Implementation of Nexus: Dynamic Hardware Management Support for Multicore Platforms

Efrén Fernández Abeledo

Abstract

Current trends in computer architecture focus on multicore platforms. The target of these new platforms is to scale the performance of the system with the number of cores. However, the performance of current architectures is limited due to thread-level parallelism overhead and programmability. StarSS is a task-based programming model that eases the programmability of multicores and tries to exploit functional parallelism within applications. However, the performance of StarSS does not scale efficiently for fine-grained tasks, as for such tasks the task management overhead becomes significant in comparison to the execution of the tasks. Nexus is a dynamic hardware support system that aims to alleviate the current overhead of StarSS, by offloading the dependency resolution process and the synchronization with the cores to hardware. In this work, we implement Nexus by defining and connecting the new hardware in a Cell architecture simulator. The scalability, performance, and throughput of the implementation are evaluated for different task sizes and number of cores, using several dependency patterns. Furthermore, different configuration parameters are evaluated, such as the dimension of the new hardware inserted in the existing architecture.

Results show a large improvement of the scalability offered by Nexus in comparison with StarSS, especially for fine-grained tasks. Nexus succeeds at alleviating the overhead of StarSS by accelerating the dependency resolution process and the synchronization with the worker cores. Furthermore, the evaluation of the Nexus system dimensions has shown that its scalability decreases slightly with its area.
Implementation of Nexus: Dynamic Hardware Management Support for Multicore Platforms

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Efrén Fernández Abeledo
born in Santiago de Compostela, Spain

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Implementation of Nexus: Dynamic Hardware Management Support for Multicore Platforms

by Efrén Fernández Abeledo

Abstract

Current trends in computer architecture focus on multicore platforms. The target of these new platforms is to scale the performance of the system with the number of cores. However, the performance of current architectures is limited due to thread-level parallelism overhead and programmability. StarSS is a task-based programming model that eases the programmability of multicores and tries to exploit functional parallelism within applications. However, the performance of StarSS does not scale efficiently for fine-grained tasks, as for such tasks the task management overhead becomes significant in comparison to the execution of the tasks. Nexus is a dynamic hardware support system that aims to alleviate the current overhead of StarSS, by offloading the dependency resolution process and the synchronization with the cores to hardware. In this work, we implement Nexus by defining and connecting the new hardware in a Cell architecture simulator. The scalability, performance, and throughput of the implementation are evaluated for different task sizes and number of cores, using several dependency patterns. Furthermore, different configuration parameters are evaluated, such as the dimension of the new hardware inserted in the existing architecture.

Results show a large improvement of the scalability offered by Nexus in comparison with StarSS, especially for fine-grained tasks. Nexus succeeds at alleviating the overhead of StarSS by accelerating the dependency resolution process and the synchronization with the worker cores. Furthermore, the evaluation of the Nexus system dimensions has shown that its scalability decreases slightly with its area.

Laboratory : Computer Engineering
Codename : CE-MS-2010-34

Committee Members :

Advisor: Arjan van Genderen, CE, TU Delft
Advisor: Cor Meenderinck, CE, TU Delft
Advisor: Ben Juurlink, TU Berlin
Chairperson: Koen Bertels, CE, TU Delft
Member: Hai Xiang Lin, TWI, TU Delft
Dedicated in loving memory of my grandfather Julio Fernández Malde. I know that, wherever you are now, you stand proud of me.
## Contents

