Nexus: Hardware Support for Task-Based Programming

Cor Meenderinck
Delft University of Technology
Delft, the Netherlands
cor@ce.et.tudelft.nl

Ben Juurlink
Technische Universität Berlin
Berlin, Germany
b.juurlink@tu-berlin.de

***Abstract*—To improve the programmability of multicores, several task-based programming models have recently been proposed. Inter-task dependencies have to be resolved by either the programmer or a software runtime system, increasing the programming complexity or the runtime overhead, respectively. In this paper we therefore propose the Nexus hardware task management support system. Based on the inputs and outputs of tasks, it dynamically detects dependencies between tasks and schedules ready tasks for execution. In addition, it provides fast and scalable synchronization. Experiments show that compared to a software runtime system, Nexus improves the task throughput by a factor of 54 times. As a consequence much finer-grained tasks and/or many more cores can be efficiently employed. For example, for H.264 decoding, which has an average task size of $8.1\mu s$, Nexus scales up to more than 12 cores, while when using the software approach, the scalability saturates at below three cores.**

***Keywords*-task management; hardware support; StarSS; parallel programming;**

I. Introduction

To enable a large body of software engineers to time-efficiently program novel multicore platforms, new parallel programming models are being developed. The task abstraction is one of the most used constructs for expressing parallelism. Inter-task dependencies, however, can complicate task-based parallel programming. Several programming models, such as StarSS [1], Rapidmind [2], and Sequoia [3] solve this issue by handling task dependencies automatically at runtime. The programmer only has to specify the tasks' input and output by annotating the code.

The ease-of-programming provided by these models comes, however, at the cost of dynamic overhead, affecting the scalability of the system. Especially for fine-grained tasks, the runtime system constrains the speedup. In [4] the scalability of StarSS was analyzed. It was proposed to improve the scalability by providing hardware acceleration of the runtime system, and a concept of such a hardware system was presented. Based on that concept, in this paper we describe the design and evaluation of the Nexus hardware support system for task-based programming models, specifically for StarSS. Nexus accelerates the task dependency resolution process and provides scalable synchronization, thereby improving the scalability of the overall multicore system.

This paper is organized as follows. The Nexus system is described in Section II. Nexus is evaluated in Section III. Section IV concludes the paper.

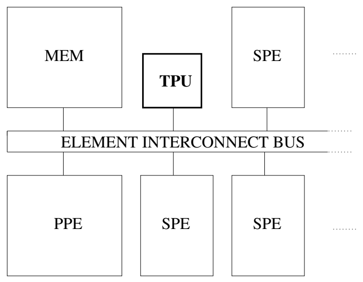


Fig. 1. Overview of the Nexus hardware (in bold) integrated in the Cell processor.

II. Nexus: a Hardware Task Management Support System

In this section we present the design of the Nexus system, which provides the required hardware support for task management in order to efficiently exploit fine-grained task parallelism with StarSS. Nexus can be incorporated in any multicore architecture. In this section, as an example, we present a Nexus design for the Cell processor.

Nexus consists, in its simple form, of a single hardware unit that is added to the system as shown in Figure 1. This Task Pool Unit (TPU) receives tasks descriptors from the PPE, which contain the meta data of the tasks, such as the function to perform and the location of the operands. It resolves dependencies, enqueues ready tasks to a memory mapped hardware queue, and updates the internal task pool for every task that finishes. The TPU is designed for high throughput and therefore it is pipelined. It is directly connected to the bus to allow fast access from any core. The TPU is a generic unit and can be used in any multicore platform.

A. Design of the Task Pool Unit

The block diagram of the Nexus TPU is depicted in Figure 2. Its main features are the following. Dependency resolution consists of table lookups only, and therefore has low latency. The TPU is pipelined to increase throughput. All task descriptors are stored in the task storage to avoid off-chip communication.

