Cor Meenderinck, Ben Juurlink

# A case for hardware task management support for the StarSS programming model

**Conference object, Postprint version**
This version is available at http://dx.doi.org/10.14279/depositonce-5776.

**Suggested Citation**
Meenderinck, Cor; Juurlink, Ben: A case for hardware task management support for the StarSS programming model. - In: 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools : DSD. - New York, NY [u.a.], IEEE, 2010. - ISBN: 978-1-4244-7839-2, pp. 347-354. - DOI: 10.1109/DSD.2010.63. *(Postprint version is cited, page numbers differ.)*

**Terms of Use**
© © 2010 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

WISSEN IM ZENTRUM UNIVERSITÄTSBIBLIOTHEK

Technische Universität Berlin

# A Case for Hardware Task Management Support for the StarSS Programming Model

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*—StarSS is a parallel programming model that eases the task of the programmer. He or she has to identify the tasks that can potentially be executed in parallel and the inputs and outputs of these tasks, while the runtime system takes care of the difficult issues of determining inter task dependencies, synchronization, load balancing, scheduling to optimize data locality, etc. Given these issues, however, the runtime system might become a bottleneck that limits the scalability of the system. The contribution of this paper is two-fold. First, we analyze the scalability of the current software runtime system for several synthetic benchmarks with different dependency patterns and task sizes. We show that for fine-grained tasks the system does not scale beyond five cores. Furthermore, we identify the main scalability bottlenecks of the runtime system. Second, we present the design of Nexus, a hardware support system for StarSS applications, that greatly reduces the task management overhead.**

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

## I. Introduction

Currently, processor performance is mainly increasing due to a growing number of cores per chip. According to Moore's Law the number of transistors per die is expected to double every two years, allowing chips with hundreds of cores within 10 to 15 years. Successful deployment of such many-core chips, depends on the availability of parallel applications. Creating efficient parallel applications, though, is currently cumbersome and complicated as, besides parallelization, it requires task scheduling, synchronization, programming or optimizing data transfers, etc.

Currently there are several efforts to solve the programmability issues of multicores. Among these is the task-based StarSS programming model [1]. It requires the programmer only to indicate functions, referred to as tasks, that are amenable to parallel execution and to specify the input and output operands. All the other issues, such as determining task dependencies, scheduling, programming and issuing data transfers, etc., are automatically handled by StarSS.

The task management performed by the StarSS runtime system is rather laborious. This overhead affects the scalability and performance, and potentially limits the applicability of StarSS applications. In this paper we study the benefits of accelerating task management with hardware support. We show that for fine-grained tasks, the runtime overhead greatly reduces performance and severely limits scalability. Fine-grained task parallelization might be important for future many-core systems as it has greater potential for parallelism than coarse grained task parallelization. Through extensive analysis of the current system's bottlenecks, we specify the requirements of a hardware support system. As an example of such a system, we present Nexus: a hardware task management support system compliant with the StarSS programming model and potentially applicable to other programming models using dynamic task management.

Some other works have proposed hardware support for task management. Most are limited, in contrast to our work, to scheduling of independent tasks leaving it up to the programmer to deliver tasks at the appropriate time. Carbon [2] provides hardware task queueing enabling low latency task retrieval. In [3] a TriMedia-based multicore system is proposed containing a centralized task scheduling unit based on Carbon. For a multicore media SoC, Ou [4] proposes a hardware interface to manage tasks on the DSP, mainly to avoid OS intervention. Architectural support for the Cilk programming model has been proposed in [5], mainly focussing on cache coherency.

A few works include hardware task dependency resolution. In [6] a look-ahead task management unit has been proposed reducing the task retrieval latency. It uses task dependencies in a reversed way to predict due available tasks. Because of the programmable nature, its latency is rather large. The latter issue has been resolved in [7] at the cost of generality: the proposed hardware is specific for H.264 decoding. For System-on-Chips (SoCs), an Application Specific Instruction set Processor (ASIP) has been proposed to speedup software task management [8]. Independently, Etsion et al. [9] have also developed hardware support for the StarSS programming model. They observe a similarity between input and output dependencies of tasks and instructions and have proposed task management hardware that works similar to an out-of-order instruction scheduler.