### List of Figures  v
### List of Tables  ix
### Listings  xi
### Acknowledgements  xiii

1 Introduction  1

1.1 Related Work  2

2 Background  3

2.1 The Cell BE  3

2.2 The StarSS Programming Model  5

2.2.1 Overview  5

2.2.2 Runtime System Evaluation  7

2.3 Conclusions  10

3 Nexus: Concept Description  11

3.1 Nexus Design Goals  11

3.2 Design Overview  13

3.3 Dependency Resolution  15

3.4 Conclusions  21

4 Experimental Environment  23

4.1 UNISIM  23

4.2 CellSim  24

4.2.1 Fixes in CellSim  27

4.2.2 Block Attachment to the Existing Architecture  27

4.2.3 Simulator Validation  33

4.3 Conclusions  35

5 Nexus Implementation  37

5.1 Block Overview  37

5.2 The Task Life Cycle  44

5.3 Control Logic  47

5.3.1 Descriptor Loader  47

5.3.2 Descriptor Handler  49

5.3.3 Finish Handler  51

5.4 Register Interface  53
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.1</td>
<td>Read Register</td>
<td>53</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Write Register</td>
<td>54</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Task Storage Operations</td>
<td>54</td>
</tr>
<tr>
<td>5.5</td>
<td>Hash Function and Collision Resolution</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>TPU Types and Configuration</td>
<td>60</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusions</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>Experimental Results and Analysis</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>Benchmarks</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>Nexus Evaluation</td>
<td>68</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Scalability of the CD, SD, and ND Benchmarks</td>
<td>68</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Scalability for a Variable Task Size</td>
<td>75</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Performance Improvement by Adding the PPE to the Execution of Tasks</td>
<td>77</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Impact of TPU Latency in the Resulting Scalability</td>
<td>79</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Impact of Task Table Size in the Resulting Scalability</td>
<td>81</td>
</tr>
<tr>
<td>6.3</td>
<td>Conclusions</td>
<td>85</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and Future Work</td>
<td>87</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>87</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td><strong>Bibliography</strong></td>
<td>92</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 Architecture of CellBE. ........................................... 3
2.2 Overview of the SPE. ............................................. 4
2.3 Scalability of StarSS with the CD benchmark. ...................... 7
2.4 Scalability of StarSS with the SD benchmark. ...................... 8
2.5 Scalability of StarSS with the ND benchmark. ...................... 9

3.1 Nexus hardware in Cell processor. ................................ 11
3.2 Block diagram of the TPU. ...................................... 14
3.3 Dependency graph example. ...................................... 16
3.4 Dependency resolution process. ................................ 19

4.1 Communication in UNISIM. ..................................... 23
4.2 CellSim block diagram. .......................................... 25

5.1 Nexus implementation block overview. .......................... 38
5.2 The life cycle of task. ........................................... 44
5.3 Descriptor loader and its interactions within the TPU. ........... 48
5.4 Descriptor handler and its interactions within the TPU. .......... 49
5.5 Finish handler and its interactions within the TPU. .............. 51
5.6 Dependency table entry. ........................................ 57
5.7 Initial available-space list. .................................... 57
5.8 Insertion procedure of addresses from 1 to 3. .................... 58
5.9 Deletion of address 2 from the table. .......................... 59

6.1 Dependency patterns of the benchmarks. ........................ 67
6.2 Scalability of Nexus for the CD benchmark. ...................... 70
6.3 Scalability of Nexus for the SD benchmark. ...................... 72
6.4 Scalability of Nexus for the ND benchmark. ...................... 75
6.5 Scalability of Nexus for the CD benchmark and a variable task size. 76
6.6 Speedup obtained from the PPE executing tasks while not adding them. 78
6.7 Impact of the TPU latency in the scalability. ................... 80
6.8 Maximum attainable scalability using 16 cores. ................ 81
6.9 Impact of the task table size in the scalability. ................. 82
6.10 Execution example with 512 entries in task table and 8 parallel tasks. 83
6.11 Execution example with 512 entries in task table and 11 parallel tasks. 84
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Physical page range</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Physical page range with TPU</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Validated parameters</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Validation results</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>TPU register interface memory mapping</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>Average throughputs of life cycle stages in the CD benchmark</td>
<td>69</td>
</tr>
<tr>
<td>6.2</td>
<td>Average throughputs of life cycle stages in the SD benchmark</td>
<td>71</td>
</tr>
<tr>
<td>6.3</td>
<td>Average throughputs of life cycle stages in the ND benchmark</td>
<td>74</td>
</tr>
<tr>
<td>6.4</td>
<td>Speedup obtained from the PPE executing tasks while not adding them</td>
<td>78</td>
</tr>
</tbody>
</table>
First of all, I would like to thank my parents, for their unpayable love and support through my entire life. I couldn’t be where I am right now without them. I would also like to thank my godmother, for being an unbelievable support as both, part of my family and friend. I would like to thank my grandparents, who I admire for their continuous love and the example they have given to me of their unflagging effort. I would like thank the rest of the family: aunts, uncles, and cousins, thank you for being there. I would to thank my friends from my hometown Fene, you have proved to be real friends, regardless of the distance or the time. I would also like to thank all the friends I have left in my years in Vigo, thank you all for having traveled across that long journey with me. I would like to thank all my friends I have met in my last years in the Netherlands, thank you for making these last years amazing. Thank you all, life is much easier at your side.