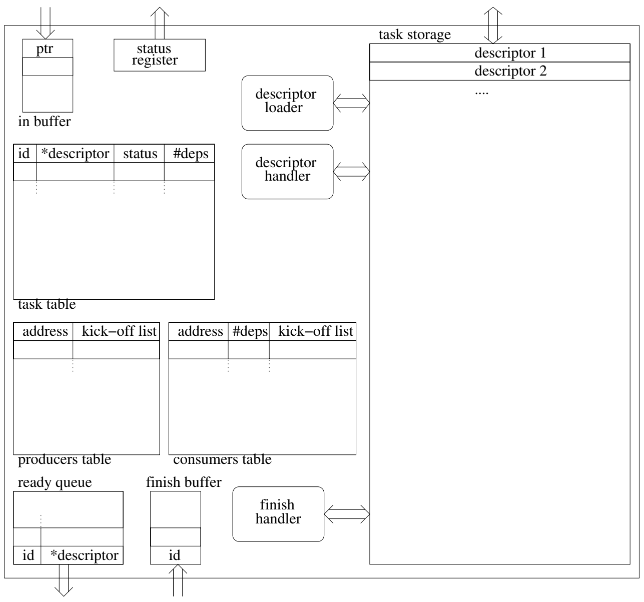


Fig. 2. Block diagram of the Nexus TPU. Internal communication is not depicted, except for that to the task storage.

The life cycle of tasks starts at the PPE that prepares the task descriptor and writes the pointer to it in the in-buffer of the TPU. The descriptor loader reads task descriptor pointers from the in-buffer and loads the descriptor into the task storage, where it remains until the task is completely finished.

Once the descriptor is loaded into the task storage, the descriptor handler processes the descriptor and fills the three tables with the required information. These three tables together, resolve the dependencies among tasks, as described below. This process can be performed fast, as it consists of simple lookups only. There is no need to search through the tables to find the correct item.

The task table contains all tasks in the system and records their status and the number of tasks it depends on. Tasks with a dependency count of zero are ready for execution and added to the task queue. The producers table contains the addresses of data that is going to be produced by a pending task. Any task that requires that data, can subscribe itself to that entry. Thus, this table is used to prevent write-after-read hazards. Similarly, the consumers table is a table containing the addresses of data that are going to be read by pending tasks. Any new task that will write to these locations can subscribe itself to the kick-off list. This table prevents read-after-write hazards. The lookups in the producers and consumers tables are addressed based. As the lists are much smaller than the address space, a hashing function is used to generate the index. Write-after-write hazards are handled by the insertion of a special marker in the kick-off lists.

The three tables, the two buffers, and the ready queue all have a fixed size. Thus, they can be full in which case the pipeline stalls. For example, if the task table is full, the descriptor handler and the descriptor loader stall. If no entry of the task table is deleted, the in-buffer will quickly be full too, which stalls the process of adding tasks by the PPE. Deadlock can not occur, because tasks are added in serial execution order.

The SPEs obtain tasks by reading from the ready queue. The task descriptor is loaded from the task storage after which the task operands are loaded into the local store using DMA commands. When execution is finished and the task output is written back to main memory, the task id is written to the finish buffer. Optionally, double buffering can be applied, by loading the input operands of the next task while executing the current task.

The finish handler processes the task ids from the finish buffer. It updates the tables and adds tasks whose dependencies are met to the ready queue. Finally, the finished task is removed from all tables.

B. Dependency Resolution

Task dependencies can be represented by a task dependency graph. Building and maintaining such a task dependency graph, however, is very time-consuming. In the TPU, dependencies are resolved using three tables. Thus, the dependency resolution process can be performed fast, as it consists of simple lookups only. There is no need to search through the tables to find the correct item.

The task table contains an entry for all tasks in the system. An entry contains the task id (*id* in Figure 2), a pointer to the task descriptor (**descriptor*), the status of the task (*status*), and the dependency count of the task (*#deps*). The dependency count field contains the number of unresolved dependencies for the task. Tasks with a dependency count of zero are ready for execution and added to the ready queue. The task table is indexed by the task id.

