FPGA-Based Prototype of Nexus++ Task Manager

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

ABSTRACT

StarSs is one of several programming models that try to relieve parallel programming. In StarSs, the programmer has to identify pieces of code that can be executed as tasks, as well as their inputs and outputs. Thereafter, the runtime system (RTS) determines the dependencies between tasks and schedules ready tasks onto worker cores. Previous work has shown, however, that the StarSs RTS may constitute a bottleneck that limits the scalability of the system and proposed a hardware task management system called Nexus++ to eliminate this bottleneck. The first prototype of Nexus++ was implemented in SystemC. Its architecture also had a nondeterministic multi-cycle search algorithm in its critical path, potentially limiting its scalability. In this paper, we improved the architecture of Nexus++ and employed a multi-way set-associative cache-like data structures to optimize its search algorithm and increase task throughput. We also modeled the new architecture in VHDL and targeted a Virtex 5 FPGA from Xilinx. Experimental results show that the new architecture is very resource-efficient utilizing only 19% of the target FPGA. It also shows that Nexus++ achieves a speedup of up to 81× using some synthetic benchmarks modeled after H.264 decoding. Hence, Nexus++ significantly enhances the scalability of applications parallelized using StarSs.

1. INTRODUCTION

Due to the advent of multicore architectures, several parallel programming models have been proposed that aim at relieving parallel programming. Examples include Google's MapReduce [5], Intel's TBB [14], and StarSs [13]. StarSs, like OpenMP [3], enables the programmer to express parallelism by adding pragmas to the code. These pragmas identify pieces of code that can be executed as *tasks*, as well as their *inputs* and *outputs*. Based on the inputs and outputs, the RTS can determine the dependencies between tasks and schedule ready tasks onto cores that execute the tasks. The programmer, therefore, does not have to explicitly express dependencies between tasks and the corresponding synchronization. Furthermore, the RTS can also transparently optimize data reuse between tasks and coarsen tasks, thereby relieving the programmer from these burdens. Previous work [11] has shown, however, that the StarSs RTS, when implemented in software, can be a bottleneck that limits the scalability of applications parallelized using StarSs. Roughly speaking, the RTS cannot compute task dependencies and attend to finished tasks fast enough to keep all *worker cores* that execute the tasks busy. The same work therefore proposed a hardware task management system called *Nexus* to accelerate the RTS. *Nexus* was integrated in a simulator of the Cell processor, and could improve the scalability of a synthetic application modeled after a trace of parallel H.264 decoder by a factor of 4.3 when using 16 cores.

Even though Nexus improves the scalability significantly, it has limitations on the number of inputs and outputs a task can have (up to 5 in [11, 10]). Similarly, the number of tasks that can depend on a certain data segment is limited, which limits the applicability of Nexus. Moreover, Nexus does not support double buffering.

In our earlier work [4], these limitations were solved, where a new architecture (*Nexus++*) is described and implemented in SystemC, which is also more efficient than that in Nexus [11], since it uses fewer and simpler tables. Nevertheless, *Nexus++* has a non-deterministic search algorithm for the task graph, which might take more than 10 cycles to search for a certain memory address. Moreover, since implemented in SystemC, *Nexus++* makes many assumptions and approximations regarding for example, memory access times, communication times, driving clock frequency, etc. For this, we present in this paper a VHDL implementation of *Nexus++*.

The main contributions of this paper include: (1) a proof-of-concept fully configurable VHDL prototype targeting a Virtex 5 FPGA from Xilinx. (2) A generic way to be integrated in any task-based RTS, by implementing the *Nexus++ Plugin*, which defines the interface between the target multicore RTS and the VHDL prototype. (3) An improved search algorithm which implements a cache-like multi-way set-associative task graph. Furthermore, the VHDL prototype proves its resource-utilization effectiveness since it utilizes only 19% of the available resources on the target FPGA which is at least 6× more efficient compared to one other major StarSs hardware task manager.

This paper is organized as follows. Overview of the StarSs programming model and related work are described in Section 2. Nexus++ and its features are described in Section 3. In Section 4 the simulation environment and the employed benchmarks are described. The experimental results are presented in Section 5, and conclusions are drawn in Section 6.