I would like to thank also my supervisors Ben Juurlink and Cor Meenderinck, for giving me the chance to work with them, and for guiding and advising me through this thesis. I would like also to thank Arjan van Genderen for being my supervisor in the final steps of this thesis, and for being part of the committee. I would also like to thank Hai Xiang Lin for being part of the committee.

Efrén Fernández Abeledo
Delft, The Netherlands
October 26, 2010
In the past years, computer architecture has witnessed a shift towards multicores, as the previous techniques to increase the performance of the processors showed decreasing improvement. These previous techniques improved performance by exploiting Instruction-Level Parallelism (ILP) and using increasing processor frequencies. ILP exploitation, however, brought along large area and power consumption, and showed diminishing results. At the same time, power constraints have limited the increase of processor frequencies. As a result, industry has turned towards multicore platforms. According to Moore’s Law, the number of transistors on a die doubles every year, and the number of cores seems to follow the same law. Theoretically, performance scales with the number of cores and thus sustained performance improvements are expected.

Among the novel multicore platforms, a state-of-the-art processor is the Cell Broadband Engine (CBE). It is a heterogeneous multicore developed as a joint initiative from IBM, Sony, and Thosiba. The architecture of this platform combines a PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs), connected through the Element Interconnect Bus (EIB). The performance improvements that can be obtained by this type of multicore platforms are threatened by several factors, among which are communication overhead, Thread-Level Parallelism (TLP) overhead, and programmability. To enable performance scalability, these issues must be addressed.

To improve the programmability of multicores, several programming models have been proposed recently. A significant number of these are task-based, i.e., they try to exploit functional parallelism within applications. One of these task-based programming models that can be used in combination with the CBE is StarSS[21]. The StarSS programming model allows the programmer to indicate functions that can run in parallel, by annotating the serial code. The runtime system of StarSS dynamically determines the dependencies among tasks, based on their input and output operands. Tasks, whose dependencies are resolved, are scheduled to available cores for execution. The ease of programming, offered by StarSS, comes at the cost of overhead introduced by the runtime system. Although StarSS scales fairly well for large tasks and a small number of cores, its overhead limits the scalability for fine-grained tasks and a large number of cores.

To overcome the limitations of StarSS we use Nexus, an enhancement for task-based programming models, of which the design was proposed in [16]. Nexus is a hardware support system that aims to reduce the overhead introduced by the programming models. It is based on accelerating the task dependency resolution process and providing scalable synchronization between cores. In this thesis, Nexus is implemented in a CBE simulator. We evaluate its performance, scalability, and throughput using several benchmarks and different task sizes. Moreover, as different configuration parameters within Nexus may affect its performance and scalability, several cases are evaluated to study their impact.
CHAPTER 1. INTRODUCTION

The remainder of this thesis is organized as follows. In Chapter 2, background information on the CBE architecture and the StarSS programming model is given. Next, in Chapter 3 the design of Nexus is explained. Chapter 4 introduces the UNISIM simulator environment and the CBE simulator built on such environment, CellSim. Chapter 5 describes the implementation of the hardware block and the decisions made at the implementation time. In Chapter 6, the benchmark used for the scalability analysis is introduced. Next, the scalability results of the benchmarks are illustrated, as well as the impact of different internal latencies and sizes within the Nexus implementation. In Chapter 7, the conclusions are drawn, and the ideas for future work are presented.

1.1 Related Work

Several works have been done in the hardware support for task management. However, the majority of these works do not offer the dependency resolution among tasks, as they are limited to schedule available task for execution. Therefore, these approaches rely on the programmer to deliver the tasks properly. As the first example, Carbon[13] suggests a hardware tasks queuing that achieves low latencies at the retrieval of tasks. A more flexible scheduling of the tasks is achieved in ADM[22] through the use of inter-thread synchronization. Based on Carbon, in [11] it is proposed a TriMedia multicore platform containing an unit that provides centralized task scheduling. In a multicore media SoC, in [18] it is proposed a hardware interface to manage the tasks on the DSP in order to alleviate the OS. In [14], an architectural support focused on the cache coherency for the Cilk programming model it has been proposed.