The producers table is used to resolve read-after-write hazards. It contains an entry for each address of data that is going to be produced by a pending task. The entry contains the address of the data and a kick-off list, which contains the ids of tasks that will consume that data. When a task finishes and the data is produced, the finish handler processes the kick-off list of that data in the producers table. That is, for each task id in the kick-off lists, it decreases the dependency count of that task in the task table. When a new task is added to the system, the addresses of its input operands are checked for in the producers table. If the address is found, the new task is subscribed to that entry by adding its id to the kick-off list and increasing its dependency count in the task table.

To illustrate this process, consider the example depicted in Figure 3. Figure 3(a) depicts the data that is produced or consumed by tasks $T1$ through $T3$. Figure 3(b) illustrates the dependencies. The dependency resolution process is illustrated in Figure 4. Each row shows the changes to the relevant entries of the three tables for a particular step. In the top three rows, the tasks $T1$ through $T3$ are added to the system, which is performed by the descriptor handler. The bottom two rows show how the finish handler updates the tables when it processes $T1$ and $T2$.

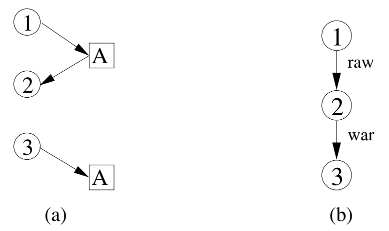
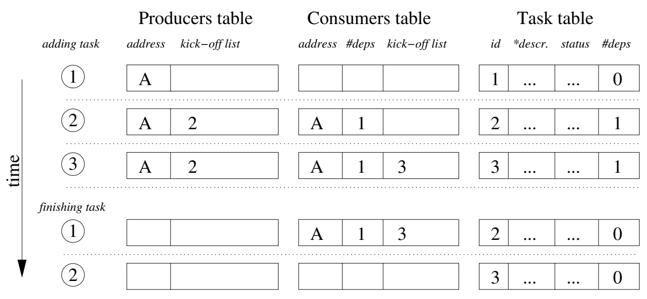


Fig. 3. (a) An example of tasks writing to and reading from the same address A and (b) the corresponding task dependency graph.



	Producers table		Consumers table			Task table			
adding task	*address*	*kick−off list*	*address*	*#deps*	*kick−off list*	*id*	**descr.*	*status*	*#deps*
1	A					1	...	...	0
2	A	2	A	1		2	...	...	1
3	A	2	A	1	3	3	...	...	1
finishing task									
1			A	1	3	2	...	...	0
2						3	...	...	0

Fig. 4. The dependency resolution process for the example of Figure 3.

In the example task $T1$ writes to address A while task $T2$ reads from address A. Thus, $T2$ depends on $T1$. When $T1$ is added by the descriptor handler, an entry for address A is generated in the producers table, with an empty kick-off list (row 1). $T1$ is also added to the task table. When $T2$ is added, the producers table is checked for address A. As it is present, $T2$ is added to the kick-off list of that entry (row 2). Further, $T2$ is added to the task table with a dependency count of one. Once $T1$ finishes, the finish handler reads the entry of address A in the producers table. In the kick-off list, it finds $T2$ and decreases the dependency count of $T2$ in the task table (row 4). As the dependency count is zero, $T2$ is added to the ready queue.

Similarly, the consumers table is used to resolve write-after-read hazards. This table contains an entry for each address of data that is going to be read. An entry contains the address of the data, the number of pending tasks that have to read the data (*#deps*), and a kick-off list. Any new task that will read from the specified address increases *#deps*, and any new task that will write to the specified address is subscribed to the kick-off list.