2. BACKGROUND

2.1 StarSs

StarSs is a task-based programming model, which enables exploitation of task-level parallelism, regardless of the target architecture. StarSs provides programmers with *pragmas*, an annotations added to the serial code, to identify potential pieces of code that can run in parallel. The programmer does not need to care about synchronization between tasks, as this is done implicitly by the StarSs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

RTS. Listing 1 shows an example of exploiting parallelism using pragmas.

The example in Listing 1 shows that function *decode()* is called inside a nested loop, processing the elements of matrix X. Calculating *decode()* for each element requires the values of the left and upright cells.This example represent macroblock wavefront decoding in H.264 [16], for one 1920×1088 frame in blocks of 16×16, and it is one of the benchmarks used to evaluate Nexus++.

```
MB_type* X[120][68];
//MB_type: a data str. that rep. MB dependencies.
#pragma css task input(left, upright) inout(this)
void decode(int* left, int* upright, int* this){...}
void main(){
 int i, j;
 init_matrix(X) ;
 for(i=0; i<120; i++)
  for(j=0; j<68; j++)
   decode(X[i][j-1], X[i-1][j+1], X[i][j]);
 #pragma css barrier
}
```

Listing 1: StarSs example of macroblock wavefront decoding in H.264

Annotating a function with the *css task* pragma defines a task. The inputs/outputs of the task should also be specified as with function *decode()* in Listing 1. StarSs also provides several synchronization pragmas such as the *css barrier* pragma.

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. As in the example of Listing 1, every time function *decode()* is called, a task is generated.

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

2.2 Related Work

Several 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 [8] assumes independent tasks and uses hardware queues to retrieve tasks with low latency.

In StarSs, tasks can be dependent and it is the responsibility of the RTS to determine their dependencies. An example of a hardware accelerator that targets a certain application domain is a hardware task scheduler optimized for H.264 decoding [1]. It requires, however, that the programmer specifies the dependencies between blocks. Etsion et al. [7] 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 [22], it was only evaluated using high-level simulations. The hardware implementation, compared to ours, is relatively expensive.

As mentioned before, our work builds upon Nexus [11], which was integrated in a simulator of the Cell processor.

3. NEXUS++ HARDWARE TASK MANAGER

The Nexus++ [4] task manager is a hardware accelerator for runtime systems of task-based programming models, StarSs for example. Nexus++ tracks tasks inputs/outputs information and utilizes simple table lookups to find out ready tasks and schedule them to worker cores.

In [4], we presented a SystemC prototype of Nexus++, which thoroughly describes the design and implementation, as well as a

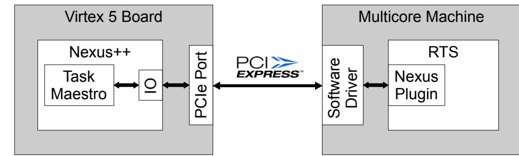

Figure 1: Nexus++ high level system overview

trace-driven evaluation testbench. In this section, we briefly describe Nexus++ highlighting some design changes.

3.1 Design Overview

Nexus++ is rethought to be integrated with real multicore systems. For this, Nexus++ was implemented in VHDL with the Xilinx XUPV5-LX110T [20] FPGA development board. The high level design is depicted in Figure 1. The Nexus++ task manager resides on the FPGA board, and communicates with the RTS on the multicore system using the PCIe bus.

To achieve this:

- Nexus++ has to be implemented, synthesized, and realized on the FPGA board,
- a communication interface that utilizes the (1-lane) PCIe port on the FPGA and exchanges data with Nexus++ has to be implemented,
- a software driver that enables data flow between the RTS and the FPGA using the PCIe bus has to be implemented,
- the RTS has to be modified in order to replace its current task graph management mechanism by a communication unit (*Nexus Plugin*) with Nexus++,

Eventually, the FPGA board can be plugged in the host multicore machine and the task graph management responsibilities can be downloaded to Nexus++.

3.2 Functional Overview

Figure 2 shows the block diagram of our proposed task manager. It is mainly composed of two units; Nexus input/output unit (*Nexus IO*) which handles communication with the host RTS, 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 multicore system under consideration is assumed to have one *Master Core* that executes the main thread and creates *Task Descriptors*, and several worker cores that execute the tasks. A *Task Descriptor* contains task-related information such as its function pointer and input/output list. Nexus++ is responsible for task graph management carried out by the RTS.

