

When a new task is submitted and Nexus++ is present in the system, instead of running the dependency resolution algorithm described in Section IV-A, the runtime formats the information about the task's arguments into a Nexus++ task descriptor and writes it to the *New Tasks FIFO*. Similarly, at the end of a task, it will write a finished task notification packet containing the task's identifier to the *Finished Tasks FIFO*. In addition to these modifications to the runtime, a function call is also inserted into the scheduling function to read the *Ready Tasks FIFO* to obtain the tasks that Nexus++ has determined to have all their dependencies fulfilled and may now be executed.

For a task with $n \geq 0$ arguments, a $(n+1)$-word task descriptor is sent to Nexus++, followed by reading 1 word from the *Ready Tasks FIFO*, and then writing a finished task notification of 1 word. So in total, $n+2$ words are written and 1 word is read for a task with n arguments. Ideally, this would lead to a communication overhead of 1150-2150ns for tasks with 1-5 arguments. However, to communicate with the current Nexus++ implementation, the processor also needs to read the status registers to ensure flow control, and change the endianness to conform to the PCIe protocol. These steps incur additional overhead.

V. Performance Evaluation

To evaluate the performance of Nexus++, we run a series of trace-based simulations as well as execute the benchmarks using the Nexus++ FPGA implementation on a PCIe extension board.

A. Benchmarks

The benchmarks used for the evaluation are for the most part taken from the Starbench benchmark suite [17]. These include *c-ray* (ray tracing), *h264dec* (H.264 video decoding), *rot-cc* (image rotation and color conversion) and *streamcluster* (k-median clustering). In addition, two benchmarks from other sources were selected: *emptytask* (a synthetic benchmark) and *sparselu* (sparse LU matrix factorization).

	# tasks	total work (ms)	avg task size (μs)	# deps
c-ray	1200	7381	6151	1
emptytask	1000	1	1	2
h264dec	57051	833	15	2-6
rot-cc	16262	8150	501	1
sparselu	54814	38128	696	1-3
streamcluster	652776	237908	364	1-3

Durations obtained from traces collected on Xeon E7-4870

TABLE II. Overview of Benchmarks

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

emptytask is a synthetic benchmark consisting of a sequence of 1000 tasks of minimal length (one addition) each dependent upon the previous. In effect, this represents a worst case, as there is no parallelism in the application at all, and runtime overhead cannot be hidden behind task duration either. Together with *c-ray* this benchmark gives an idea of the performance range of the different runtimes.

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, has small tasks with many dependencies. This fine-grain parallelism is especially challenging to manage.

B. Simulations

1) Experimental Setup: We collected traces from the execution of each benchmark on a 40-core Xeon E7-4870 machine running at 2.40GHz. These traces include the task descriptors (which specify the inter-task dependencies) and the execution time of each task. Using the information from the traces, we performed three sets of simulations:

No Overhead: To determine the lower bound for the execution time of the benchmarks, we simulated the execution of an application without any overhead. 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: This simulation additionally accounts for the dependency resolution overhead incurred by the Nexus++ core. Failure to scale in this simulation 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.

Nexus++ and runtime: Here, an additional delay of $(n + 1) * 250 + 400$ns is introduced between Nexus++ reporting a task as ready and the start of execution of the task by the worker core, as well as a delay of 250ns between the end of the task and the reception of the finished task notification by Nexus++. This represents the overhead of communication between the processor cores and Nexus++, as described in Section IV-C. Additionally, we measured the overhead due to runtime features that Nexus++ does not replace (such as setting up the stack for the task and switching execution to it) in VSs to be approximately 5μs per task. In this simulation, task length is increased by this constant, to account for all necessary parts of execution.

The VHDL testbench is set up to run Nexus++ at 12.5MHz and the *Nexus IO* FIFOs at 100MHz, to match the speeds at which we run the FPGA implementation.

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.

Of special note is the *h264dec* benchmark, which is the only benchmark to use the `#pragma taskwait on (*p)` construct, which is not currently supported by Nexus++. A `taskwait` instructs the issuing thread to suspend until all child tasks have finished execution, and `taskwait on (*p)` requires to wait only for those child tasks that access data structure `p`. It is thus functionally correct to replace instances of `taskwait on (*p)` with `taskwait`, as our runtime does in the presence of Nexus++, but it will decrease the available parallelism of the application. We report both the *No Overhead* simulation with `taskwait on (*p)`, which represents the parallelism available when executing with the Nanos runtime library, as well as the *No Overhead* simulation with `taskwait on (*p)` replaced by `taskwait`, which is the parallelism that remains available to Nexus++.