This paper is organized as follows. The StarSS programming model and the target processor platform are briefly

reviewed in Section II. The benchmarks used throughout this work are described in Section III. In Section IV the StarSS runtime system is analyzed. Section V compares the performance of the current StarSS system to that of a manually parallelized implementation and illustrates the need for hardware task management support. The proposed Nexus system is described in Section VI. Section VII concludes the paper.

## II. BACKGROUND

The StarSS programming model is based on simple annotations of serial code, by adding pragmas. The main pragmas are illustrated in Listing 1. The pragmas `css start` and `css finish` indicate the initialization and finalization of the StarSS runtime system. Tasks are functions that are annotated with the `css task` pragma, including a specification of the input and output operands. The programmer does not have to specify which task can be executed in parallel as the runtime system detects this, based on the input and output operands. In addition to the pragmas illustrated in this example, StarSS provides several synchronization pragmas.

```
int *A[N][N];

#pragma css task input(base[16][16])\
                 output(this[16][16])
void foo(int* base, int* this){
  ...
}

void main(){
  int i,j;
  ...
  #pragma css start
  foo(A[0][0], A[i][j]);
  ...
  #pragma css finish
  ...
}
```

Listing 1. Basics of the StarSS programming model.

The current implementation of StarSS uses one core with two threads to control the application, while the others (worker cores) execute the parallel tasks. A source-to-source compiler, processes the annotated serial code and generates the codes for the control and worker cores. The first thread of the control core runs the main program code that adds tasks and the corresponding part of the runtime system. The second thread handles the communication with the worker cores and performs the scheduling of tasks.

Tasks are added dynamically to the runtime system, which builds the task dependency graph based on the addresses of input and output operands. Tasks whose dependencies are met, are scheduled for execution on the worker cores. Furthermore, the runtime system manages data transfers between main memory and local scratchpad memories, if applicable. It tries to minimize the execution time by applying a few optimizations. First, it creates groups of tasks, referred to as bundles within StarSS. Using bundles, reduces the overhead per task for scheduling. In addition, the runtime system optimizes for data locality by assigning chains within the task graph to bundles. Within such bundles, data produced by one task is used by a next, and thus locality is exploited.

Throughout this paper, we use the StarSS instantiation for the Cell Broadband Engine, which is called CellSS. We use CellSS version 2.2 [10], which is the latest release at time of writing. It has an improved runtime system with significantly lower overhead than prior releases. CellSS has several configuration parameters that can be used to tune the behavior of the runtime system. We manually optimized these parameters to obtain the best performance for small task sizes.

Our experimental platform is the Cell processor, which contains one PowerPC Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The PPE is a general purpose core that runs the operating system. The SPEs are autonomous but depending on the PPE to receive work to do. The CellSS runtime system runs on two threads of the PPE. The SPEs wait to receive a signal from the helper thread indicating what task to perform. Once they are ready, they signal back and wait for another task. The SPEs have scratchpad memory only and use DMA (Direct Memory Access) commands to load data from main memory into their local store. When using bundles, CellSS applies double buffering, which hides the memory latency by performing data transfers concurrently with task execution. The measurements are performed on a Cell blade containing two Cell processors, running at $3.2GHz$. Thus, in total 16 SPEs are available.

One of the main metrics used throughout this paper is scalability, which is defined as the speedup of a parallel execution on $N$ SPEs compared to execution using one SPE. Furthermore, scalability efficiency is defined as $S/N$, where $S$ is the scalability and $N$ is the number of SPEs. For example, if an application takes 1 $ms$ using a single SPE and 12 $ms$ using 16 SPEs, the scalability is 12 and the scalability efficiency is 75%.

## III. BENCHMARKS

To analyze the scalability of StarSS under different circumstances, we used three synthetic benchmarks. We used synthetic benchmarks rather than real applications because it allows controlling the application parameters such as the dependency pattern, the task execution time (and, hence, the computation to communication ratio), etc. The three benchmarks are referred to as CD (Complex Dependencies), SD (Simple Dependencies), and ND (No Dependencies). The dependency pattern of the CD benchmark is similar to the dependency pattern in H.264 decoding [11], [12].