As shown in Figure 1, data communication occurs between Nexus++ and the RTS (via the *Nexus Plugin*). So when the master thread creates new tasks, the RTS submits them to Nexus++. Nexus++ sends ready tasks ids to the RTS, and whenever a worker core finishes a task, the RTS notifies Nexus++ of that.

Each FIFO list in the design is generated using Xilinx Coregen v14.4 FIFO Generator 9.3 [21]. We chose to use First-Word Fall-Through FIFOs, which are FIFOs with registered output. This enables the designer to look ahead to the next available word in the list without issuing a read operation.

The *Nexus IO* unit is designed to communicate with the PCIe port at the FPGA board. Xilinx provides an integrated endpoint for PCIe designs Compliant with the *PCI Express Base 1.1* specification, along with an example design that supports single double word (32-bit) payload read and write PCIe transactions [18].

The RTS submits new tasks to the *Nexus IO* unit, which stores them as 32-bit (PCIe payload width) words in the *New Tasks FIFO*

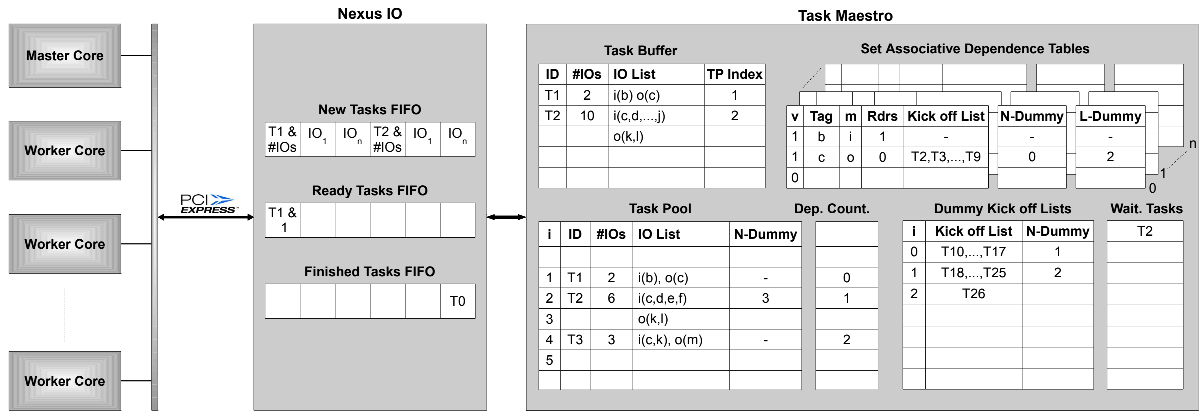

Figure 2: Nexus++ block diagram in a multicore system

list. Each task starts with a header word containing its function pointer and the number of inputs/outputs, and later words containing task's inputs/outputs. Checking the *New Tasks FIFO* list's full flag before submitting new tasks, the RTS stalls when necessary.

The *Task Maestro* then reads the *New Tasks FIFO* list, generates *Task Descriptors* and stores them in (1) a temporary task storage table called the *Task Buffer*, and (2) in the *Task Pool*, the main task storage table in Nexus++, where tasks reside until the end of their life cycles. The *Task Pool* index at which a certain *Task Descriptor* is being stored becomes the unique identifier of this task inside Nexus++. This is a smart way of addressing the *Task Pool*, since no address lookup is required at all.

The *Task Buffer* and *Task Pool* are shown in Figure 2. The *Task Buffer* is relatively small (8 - 16 tasks) and is important, along with the *New Tasks FIFO* list, to decouple task submission from processing new tasks. Both tables are implemented as dual-port block rams. One port is accessed by the process writing new tasks to them. The second port of the *Task Buffer* is accessed when processing new tasks, and this happens in order. That is why this table is of a small size and tasks' data is not preserved after processing it. The *Task Pool*, on the other hand, is much larger (256 - 1024 tasks), and its second port is accessed arbitrarily upon processing finished tasks. Having three processes accessing the dual-port tables, the *Task Buffer* could not be merged in the *Task Pool*.