2) Results: Figure 6 shows the scaling behaviour of the three simulations, as well as the Nanos-linked run for comparison. Performance is reported as speedups relative to the *No Overhead* simulation on one core (origin of the solid red line), which corresponds to the total amount of work in the application.

a) Performance of Nexus++: First, we discuss the results of the *Nexus++ only* simulations.

For three of the four benchmarks with sufficient available parallelism to scale linearly to 32 cores (*c-ray*, *rot-cc* and *sparselu*), we observe that Nexus++ itself adds overheads of less than 1% in all cases. In comparison, Nanos adds at least 2% overhead in the best case (*c-ray*, *rot-cc*), and up to 194% for the benchmarks with complex inter-task dependencies (*sparselu*).

The *emptytask* benchmark allows us to evaluate the response to lack of parallelism, as well as estimate the overhead in absolute terms. In this benchmark, only a single task is ready at a time, so additional cores cannot improve the performance, but should avoid degrading it. Both Nexus++ and Nanos deal with this case well. In absolute numbers, we see an average overhead per task of 3μs for Nexus++ and 10μs for Nanos.

The final two benchmarks, *h264dec* and *streamcluster*, are applications with limited parallelism. For the H.264 decoder, using the `taskwait on (*p)` construct, a speedup of up to $13\times$ could theoretically be achieved (solid red line). The Nanos runtime, however, is incapable of using this parallelism, achieving less than sequential performance no matter how many cores it uses. Because it does not support `taskwait on (*p)` and instead replaces it with `taskwait`, the parallelism available to Nexus++ is bounded at $3.5\times$ speedup (dashed red line). Even with this limitation, Nexus++ performs much better than Nanos, achieving $2.8\times$ speedup. This shows that for fine-grained tasks, hardware task management is of great benefit.

streamcluster in theory continues to scale linearly beyond 8 cores. In practice, some bottlenecks intervene to limit scaling at $8\times$ for Nexus++, and performance degrades even worse for Nanos when run on more than 4 cores. One peculiarity of the