In the example $T2$ reads from address A while $T3$ writes to address A. In this case $T3$ depends on $T2$. $T3$ does not need the output of $T2$, but it should not write to address A before $T2$ has read from it. This write-after-read hazard is handled as follows. When $T2$ is added to the system by the descriptor handler, an entry for address A is generated in the consumers table (row 2). The dependency count of that entry is set to one, while the kick-off list remains empty. In case an entry for address A already exists only the dependency count of that entry is increased by one. When $T3$ is added by the descriptor handler, the consumers table is checked for address A. As the entry is found, $T3$ is added to the kick-off list of that entry (row 3). Furthermore, $T3$ is added to the task table with a dependency count of one. When $T2$ finishes, the finish handler decreases the dependency count of entry A in the consumers table. As it reaches zero, all consumers of address A have read the data, and thus the kick-off list is processed. $T3$ is found in the kick-off list and thus its dependency count in the task table is decreased by one (row 5). It becomes zero, and therefore $T3$ is added to the ready queue.

The mechanisms described above can not deal with all producer-consumer relations. For example, the producers table contains one kick-off list per address. This is sufficient as long as there is only one producer per address. If multiple tasks write to the same address, multiple kick-off lists for that address are required. We solve this issue by using *barriers* in the kick-off lists. Barriers divide kick-off lists in multiple sub-lists, as if they are separate lists. More details can be found in the online technical report.

III. Evaluation of Nexus

To evaluate the Nexus system we used CellSim [5], which is a modular simulator built in the Unisim environment. The simulator was calibrated with the real processor. Most properties are modeled within a 3% accuracy except the DMA latency, which has an error of 7.4% that could not be lowered without affecting the accuracy of other properties. The current trade-off seems reasonable. Overall, the simulator accuracy is fairly well and sufficient for its purpose.

We employed both an H.264 decoding benchmark and a synthetic benchmark to evaluate the Nexus system. The H.264 benchmark is parallelized according to the method described in [6], contains 8160 tasks, with an average execution time of $8.1\mu s$. Each task depends on its left and top-right neighbors if they exists. Due to this dependency pattern, the amount of available parallelism is not constant during execution. In contrast to the H.264 benchmark, the synthetic benchmark allows controlling the application parameters of which the most important one is the task execution time. They are similar with respect to the dependency and data access pattern.

A. Throughput

First we analyze the throughput of the Nexus system. Table I shows the throughput of all components, except for the task execution itself. The throughputs are measured in the simulator and averaged over the entire benchmark. For most parts, the throughput is depending on the number of dependencies, the state of the task pool, and the occurrence of hash collisions. Each of these influences the number of logic operations and table lookups and thus the average total time spent per task.

The descriptor handler has the lowest throughput of all TPU units as it performs the most complex operation. The throughput of the total system, however, is limited by the PPE. It spends approximately 30% of its time filling the task descriptor, i.e., computing and writing data to memory. The remaining time is spent on writing two 32-bit words to the in-buffer of the TPU. As the PPE has no DMA unit, these writes

TABLE I
THE TASK THROUGHPUT OF THE SYSTEM MEASURED USING THE H.264 BENCHMARK.

	throughput	
	cycles/task	tasks/μs
PPE (prepare and submit tasks)	832	3.8
Descriptor loader	6	533
Descriptor handler	81.8	39.1
Finish handler	45.1	71.0

TABLE II
THE EXECUTION TIME AND SCALABILITY OF THE H.264 BENCHMARK.

SPEs	Execution time (ms)		Speedup	Scalability	
	StarSS	StarSS + Nexus		StarSS	StarSS + Nexus
1	118.5	74.9	1.6	1.0	1.0
2	64.9	37.5	1.7	1.8	2.0
4	41.7	19.0	2.2	2.8	3.9
8	40.6	9.9	4.1	2.9	7.6
16	41.1	6.1	6.8	2.9	12.4

are performed by normal write instructions. These retire once the write has been completed and thus incur large delays.

The memory bandwidth utilization of Nexus does not limit the throughput. Only 1.9% of the bandwidth is taken by the process of adding tasks. We conclude that the throughput of the Nexus system is limited by the PPE preparing and submitting tasks, and is maximally 3.8 tasks/μs. Measurements of the software StarSS runtime system show a throughput of 0.07 tasks/μs of the software runtime system. Therefore, the Nexus system provides a 54X throughput improvement.