For the dependency resolution among tasks, few are the works that have appeared. In [23] it has been proposed a look-ahead task management unit that reduces the latency at the retrieval of tasks. Such unit only determines the tasks that are ready to be executed, as dependencies among tasks must be defined from the user, instead of being automatically generated. To predict the ready tasks, the task dependencies are used in a reverse fashion. However, the latency of this system is rather large due to this system being programmable. This drawback has been tackled in [5], although it has lost generality as it is oriented to H.264 decoding. In SoCs, to speedup the software task management, an Application Specific Instruction set Processor (ASIP) has been proposed[7].

A hardware acceleration for task-based programming models has been proposed in [26], which is the most similar work to Nexus. Here, the similarity between input and output dependencies of tasks and instructions has been observed. Thus, their proposal is a hardware for dependency resolution, which working principle is similar to an out-of-order instruction scheduler.
2.1 The Cell BE

The Cell Broadband Engine (CBE) \cite{12} entails the joint effort of Sony, Toshiba, and IBM in building a multicore processor. It is based on a heterogeneous architecture with two types of cores, each one optimized for a specific role, and a shared and coherent memory scheme. The CBE has been designed to offer an outstanding performance on an ample variety of applications, with a special focus on computationally intense environments such as multimedia and gaming.

Figure 2.1: Architecture of CellBE.

The CBE, as illustrated from Figure 2.1, is composed of a 64-bit PowerPC core, 8 synergistic processors, on-chip controllers for memory, the Memory Interface Controller (MIC), and IO, and an Interconnection Bus (EIB) which serves as the communication interface among the different modules present in the platform. The PPE is a general purpose, dual-threaded processor with a RISC instruction set and with vector media extensions to support a SIMD organization, which has been demonstrated to be very efficient in speeding up multimedia applications. This processor has a cache hierarchy
CHAPTER 2. BACKGROUND

of two levels, as shown in Figure 2.1. In the first level we find two different caches, the first-level cache is composed of two elements with a size of 32-KB, one for data and another one for instructions, and a 512-KB second-level cache. PPE is responsible for running the operating system. The SPEs run autonomously, but do not run an operating system and are depending on the PPE for receiving the work they must perform.

Figure 2.2: Overview of the SPE.

The Synergistic Processing Elements (SPEs) [9] [11], depicted in Figure 2.2, are 128-bit RISC independent processors optimized for both area and power in order to carry out heavy workloads. They implement an Instruction Set Architecture (ISA) specific to the CBE. Their internal design consists on a SPU (Synergistic Processor Unit) containing a LS (Local Store), and a MFC (Memory Flow Controller) with an inside DMA controller. Instead of using a cache-based architecture, the SPU operates on the local store that serves as the main storage for the SPU. This is from where instructions are fetched and data is loaded and stored. It has a size of 256-KB, being the largest element of the SPE. To access data outside the SPE, the MFC is programmed to perform an asynchronous DMA. Using the DMA it is possible to reach main memory but also local stores of other SPEs. To program this DMA unit or to access any other unit in the CBE, each SPU has available a channel interface that serves for communication with the other entities. The MFC module can issue a sequence of DMA transfers writting the DMA-list command on the channel interface. The memory flow controller can process up to 16 transfers simultaneously, each of them with a size not bigger than 16KB of data. The status of the MFC can be checked by reading its MMIO registers. This check can be performed
from the PPE or any SPE. These registers are accessible from its own SPU through the channel interface and from the other modules as they are included in the common address space.

The Element Interconnect Bus (EIB) is the communication resource shared by all controllers (memory and IO) and processing element present of the architecture. This interconnecting bus supports full memory-coherent and Symmetric MultiProcessor (SMP) operations, allowing building coherent clusters of CBE processors. It is formed by four rings, each of them with a data with of 16 bytes and transferring 128 bytes at the same time, which equals the size of a cache line. Each of these rings can handle data transfers concurrently being able to issue more than 100 DMA transfers among main memory and SPEs, with a maximum internal bandwidth of 96 bytes per processor clock cycle. For the sake of speed, programmers must be aware of the fact that latency depends on the ID associated to each of the elements plugged to the EIB, as latency increases with the number of hops separating the two edges of the communication. In order to be able to address every module in the architecture, each block connected to the EIB is memory mapped to a certain physical address. Thus, different MFCs, LSs, the PPE or the main memory are mapped to different physical ranges. Accordingly, the EIB routes every piece of data depending on its targeted physical address.