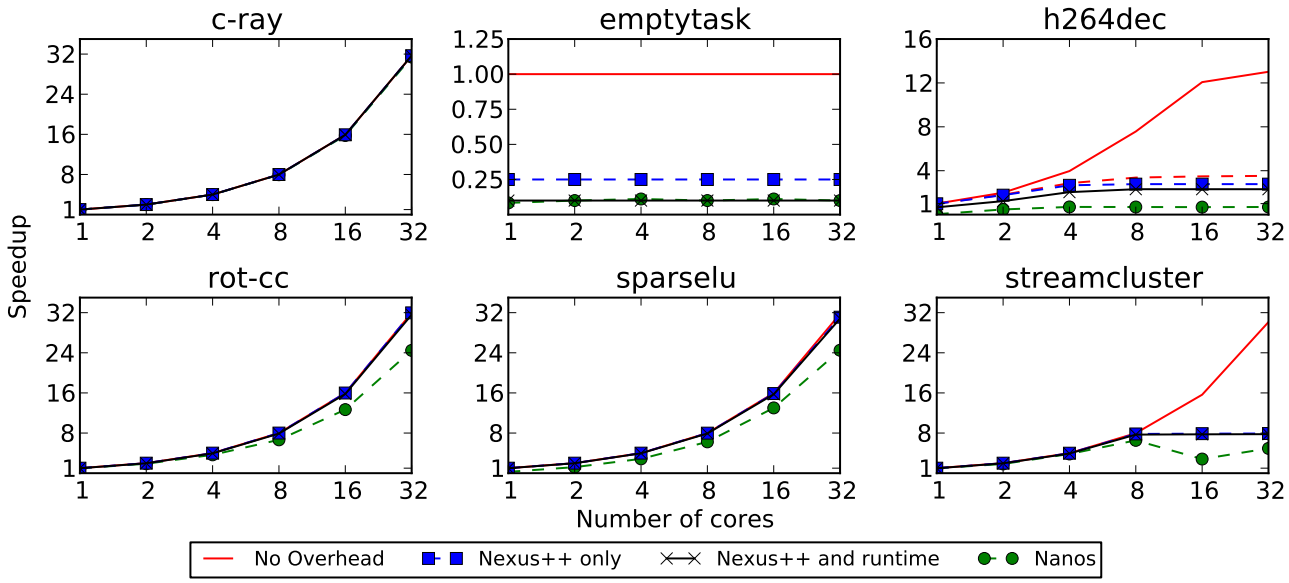

Fig. 6. Scaling behaviour of benchmarks

streamcluster benchmark is that it has many tasks demanding read access to the same memory location, potentially leading to contention of the resources used to manage it. We suspect this may be the reason for the failure to scale beyond 8 cores, with either Nanos or Nexus++.

b) Impact of communication and other overhead: The decision to move dependency resolution away from the worker cores leads to communication, as information about task status is exchanged between cores and the Nexus++ board. In the previous simulation, this information is transmitted in one cycle (10ns), which to realize would necessitate very tight integration into the processor core. In practice, the Nexus++ design will be off-chip, connected via PCIe, which has much higher transmission latencies (1150-2150ns), as noted above.

In addition to the overhead of resolving dependencies, the runtime also has to perform some other functions that Nexus++ cannot replace, such as setting up the stacks and launching the tasks. These functions were measured in VSs to take up to 5μs per task.

If we apply these values to the *Nexus++ and runtime* simulation, we obtain the results shown in black in Figure 6. Although not much space for improvement remains for simple benchmarks like *c-ray* or *emptytask*, an efficient Nexus++-enabled runtime could provide some small benefits compared to Nanos, even when the relatively high PCIe transmission latency is taken into account. For non-trivial benchmarks (*rot-cc*, *sparselu*, *streamcluster*), more significant benefits, up to 3.75×, are indicated. For the fine-grained *h264dec*, despite the fact that a Nexus++-enabled runtime cannot access the full parallelism in the application, it can nevertheless provide 3.34× speedup over Nanos performance on 32 cores.

In summary, we see the following speedups from *Nexus++ and runtime* compared to Nanos:

c-ray	emptytask	h264dec	rot-cc	sparselu	streamcluster
1-1.01×	0.9-1.2×	2.61-16.75×	1.02-1.29×	1.21-3.75×	1.01-2.75×

C. Execution using the Nexus++ FPGA implementation

To confirm that Nexus++ works correctly in practice, we implemented a prototype on an FPGA development board.

1) Experimental Setup: We programmed the Nexus++ design described in Section III on a Xilinx XUPV5- LX110T development board and installed it in a machine equipped with a Intel Core i5-2500K 4-core CPU. The same benchmarks as before were compiled and linked against the VSs runtime described in Section IV. Three versions were prepared:

Software dependency resolution: This is the baseline against which the Nexus++ FPGA implementation is evaluated.

Nexus++: This version uses Nexus++ for dependency resolution. Communication over PCIe is implemented naively.

Nexus++ and improved flow control: This version also uses Nexus++, but the *Nexus Plugin* in VSs is aware of the size of the *New Tasks FIFO* and the *Finished Tasks FIFO*. Instead of checking for every packet if there is sufficient space to receive it, it will read the FIFO occupancy counter once and keep a conservative estimate of space available. Only when there is a possibility of the FIFO being full will it check again. (This version triggers a runtime error in the *h264dec* benchmark, so no results are provided for it.)

2) Results: The results of running the three above-mentioned versions of each of the benchmarks are shown in Figure 7 (the bars for *emptytask* and *sparselu* were scaled by the indicated constant factor to fit the graph). The communication over PCIe shows a significantly greater impact than would be expected from the previous simulations. From the *emptytask* benchmark, we can again derive an absolute estimate of overhead. The naive implementation takes 12.5μs per task. With flow control optimization, the overhead is reduced to 6.9μs per task. In comparison, the software version only takes 1.2μs per task.