The *Task Maestro* reads incoming tasks one by one from the *Task Buffer*, builds up the task graph and calculates the *Dependence Count* for the task in progress. This is done by comparing every single input/outputof the new task against all inputs/outputs of all previously submitted tasks. The resulting *Dependence Count*, if > 0, is stored in the *Dependence Counts* table shown in Figure 2. 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* list inside the *Nexus IO* unit. The RTS polls the later list *valid* flag, reads it when it goes high, and schedules ready tasks to run.

Whenever a task is finished, the RTS communicates its *Task Pool* index to the *Nexus IO*, which writes the incoming data to the *Finished Tasks FIFO* list. After that, the *Task Maestro* reads the latter list for the address of the finished task in *Task Pool*, then reads the finished task info from the *Task Pool* and updates the task graph, and finally deletes the finished task entry from the *Task Pool*.

The detailed data/control flow inside Nexus++ was described in [4]. There, a *Task Controller* per worker core was responsible for double buffering and communicating with Nexus++. In the new design described above, those *Task Controllers* are to be implemented as part of the *Nexus Plugin* of the RTS.

3.3 Set Associative Dependence Table

The *Dependence Table* shown in Figure 2 is the storage place of the task graph. Every memory location accessed by one or more tasks will have an entry in the *Dependence Table*. Every time a new memory location is submitted (as an input/output of a task) to Nexus++, the *Task Maestro* searches the *Dependence Table* for this memory location. If it was not found, the *Task Maestro* inserts the new memory location to the dependence table (by writing the *Tag* and access mode *m* fields).

In [4], the dependence table was a hash table with a simple separate chaining hash collisions resolution algorithm $h()$. There, each memory location can map to one location in the *Dependence Table*. If a collision occurred between memory locations *x* and *y* on a certain *Dependence Table* entry, *a* for example, the later memory location (*y*) will be assigned another entry in the *Dependence Table*, *b* for example, and a link to *b* will be inserted in *a*, creating a linked-list structure inside the *Dependence Table*. This implies that searching the *Dependence Table* will include multi-hub accesses until finding the correct memory location. Moreover, if a certain memory location was accessed for the first time, but its hash address in the *Dependence Table* had a long chain of entries, then the *Dependence Table* will be accessed many times until the end of the chain, with ultimately a negative search result. For this reason, we introduce the *Set-Associative Dependence Table*, a cache-like structure for maintaining the task graph.

When Inserting a memory location in the new *Dependence Table* shown in Figure 2, it can be stored in one of the n-way structure. If all n locations, which a certain memory address *A* maps to, are full, then *A* has to wait and the *Task Maestro* stalls. Unlike the *Dependence Table* in [4], searching the n-way set-associative table for a certain memory address costs only one read operation of the different lanes. Comparing the valid*(v)* and *Tag* fields will determine whether the searched memory address is found or not.

Dependency resolution is performed by maintaining a *Kick-Off List* for each memory address. A *Kick-Off List* of a memory address has room for 8 tasks. It records for each task valid and access mode flags, in addition to the task's *TP index*. There is a counter per *Kick-Off List* in addition to pointers to the head and tail of the list. For example, when task *T2* is submitted to Nexus++, its input/output list will be processed one by one. Task *T2* has 4 inputs and 2 outputs as shown in Figure 2. When searching the *Dependence Table* for *T2's* first input, namely memory address *c*, the *Task Maestro* will find that *c* was previously inserted to the *Dependence Table* as an output to an older task. Therefore, *T2* will be added to the *Kick-Off List* of *c*, and the *Dependence Count* of *T2* will be incremented once. On the other hand , if *c* was inserted as input to an older task, and since that *c* is also input to *T2*, then only the readers count *Rdrs*

field in the *Dependence Table* will be incremented, without changing the *Dependence Count*. After processing all memory pointers in the input/output list of the new task, if the resulting *Dependence Count* is 0, then this task is ready and will be written in the *Ready Tasks FIFO* list.

Whenever a task has finished executing, its input/output list will be fetched from the *Task Pool* and processed. For example, when task *T1* finishes, its input/output list is looked up in the *Dependence Table*. Memory address *b* has an empty *Kick-Off List*, and therefore can be invalidated. Memory address *c*, on the other hand, has some tasks in its *Kick-Off List*.The *Task Maestro* reads the first task (*T2*) in this *Kick-Off List* and decrements its *Dependence Count*. If the resulting *Dependence Count* equals 0, then task *T2* will be sent to the *Ready Tasks FIFO* list. Finally, depending on whether *T2* is reading-only or writing memory address *c*, the *Task Maestro* decides to further read tasks from the *Kick-Off List(c)* or not.