All three benchmarks process a matrix of $1024 \times 1024$ integers in blocks of $16 \times 16$ . Each block operation is one task, adding up to a total of 4096 tasks. The task execution time is varied from approximately $2\mu s$ to over $2ms$.

```
int *A[64][64];

#pragma css task input(left[16][16],\
  topright[16][16]) inout(this[16][16])
void foo(int* left, int* topright, int* this){
  ...
}

void main(){
  int i,j;

  init_matrix(A);

  #pragma css start
  for(i=0; i<64; i++){
    for(j=0; j<64; j++){
      foo(A[i][j-1], A[i-1][j+1], A[i][j]);
    }
  }
  #pragma css barrier
  #pragma css finish
}
```

Listing 2. The code of the CD benchmark including StarSS annotations.

In the CD benchmark, each task depends on its left and top-right neighbor if it exists. Listing 2 shows the simplified code of the CD benchmark, including the annotations. Due to the task dependencies, at the start of the benchmark only one task is ready for execution. During the execution, the number of available parallel tasks gradually increases, until halfway, after which the available parallelism similarly decreases back to one. Therefore, the amount of available parallel tasks at the start and end of the benchmark is lower than the number of cores. Due to this ramping effect in the CD benchmark, the maximum obtainable scalability is 14.5 using 16 cores.

In the SD benchmark, each task depends on its left neighbor if it exists. Thus, all 64 rows are independent and can be processed in parallel. In the ND benchmark all tasks are independent of each other and thus the application is fully parallel. Therefore, the maximum obtainable scalability for the SD and ND benchmarks is 16 when using 16 cores.

## IV. Scalability of the StarSS Runtime System

The main purpose of this section is to quantify the task management overhead of the StarSS runtime system. Besides measuring performance and scalability, we have also generated Paraver traces to study the runtime behavior in more detail. First, we study the CD benchmark in detail, especially for small task sizes, as it resembles H.264 decoding, which has an average task size of approximately $20\mu s$.

### A. CD Benchmark

For the CD benchmark a maximum scalability of 14.2 is obtained, as depicted in Figure 1. This near optimal scalability, however, is only obtained for large task sizes. The figure also shows that in order to obtain a scalability efficiency of 80% when using 16 SPEs, that is a scalability of 12.8, at least a task size of approximately $100\mu s$ is required. Similarly, for 8 SPEs a task size of $50\mu s$ is required. For a task size of $19\mu s$ the scalability is only 4.8.

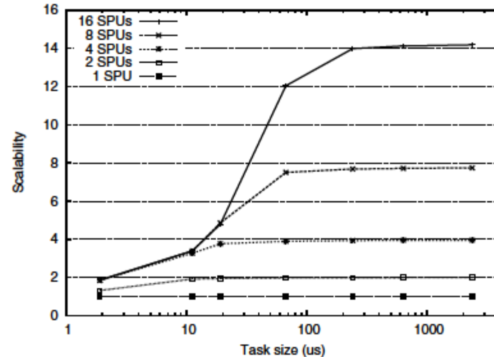


Figure 1. Scalability of StarSS with the CD benchmark.

Figure 2 depicts a part of the Paraver trace of the CD benchmark with a task size of $19\mu s$. The colored parts indicate the execution phases, of which the legend is depicted in Figure 3. The first and second row of the trace correspond to the main and helper thread, respectively. The other rows correspond to the 16 SPEs. The yellow lines connecting the rows indicate communication. The helper thread signals to the SPEs what task to execute. The SPEs, in turn, signal back when the task execution is finished.

The second row of the trace shows that scheduling (yellow), preparing (red), and submitting a bundle (green) takes approximately $6\mu s$, while removing a task (magenta) takes approximately $5.8\mu s$. As the total task size, including task stage in, wait for DMA, task execution, task stage out, and task finished notification, is $22.5\mu s$, the helper thread can simply not keep up with the SPEs. Before the helper thread has launched the sixth task, the first one has already finished. However, its ready signal (yellow line) is processed much later, and before this SPE receives a new task, even more time is spent.