Measuring where time is spent inside the *Nexus Plugin* reveals that there are two main sources of overhead. The first,

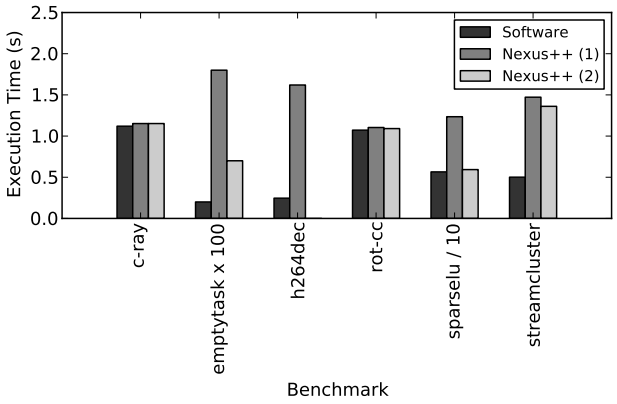

Fig. 7. Dependency resolution in software vs (1) with Nexus++ and (2) with Nexus++ and improved flow control

reading the status registers for flow control, is addressed by the optimization described above. The second is in the nature of the communication protocol: the runtime has to poll the board for new tasks to execute, as the Nexus++ board is not capable of initiating transfers in this implementation. This second source of overhead is especially important for benchmarks which have few parallel tasks at a time, and thus higher likelihood of idle cores polling repeatedly and interfering with transfer of task finish notifications. Accordingly, the results show that for the benchmarks with large amounts of available parallelism such as *rot-cc* and *sparselu*, optimizing the flow control reduces overhead by 40-95%. For *streamcluster*, which only spawns up to 20 tasks at a time, improvement is only 11%, as polling overhead dominates.

For further optimizations, the communication protocol with the Nexus++ FPGA implementation would have to be redesigned. Adding DMA capability to the board would make it possible to remove most of the runtime overhead currently dedicated to communication. For instance, locating the *Ready Tasks FIFO* in the host RAM would obviate the need for polling. In that case, no flow control messages would need to be exchanged over PCIe as the board would only initiate transfers if it wished to acquire new data.

VI. Conclusions

We implemented and evaluated Nexus++, a hardware accelerator for OmpSs as a PCIe extension board, as well as VSs, a runtime library capable of making use of the accelerator. We found that the Nexus++ core can process the dependency resolution for 32 cores efficiently, adding less than 1% overhead in simulation. Communication latency between the extension board and the worker cores over PCIe incurs somewhat higher overheads, but is not the limiting factor. Even taking into account these overheads, we still find expected performance benefits of 240% for fine-grained applications, and 25-30% for applications optimized for the Nanos runtime.

We also ran an early prototype FPGA implementation with actual applications. Unfortunately, the communication protocol between the VSs runtime system and the Nexus++ FPGA implementation is currently suboptimal and adds significant delays. Nevertheless, the extension board is functional and performance reaches levels similar to the software version after only a little optimization. This leads us to conclude that with an improved communication protocol, Nexus++ would significantly speed up dependency resolution for OmpSs applications.

References

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

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

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

[4] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical Task-Based Programming With StarSs," *Int. J. High Perf. Comp. Appl.*, 2009.

[5] 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*, vol. 21, no. 2, pp. 173–193, 2011.

[6] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "Analysis of Dependence Tracking Algorithms for Task Dataflow Execution," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 61:1–61:24, Dec. 2013.

[7] 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.

[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*, 2012, pp. 367 –374.

[9] 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.

[10] M. Andersch, C. C. Chi, and B. Juurlink, "Using OpenMP Superscalar for Parallelization of Embedded and Consumer Applications," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2012.

[11] C. Meenderinck, "Improving the Scalability of Multicore Systems, with a Focus on H.264 Video Decoding," Ph.D. dissertation, Delft University of Technology, 2010.

[12] 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.

[13] 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.

[14] Y. Etsion, A. Ramirez, and R. M. B. Jesuslabarta, "Cores as Functional Units: A Task-Based, Out-of-Order, Dataflow Pipeline," in *Proc. Int. Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, 2009.

[15] 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*, vol. 18, no. 0, pp. 339 – 348, 2013, 2013 International Conference on Computational Science.

[16] S. Halle and A. Cohen, "A Mutable Hardware Abstraction to Replace Threads," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, S. Rajopadhye and M. Mills Strout, Eds. Springer Berlin Heidelberg, 2013, vol. 7146, pp. 185–202.

[17] M. Andersch, C. C. Chi, and B. Juurlink, "Programming Parallel Embedded and Consumer Applications in OpenMP Superscalar," in *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 281–282.