The short example above shows how Nexus++ handles read-after-write dependencies. Nexus++ handles also write-after-read and write-after-write hazards (although these two are false dependencies) as was described in [4].

3.4 Dummy Tasks and Entries

In order to support arbitrary number of of inputs/outputs per task, we introduced dummy tasks in [4]. For example, task *T2* shown in the *Task Pool* in Figure 2 has 6 memory locations in its input/output list. Since in our design, each entry in the *Task Pool* can have up to 4 inputs/outputs, the 2 extra parameters are stored in another entry (at TP(3)) in the *Task Pool*, and a pointer to TP(3) is inserted in the next dummy (*N-Dummy*) field of TP(2) where the first 4 parameters reside. Although this solves the problem of having a fixed, limited number of inputs/outputs per task, the maximum number of inputs/outputs is still bounded by the size of the *Task Pool*.

The same principle can be deployed in the *Dependence Table*, where the *Kick-Off List* has a limited size of 8, thus restricting the number of tasks that might depend on a certain memory segment. An example is shown in the *Dependence Table* in Figure 2. Memory address *c* has more than 8 tasks waiting for it. The first 8 tasks are recorded in the direct *Kick-Off List* of *c*, and the extra ones are recorded in an additional table specially created to handle dummy *Kick-Off Lists*. Two pointers point to the dummy *Kick-Off List(s)* are recorded in the original *Dependence Table*. The next dummy (*N-Dummy*) pointer which points to the immediate following *Kick-Off List*, and the last dummy (*L-Dummy*) pointer which points to the last dummy *Kick-Off List* that might still have some room for more task ids. The *L-Dummy* pointer indicates where should any other waiting tasks be added to, while the *N-Dummy* pointer is important when the *Kick-Off List(c)* is to be read, since reading *Kick-Off Lists* should be performed in a first-in first-out order.

4. EXPERIMENTAL SETUP

4.1 Benchmarks

Several benchmarks were used to evaluate Nexus++. First, we used a trace of parallel H.264 decoder decoding one full HD frame on a Cell Broadband Engine processor [12], consisting of 8160 tasks in total. The trace consists of tasks input/output information, tasks execution times and the time they have spent reading/writing their inputs/outputs from/to memory. On average a task spends $7.5\mu s$ for accessing off-chip memory and $11.8\mu s$ for execution [2]. The benchmark processes a matrix of 120 × 68 macroblocks and the dependency pattern is shown in Figure 3(a) [15].

To evaluate Nexus++ for a range of dependency patterns, we created two additional synthetic benchmarks derived from the H.264 benchmark. Their dependency patterns are shown in Figure 3(b) and (c). We also used an additional benchmark without dependencies, i.e., one that has only independent tasks, in order to measure the maximum scalability of Nexus++.

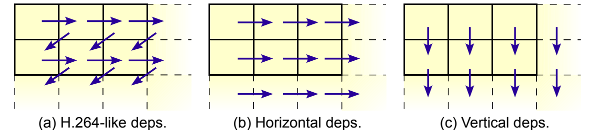

Figure 3: Dependency patterns (120×68 blocks): (a) Ramp effect, (b, c) Fixed # of parallel tasks

System Parameter	Value
Cores clock freq.	2.0 GHz
Nexus++ clock freq.	100 MHz
Nexus++ interface clock freq.	100 MHz
On chip bus bandwidth	2 GB/s
Memory bandwidth	10.67 GB/s
Task Pool size	1024 tasks * 250 bits
No. Parameters per *TD*	4
Task Buffer size	8 tasks * 218 bits
Dependence Table size	8-way * 256 entries * 190 bits
Dummy KoI table size	256 entries * 96 bits
Kick-Off list size	8 tasks

Table 1: System parameters

To validate the dummy tasks/entries approach, the task graph of Gaussian elimination with partial pivoting [16] 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 4, assuming an $n \times n$ matrix.