The stress on the helper thread can be reduced by allowing the runtime system to create bundles. Our measurements show that bundles of four tasks provides the optimal performance for this task size. Grouping 4 tasks in a bundle reduces the scheduling time with 45%, the preparation and submission of tasks with 73%, and the removal of tasks with 50%. In total, the overhead per task is reduced from $11.6\mu s$ to $5.5\mu s$ resulting in an overall performance improvement of 20%. The scalability, however, is not affected as the helper thread remains a bottleneck. To efficiently leverage 16 SPEs,

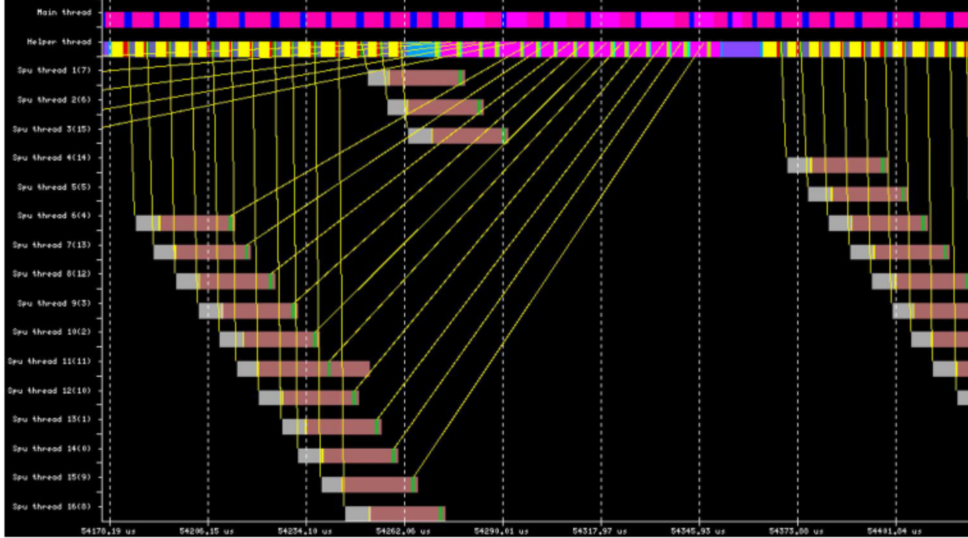

Figure 2. Partial Paraver trace of the CD benchmark with 16 SPEs, a task size of $19\mu s$, and the $(1, 1, 1)$ configuration.

Figure 3. Legend of the Paraver traces.

at most an overhead per task of $22.5/16 = 1.4\mu s$ is required.

The main reason for the bottleneck in the main thread is very subtle. Although the benchmark exhibits sufficient parallelism, due to the order in which the main thread spawns tasks (from left to right and top to bottom), at least 16 "rows of tasks" (corresponding to 1/4th of the matrix) have to be added to the dependency graph before 16 independent tasks are available that can keep all cores busy. Thus, to exploit the parallelism available in the benchmark, the addition of tasks in the main thread must therefore run ahead of the parallel execution. The trace reveals, that the main thread can only keep up with the parallel execution, and thus it limits the scalability of the system.

### *B. SD Benchmark*

In the SD benchmark, tasks corresponding to different rows are completely independent of each other, because tasks only depend on their left neighbor. The difference in dependency pattern has several effects on the runtime system. First, this greatly simplifies the task dependency graph, which in turn might lead to decreased runtime overhead. Second, the dependency pattern enables the runtime system to create efficient bundles, further reducing the overhead. Our measurements, however, show only a small improvement in scalability compared to the CD benchmark.

The scalability of the SD benchmark is slightly better than that of the CD benchmark. As Figure 4 depicts, a maximum scalability of 14.5 is obtained for large task sizes. For a task size of $19\mu s$ a scalability of 5.6 is obtained. Analysis of the Paraver trace shows that the main thread is the bottleneck. It is continuously busy with user code (26.6%), adding tasks (50.9%), and removing tasks (20.6%). On average per task these phases take $2.0\mu s$, $3.8\mu s$, and $1.5\mu s$, respectively. This indicates that the time taken by building and maintaining the task dependency graph is not much affected by the dependency pattern, unlike expected.