B. Scalability & Performance

The improved task management throughput increases scalability in two ways. First, given a certain task size, Nexus allows the system to scale to a larger number of cores. Second, given a certain number of cores, Nexus allows to efficiently employ finer grained task sizes, which enables extraction of more parallelism from applications.

Figure 5 depicts the measured scalability of the StarSS + Nexus system, for several task sizes and up to 16 SPEs using the synthetic benchmark. The maximum scalability obtained is 14.3, which is very close to the theoretical scalability limit of 14.5 for the benchmark. For a task size of $11.2\mu s$ a scalability of 13.9 is obtained, whereas for the StarSS system a scalability of 3.4 was measured. For larger task sizes the figure shows that the scalability does not increase much. This is, however, due to the limited parallelism in the benchmark. The figure also shows that for a given scalability, the StarSS + Nexus system allows to use a task size of approximately 10 times smaller.

The results for the H.264 benchmark are depicted in Table II. The performance is improved for any number of cores. Even on a single SPE, a speedup of 1.6 is obtained due to the reduced overhead. The speedup increases with the number of cores, mainly because of the limited scalability of the StarSS system. The software StarSS system scales to 2.9, whereas the StarSS + Nexus system scales up to 12.4.

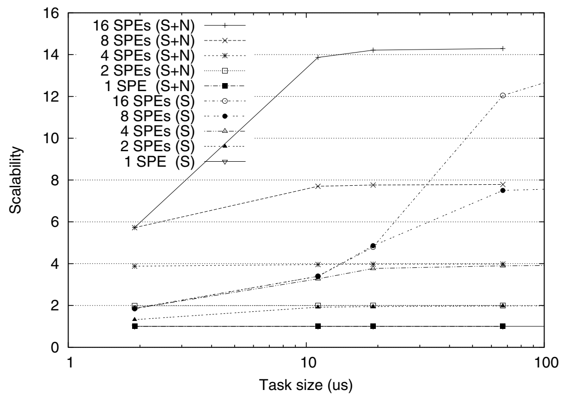


Fig. 5. Scalability of the StarSS + Nexus (S+N) and the software StarSS (S) system measured using the synthetic benchmark.

IV. CONCLUSIONS

To accelerate task management for task-based programming models we have proposed Nexus, which performs dependency resolution in hardware and provides fast synchronization with the worker cores. A comparison has been performed against the software task management of the StarSS programming model. Nexus improves the task management throughput by 54 times. This improvement provides benefits in two ways. First, given a fixed number of cores, using the Nexus hardware allows to use much finer grained task. Measurements show around 10 times (and increasing with the number of cores) smaller tasks can be used while maintaining equal efficiency. Second, given a certain task size, the system scales to much higher core counts. Despite the limitations of Nexus, these results are promising. They show that the overhead of dynamic task management can effectively be reduced. The techniques we used, such as the novel table lookup based dependency detection, are not limited to the Cell processor or the StarSS programming model. Any programming model that performs dependency resolution automatically can benefit from the techniques proposed.

REFERENCES

[1] J. Planas, R. Badia, E. Ayguadé, and J. Labarta, "Hierarchical Task-Based Programming With StarSs," *Int. Journal of High Performance Computing Applications*, vol. 23, no. 3, 2009.
[2] M. McCool, "Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform," in *GSPx Multicore Applications Conference*, 2006.
[3] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally, and P. Hanrahan, "Sequoia: Programming the Memory Hierarchy," in *Proc. Conf. on Supercomputing*, 2006.
[4] C. Meenderinck and B. Juurlink, "A Case for Hardware Task Management Support for the StarSS Programming Model," in *Proc. Conf. on Digital System Design (DSD) - Architectures, Methods and Tools*, 2010.
[5] "CellSim: Modular Simulator for Heterogeneous Multiprocessor Architectures." [Online]. Available: http://pcsostres.ac.upc.edu/cellsim/doku.php/
[6] E. van der Tol, E. Jaspers, and R. Gelderblom, "Mapping of H.264 Decoding on a Multiprocessor Architecture," in *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.