4.2 Test Environment

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

Nexus++ is simulated assuming a clock cycle time of 10 *ns* (100 *MHz* frequency). Tasks are based on experimental traces, which include tasks input/output information, and their execution and memory access times. Thus task execution and memory access delays are simply modeled by waiting for a certain time. These traces were generated after parallel H.264 video decoding on a Cell processor [12]. Thus, the experiments are assuming a local-stores, shared-memory architecture. Nevertheless, Nexus++ concept can be applied to any other multicore architecture.

The list of configurable parameters and their experimental values are shown in Table 1. The size of the different tables and lists in the *Task Maestro* were empirically determined, enlightened by the design space exploration in [4]. The size of one entry in the

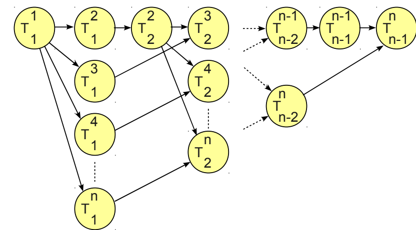

Figure 4: Dependency pattern for the Gaussian elimination benchmark. T_i^j: i, j row and column numbers respectively

Synthesis Parameter	Value	Total	Utilization
Slice Registers	6655	69120	9%
Slice look-up tables (LUTs)	13676	69120	19%
Number of Block RAM	69	148	46%
Nexus++ Maximum Frequency	125 MHz	-	-

Table 2: FPGA resources utilization

Task Pool, for example, equals 250 bits (50-bit per parameter, 22-bit task ID, 8-bit #IOs, 10-bit N-Dummy and 10-bit *Dependence Count*), assuming 4 parameters per entry. Hence the total size of the *Task Pool*, assuming 1024 in-flight tasks, equals 32,000 bytes. The 1024 *Task Pool* size implies the 10-bit width of the N-Dummy and *Dependence Count* fields. It also implies the 22-bit task ID field, since when a task becomes ready, its ID (22 bits) along with its TP index (10 bits) will be concatenated in one 32-bit word and sent to the RTS using the 32-bit PCIe bus.

The *Dependence Table* on the other hand, has 8-way $\times$ 256 entries, with 190 bits per entry, resulting in a total table size of 48,640 bytes. The other tables are relatively smaller, with the *Task Buffer* consuming 218 bytes, and the dummy *Kick-Off List* table consuming 3,072 bytes. In total, the main *Task Maestro's* data structures shown in Figure 2 occupies less than 84 KB of memory.

Table 2, shows device resource utilization collected from the synthesis report of Xilinx tools ver. 14.6 (the latest version at the time of writing this paper) targeting the XUPV5-LX110T [20] FPGA. It shows that the maximum frequency at which our proposed design can be clocked is 125 MHz. It also shows that the number of block rams (brams) used by our design is 69, each of size 36 Kb [17], i.e., total of 310.5 KB. This covers the *Task Maestro's* main 84 KB structures, along to the other FIFO lists, and the Nexus IO unit. Moreover, Table 2 shows that our proposed design consumes relatively few number of slice registers and look-up tables on the target FPGA (9% and 19% respectively of the total available resources).

Off-chip memory (RAM) access time is determined using Cacti 5.3 [9], and was found to be 12 *ns* per 128 bytes RAM chunk, assuming 32-bank 1GB of RAM, which is equivalent to a maximum memory bandwidth of 10.67 GB/s. To evaluate the maximum scalability of Nexus++, we assume a contention-free memory system.

To simulate the PCIe bus, every time the *Master Core* generates a task, the RTS submits it to Nexus++ in 32-bit words, two bits of each as word ID. The first word specifies the 22-bit task's ID (encoded function pointer) and the 8-bit number of inputs/outputs, and every other two words specifies a single parameter(including its 48-bit memory address and 2-bit access mode). The PCIe interface clock frequency according to [19] can be either 62.5 MHz or 125 MHz. For this, we chose to simulate it at 100 MHz. Hence, we assume a bus bandwidth of 400 MB/sec. Therefore, and since the state-of-the art processors run at relatively much higher speed than the 100 MHz driving our communication interface, we did not consider any task preparation delay (30 ns per task in [4]), since it will be masked by the slow communication delay.

5. EVALUATION

Nexus++ was tested under different conditions, varying the number of worker cores, the buffering depth, and with different dependency patterns.