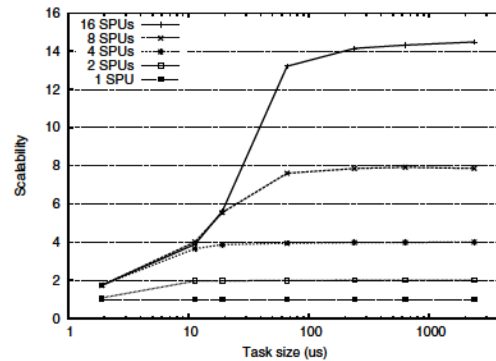


Figure 4. Scalability of StarSS with the SD benchmark.

The stress on the helper thread, on the other hand, is significantly lower compared to the CD benchmark. The reason for this is that we allowed the creation of larger bundles. On average, the time spent on scheduling, preparation of the bundles, and submission of the bundles is $1.86\mu s$, $0.10\mu s$, and $0.20\mu s$, respectively.

### C. ND Benchmark

The scalability of the ND benchmark is depicted in Figure 5. The maximum obtained scalability is 15.8, which is significantly higher than with the two other benchmarks. This is because all tasks are independent. Thus, first, there is no ramping effect and second, as soon as 16 tasks have been added, the runtime system detects 16 independent tasks to execute.

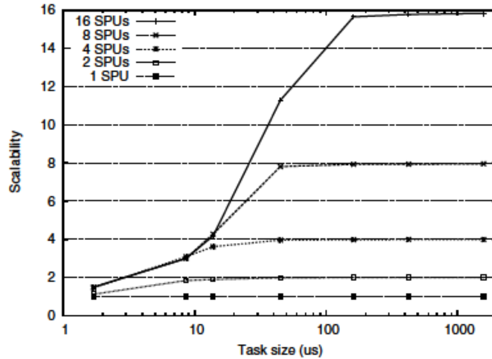


Figure 5. Scalability of StarSS with the ND benchmark.

The improvement for smaller task sizes is also due to the increased available parallelism. The time spent on building and maintaining the task graph, however, is approximately equal to the time spent on these for the SD benchmark, limiting scalability. We conclude that not the dependencies by itself are time-consuming, but the process of checking for dependencies.

## V. A Case for Hardware Support

In the previous section we analyzed the StarSS system and showed that the current runtime system has a low scalability for fine-grained tasks. To show the potential gains of accelerating the runtime system, in this section we compare StarSS to manually parallelized implementations of the benchmarks using a dynamic task pool. Based on the analysis presented in this and the previous section, we identify the bottlenecks in the StarSS runtime system. Those lead to a set of requirements that hardware support for task management should comply with.

### A. Comparison with Manually Parallelized Benchmarks

The benchmarks are also manually parallelized. A dynamic task pool is used to determine which tasks are ready for execution. As the dependencies are predetermined and hard coded, however, there is no dynamic dependency detection, in contrast to the runtime system of StarSS. Moreover, a simple Last-In-First-Out (LIFO) buffer is used to minimize the scheduling latency, where StarSS tries to apply several scheduling optimizations. Therefore, the comparison between the StarSS system to that of the manually parallelized (Manual) system, shows the cost of dynamic dependency detection and the scheduling optimizations.

The task pool implementation was taken from [13]. It is optimized for the Cell processor. The authors have investigated several synchronization primitives. The direct mapped mailboxes proved to provide the fastest communication between the PPE and the SPEs for up to 16 SPEs, although this approach has scalability issues.

The scalability of the CD benchmark with the manually coded task pool is shown in Figure 6. For a task size of $19\mu s$, the scalability using 16 SPEs is almost 12. This is much higher than the scalability of 4.8 obtained with StarSS. Similarly, the scalability of the other two benchmarks increased significantly.

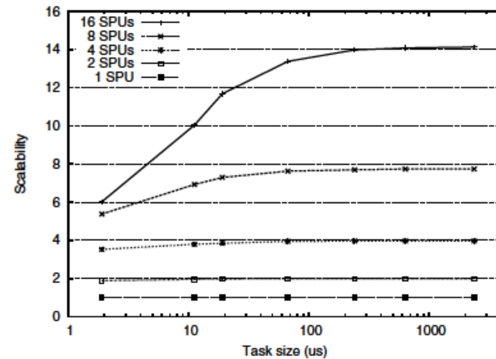


Figure 6. Scalability of the manually parallelized CD benchmark.

Figure 7 compares the scalability of the StarSS system to that of the manually parallelized system. It displays the iso-efficiency lines, which represent the minimal task size required for a certain system in order to obtain a specific scalability efficiency (80% in this graph). For the Manual system and a small number of SPEs, an efficiency of 80% is always obtained, irrespective the task size. Therefore, the corresponding iso-efficiency points do not exist and are not displayed in the graph.

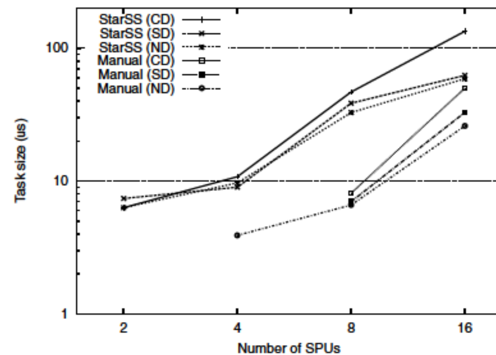


Figure 7. The iso-efficiency lines of the StarSS and Manual systems (80% efficiency).

From the iso-efficiency graph it is clear that the Manual

system achieves an efficiency of 80% for much smaller task sizes than StarSS does. For eight SPEs the difference in task size is between a factor of 5 and 6, while for 16 SPEs the difference is between 2 and 3 times. The main cause for this difference is the dependency resolution process of the StarSS runtime system. The iso-efficiency lines of both systems increase with the number of SPEs. Going from 8 to 16 SPEs, for both systems the lines increase because of off-chip communication. In general, however, there are different reasons. In the StarSS system the throughput of the two PPE threads is the limiting factor. Therefore, with increasing number of SPEs longer tasks are required to hide the task management overhead on the PPE.

In the Manual system, the iso-efficiency line mainly increases with the number of SPEs mainly due to the limited scalability of the synchronization mechanism. When SPEs have finished the execution of a task they send a mailbox signal to the PPE. The helper thread checks the mailboxes in a round-robin fashion. When it finds a signal, it processes the contents of the signal and send a mailbox back indicating a new task. The time between the SPE sending a finish signal and receiving a new task, is depending on the phase of the round-robin polling. In the worst case the SPE send a signal just after the helper thread checking that SPEs mailbox. Moreover, this synchronization latency is increasing with the number of SPEs.

Besides a lower scalability, also the absolute performance of StarSS is lower, up to four times, than that of the Manual System for fine-grained tasks. Table I depicts the execution time of the CD benchmark. For larger task sizes, the difference diminishes as the task management overhead becomes negligible compared to the task execution time. A similar difference between the StarSS and Manual system is observed in the ND and SD benchmarks.

### *B. Requirements for Hardware Support*

From the prior analysis, we conclude that the StarSS runtime system currently does not scale well for fine-grained task parallelism. The main bottleneck is the dependency resolution process. Irrespective of the actual dependency pattern, determining whether there are task dependencies is a laborious process that cannot keep up with the parallel task execution. In order to efficiently utilize 8 or 16 SPEs for H.264 decoding, roughly corresponding to the CD benchmark with a task size of $20\mu s$, the process of building and maintaining the task dependency graph should be reduced from $9.1\mu s$ to $2.5\mu s$ and $1.3\mu s$, respectively. Therefore, we describe the first requirement for task management hardware support as follows:

*Requirement 1:* The hardware support system should accelerate the task dependency resolution process by at least a factor of 3.6 and 7.3, when using 8 and 16 SPEs, respectively.

Preferably this process is accelerated even more, such that it can run ahead of execution in order to reveal more parallelism. As such a large speedup is required, we expect that software optimizations (if possible) or some hardware acceleration such as special instructions, are not sufficient. Instead, we propose to perform this process in hardware.

The second bottleneck is the scheduling, preparation, and submission of tasks, performed by the helper thread in StarSS. In total this overhead per task is $5.5\mu s$, where at least a latency of $2.8\mu s$ and $1.4\mu s$ is required, for 8 and 16 SPEs, respectively.