We examined buffering depths in detail in [4], and it was concluded that using double buffering (a worker core reads memory for next task inputs/outputs, while executing another task) is optimal for performance. Using double buffering, the independent tasks benchmark was tested varying the number of cores. Measuring the speedup against the single core experiment, the independent tasks benchmark achieved a speedup of $81\times$ on 128 cores as

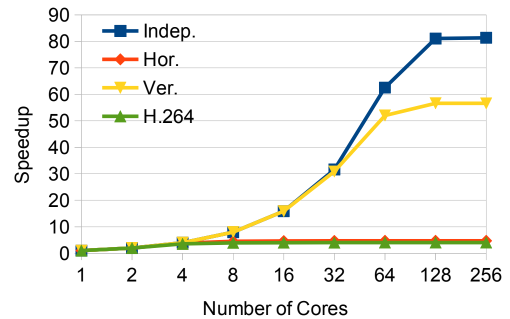

Figure 5: Speedup achieved varying number of cores running tasks with dependencies shown in Figure 3, in addition to the independent tasks benchmark.

shown in Figure 5. Compared to our earlier work [4], the achieved speedup for the same experiment was $221\times$ using 256 cores. This is mainly because Nexus++ SystemC prototype in [4] was clocked at 500 MHz with a bus bandwidth of 2 GB/s. Therefore, we assume that the speedup of $81\times$ on 128 cores is relatively good, especially when compared to StarSs [11], which achieved merely $4\times$ using a similar benchmark with similar average execution times.

Figure 5 also shows the speedup achieved after running the benchmarks in Figure 3. As before, we simulate 8160 tasks with execution and communication times obtained from a parallel H.264 decoder [2]. Showing the same speedup patterns as in [4], the independent tasks and vertical tasks scaled well, while the other two relatively achieved small speedup (Max $4.7\times$). This is due to the complex dependency pattern in the H.264 benchmark, and the serial dependency pattern in the horizontal tasks, which matches the order of task submission. In addition to the low bus bandwidth (400 MB/sec) which highly increases task submission overhead.

Figure 6 shows the speedup achieved by running the Gaussian elimination problem (Figure 4) on different multicore systems for different matrices of sizes ranging from 250 $\times$ 250 to 10000 $\times$ 10000. In this benchmark, a certain memory segment can have huge number of dependent tasks. Although the size of the *Kick-Off List* of each of the *Dependence Table* entries is equal to 8, employing dummy entries as described in Section 3.4 in the *Dependence Table*, Nexus++ could handle the Gaussian elimination problem for matrices of large sizes. As shown in Figure 6, the matrix size has a great impact on the speedup gain and the scalability of the system, since a larger matrix results in a larger number of tasks of larger granularity. A 5000 $\times$ 5000 matrix scaled up to 32 cores with a speedup factor of $25\times$. This experiment includes building and managing a task graph of 12,502,499 tasks with 3,523 FLOPs per task on average [4]. Each single core is assumed to be able to do 2 GFLOPS, which means that the average computation time of each of the aforementioned tasks equals $1.77\mu s$. As compared to [4], the speedup achieved for the same matrix size was $45\times$ on 64 cores. The new VHDL design performs relatively better than our SysctemC prototype in [4] since the former runs at 100 Mhz compared to 500 MHz driving the latter, and the bus bandwidth in [4] was 2 GB/s, compared to 400 MB/s in the VHDL prototype. This demonstrates the effect of using set-associativity in the new design of the *Dependence Table*, compared to the linked-link structure implemented in [4]. Maximum matrix size tested was 10000×10000, and Nexus++ achieved a speedup of $50\times$ on 64 cores.

Although the 250 $\times$ 250 has very small tasks ($83.5ns$ per task on average), Nexus++ scaled to 2 cores with a speedup of $2\times$. This demonstrates the applicability of Nexus++ to any kind of applications, even those with very fine grained tasks.

All tables and FIFO lists in the Nexus++ task manager do not ex-

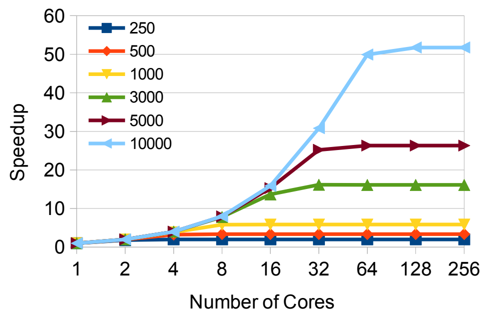

Figure 6: Speedup achieved with different multicore systems running Gaussian elimination for different matrix sizes (legend shows matrix dimension)

ceed 84 KB of memory (mapped to 310.5 KB brams on the target FPGA). Nevertheless, they are sufficient to perform all the objectives of Nexus++. The Task Superscalar [6], on the other hand, consumes more than 6.5MB and still has a static limit (19) on the number of inputs/outputs per task.

Hardware-wise comparison with [22] shows that their design consumes 29,138 registers and 110,729 LUTs respectively, which is at least 6× more than the resources needed by our design (6,655/ 13,676 registers/LUTs respectively). In fact, their design utilizes more than 51% of a Virtex 7 FPGA resources, ours uses only 19% of the smaller Virtex 5 FPGA . Moreover, while Nexus++ achieves 81× on 128-core machine running synthetic benchmark, no evaluation is presented in [22].

Utilizing only 19% of the target Virtex 5 FPGA, proves the compactness and optimization of Nexus++ while being able to achieve decent speedups. This also leaves so much room to increase the size of some tables to accommodate larger number of in-flight tasks, or even for more research ideas like distributed task management architecture by duplicating some structures and building the synchronization/communication logic between them.

The main limiting factor we observed is the PCIe communication overhead, which becomes obvious if tasks have any kind of dependencies. Although simulated at 400 MB/sec, some packets are reserved for control data in real life, which means that the PCIe provides less bandwidth in practice. Therefore, we conclude that our design should be moved on-chip.

6. CONCLUSION

We have presented the first VHDL hardware implementation of Nexus++, our hardware task management accelerator for the StarSs RTS. In addition to supporting double buffering and being able to handle tasks of arbitrary number of inputs/outputs, and arbitrary number of dependent tasks on a certain memory segment, our hardware prototype makes two other main contributions. First, it presents a proof-of-concept prototype of Nexus++. Second, it can handle large task graphs more efficiently using very low-latency lookup tables, by employing set-associativity, smart addressing scheme, and FIFO lists with registered output. Moreover, Nexus++ presents a fully configurable architecture.

Experimental results obtained using a ModelSim testbench show that Nexus++ achieved a speedup of 81× on a 128-core pseudo-machine for a benchmark modeled after H.264 decoding. 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 50× for an 10000 × 10000 matrix using 64 cores. Finally, we have found that the PCIe communication overhead limits the scalability of Nexus++, therefore, we will focus in the future to integrate Nexus++ on chip with the multicore system. Although Nexus++ targets StarSs applications, parts of it can be reused for other programming models. For example, it contains hardware queues that can be used for low-latency retrieval of independent tasks.

7. ACKNOWLEDGEMENTS

This project receives funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the LPGPU Project (www.lpgpu.org), grant agreement n° 288653.

8. 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] C. C. Chi, B. Juurlink, and C. Meenderinck. Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proc. 24th ACM Int. Conf. on Supercomputing*, 2010.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] H. Laboratories. Cacti 5.3. `http://www.hpl.hp.com/research/cacti/`.

[10] C. Meenderinck. *Improving the Scalability of Multicore Systems, with a Focus on H.264 Video Decoding*. PhD thesis, Delft University of Technology, 2010.

[11] 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.

[12] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, 2005.

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

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

[15] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.

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

[17] Xilinx. Virtex-5 Family Overview, DS100 (v5.0). `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, 2009.

[18] Xilinx. Endpoint Block Plus v1.15 for PCI Express User Guide. `http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus/v1_15/pcie_blk_plus_ug341.pdf`, 2011.

[19] Xilinx. LogiCORE IP Endpoint Block Plus v1.15 for PCI Express. `http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus/v1_15/pcie_blk_plus_ds551.pdf`, 2011.

[20] Xilinx. ML505/ML506/ML507 Evaluation Platform User Guide. `www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf`, 2011.

[21] Xilinx. LogiCORE IP FIFO Generator v9.3. `http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v9_3/pg057-fifo-generator.pdf`, 2012.

[22] 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.