*Requirement 2:* The hardware support system should accelerate scheduling, preparation, and submission of tasks by at least a factor of 2.0 and 3.9, when using 8 and 16 SPEs, respectively.

The scheduler tries to minimize the overhead by creating bundles of tasks, and tries to exploit data locality by grouping a dependency chain within one bundle. For small tasks, however, the scheduling costs more than the benefits it provides. A rudimentary, yet fast, scheduling approach might therefore be more efficient than a sophisticated one.

Although the numbers mentioned in these first two requirements are specific for the Cell processor, the requirements are in general applicable to other multicore platforms as well. The requirements stem from the fact that the task management overhead is relatively large compared to the task execution. As both are performed in software, changing platform is expected to affect the relative size insignificantly.

The latency of synchronization on the Cell processor is too long and scales poorly due to the round-robin polling mechanism. Interrupts are more scalable than the polling mechanism, but still need to execute a software routine. Instead, SPEs should be able to retrieve a new task themselves instead of waiting for one to be assigned. Hardware task queues, such as Carbon [2], is an example of such an approach. Each core can read a task from the queue autonomously, avoiding off-chip communication and execution of a software routine. Assuming a 5% overhead for synchronization is acceptable, the synchronization latency should be at most $1\mu s$, irrespective of the number of cores.

*Requirement 3:* The hardware support system should provide synchronization primitives for retrieving tasks, with a latency of at most $1\mu s$, irrespective of the number of cores.

The round-robin synchronization mechanism is specific for the Cell processor. Despite, the need for low-latency and scalable synchronization is platform independent. In [2] it was shown that synchronization through a hardware queue is beneficial for a wide range of platforms. Therefore, Requirement 3 is also generally applicable.

When compliant with these three requirements, a hardware support system will effectively improve the scalability of StarSS, while maintaining its ease of programming. In the next section we propose the Nexus system, which is developed based on this set of requirements.

Table I
ABSOLUTE EXECUTION TIME, IN $ms$, OF THE CD BENCHMARK FOR THE STARSS AND MANUAL SYSTEMS, AND USING 8 SPES.

| System | Task size | | | | | | |
|---|---|---|---|---|---|---|---|
| | **1.92 $\mu s$** | **11.2 $\mu s$** | **19.1 $\mu s$** | **66.7 $\mu s$** | **241 $\mu s$** | **636 $\mu s$** | **2373 $\mu s$** |
| StarSS | 23 | 23 | 23 | 41 | 132 | 340 | 1250 |
| Manual | 6.0 | 10 | 13 | 39 | 130 | 337 | 1250 |

## VI. 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.

### A. Nexus System Overview

The Nexus system contains two types of hardware units (see Figure 8). The first and main unit, is the Task Pool Unit (TPU). It 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.

The second unit in the system is the Task Controller (TC), which can be placed at individual cores. The TC fetches tasks from the TPU, issues the DMAs of input and output operands, and enables double buffering of tasks. TCs are optional and platform specific. In future work we will investigate for the Cell processor whether the benefits of the TC are enough to be worth its costs.

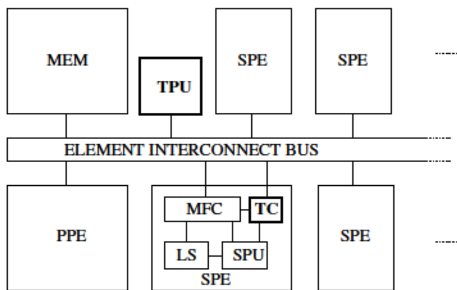

Figure 8. Overview of the Nexus system (in bold) implemented in the Cell processor. The depicted architecture could be one tile of a larger system.

Storing all tasks in a central unit provides low latency dependency resolution and scheduling. The hardware queues ensure fast synchronization. Such a centralized approach, however, eventually creates a scalability bottleneck for increasing core count. In such a case, a clustered approach can be used. Tasks are divided among TPUs such that inter TPU communication is low while task stealing allows system wide load balancing.

### B. Design of the Task Pool Unit

The block diagram of the Nexus TPU is depicted in Figure 9. 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.

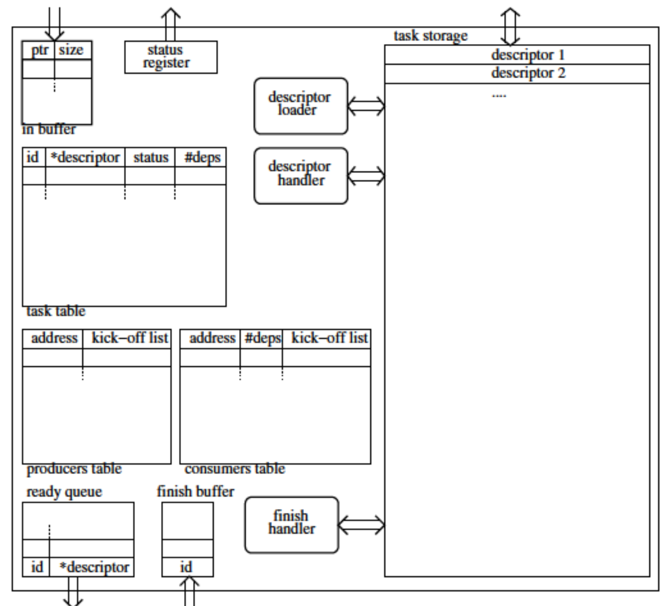

Figure 9. Block diagram of the Nexus TPU. The ports are memory mapped and can be accessed anywhere from the system.

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. This operation can be performed by either its Task Controller (TC) or by the software running on the SPU. 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.

## VII. Conclusions & Future Work

The task-based StarSS programming model is a promising approach to simplify parallel programming. In this paper we evaluated its runtime system. It was shown that for fine-grained task parallelism, the task management overhead prohibits scaling beyond five cores. More specifically, in order to efficiently utilize 8 or 16 cores for task based parallel H.264 decoding, the runtime system should be accelerated by at least a factor of 3.6 and 7.3, respectively. Mainly the task dependency resolution process is too laborious to perform in software. Other bottlenecks were found in the scheduling of tasks and the lack of fast synchronization primitives. We proposed the Nexus hardware task management system. It performs dependency resolution in hardware using simple table lookups. Hardware queues are used for fast scheduling and synchronization. We are currently implementing Nexus in a simulation environment to evaluate its performance and scalability.

## Acknowledgment

This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

## 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] S. Kumar, C. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *Proc. Int. Conf. on Computer Architecture*, 2007.

[3] J. Hoogerbrugge and A. Terechko, "A Multithreaded Multicore System for Embedded Media Processing," *Transactions on High-Performance Embedded Architectures and Compilers*, vol. 3, no. 2, 2008.

[4] S. Ou, T. Lin, X. Deng, Z. Zhuo, and C. Liu, "Multithreaded Coprocessor Interface for Multi-Core Multimedia SoC," in *Proc. Design Automation Conference*, 2008.

[5] G. Long, D. Fan, and J. Zhang, "Architectural Support for Cilk Computations on Many-Core Architectures," *ACM SIGPLAN Notices*, vol. 44, no. 4, 2009.

[6] M. Själander, A. Terechko, and M. Duranton, "A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures," in *Proc. Conf. on Digital System Design Architectures, Methods and Tools*, 2008.

[7] G. Al-Kadi and A. Terechko, "A Hardware Task Scheduler for Embedded Video Processing," in *Proc. High Performance Embedded Architectures and Compilers Conference*, 2009.

[8] J. Castrillon *et al.*, "Task Management in MPSoCs: an ASIP Approach," in *Proc. Int. Conf. on Computer-Aided Design*, 2009.

[9] Y. Etsion, A. Ramirez, R. Badia, and J. Labarta, "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*.

[10] "Cell Superscalar," http://www.bsc.es/plantillaG.php?cat_id=179.

[11] 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.

[12] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez, and A. Ramirez, "Parallel Scalability of Video Decoders," *Journal of Signal Processing Systems*, 2008.

[13] C. Chi, B. Juurlink, and C. Meenderinck, "Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine," in *Proc. Int. Conf. on Supercomputing (ICS)*, 2010.