The most innovative and groundbreaking feature of the design is the absence of a cache hierarchy on the SPEs commented above, and the presence of a local store and a DMA controller instead. The idea of DMA used to access main memory is employed to overcome the memory wall. In cache based architecture, data is picked up whenever it is needed, producing large delays because of the long distance from cores to main memory. In the DMA based approach, we must think in advance what is the data needed. But at the time this data transfer happens, also some processing can run concurrently and thus we can hide memory latencies. The concept of a local memory directly accessible from the processor and reachable from any other module, and the asynchronous DMA transfers enhances the predictability and scalability of the global design. In the cache based scheme it is possible that cache misses do impact on the processor performance, making the system not predictable. Another important characteristic of the platform is that every local store is mapped into the memory map of the entire processor, so any processing module is reachable from any other processing module present on the architecture. This fact allows carrying out local-store-to-local-store data transfers using DMA operations. However, this type of communication between two synergistic processors is not coherent, as the information contained in the local store may have changed without any warning towards the targeted local store.

2.2 The StarSS Programming Model

2.2.1 Overview

As current trends in microprocessors are maturing in the direction of placing many cores into the same chip, envisioning platforms that contain hundreds to thousands of cores, it is clear that programming models have to be developed. These models grow with the purpose of aiding programmers to build efficient implementations capable of profiting
from the existing parallelism on the applications. As an example of such tools, we find
the StarSS [21] programming model, with its particular implementation CellSS for the
Cell Broadband Engine. An example of such new programming model is StarSS, of which
an instantiation for the Cell processor exists.

StarSS is a programming model intended to exploit functional parallelism. It is not
strictly necessary for programmer to know on which platform the application is running.
Annotating the code by adding *pragmas* is sufficient. The main pragmas are shown in
Listing 2.1 although there are also some *pragmas* elements used for synchronization
issues. The css task *pragma* denotes a function that can be executed as a task in parallel
depending on the input and output operands. That will be dynamically checked by
the run-time system. Then it builds a task dependency graph at runtime, where tasks
that do not hold dependencies among them can be executed in parallel. The css start
and css finish annotations indicate the beginning and the end to this run-time system,
respectively.

The implementation of StarSS is executed on one master core running two threads
and several worker cores that perform the tasks of the application. The first thread
executes the main program code and adds tasks to the runtime system. The second
thread schedules the tasks to the worker cores and handles the communication with
them. In this work, the targeted processor is the Cell BE described above. Therefore
there is an easy match for the master core and the worker cores: the two management
threads will run on the PPE (master core) that will distribute the tasks to the SPEs
(worker cores). Consequently, the PPE will signal the SPEs whenever there are ready
tasks to be executed. Once they are aware of a new task, the worker cores will perform
a DMA request to fetch the data into their local store (handled by the run-time system)
and perform the task, after what they signal back to the PPE, indicating that they have
finished the task.

```c
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 2.1: Basics of the StarSS programming model.
2.2. THE STARSS PROGRAMMING MODEL

The StarSS runtime system has some configuration parameters to optimize the performance of the system, like the minimum number tasks to schedule to a processor, or the minimum number of tasks processed before start assigning tasks. As a consequence, StarSS offers the possibility of gathering tasks on the graphs into bundles aiming to exploit data locality. By this mean, it is possible to reduce the average overhead per task and a better scalability. On the drawbacks side of this programming model, there is the large overhead used for handling the ready tasks especially when they are characterized by small sizes. In the next subsection we show the experimental results with the most optimal configuration of the StarSS implementation for Cell BE processor, CellSS.

2.2.2 Runtime System Evaluation

The main results presented in [17] are repeated in this section as they show an evaluation of StarSS efficiency, concretely for its Cell implementation CellSS. In this document three synthetic benchmarks were tested for a variety of task sizes, ranging from approximately 2μs to slightly more than 2ms. Each of the tests was executed with a different dependency pattern over a matrix of 1024 × 1024 words divided in squared blocks of 16 × 16. Every block represents one task that must be run on the working cores, having a total number of 4096 tasks. Although the benchmarks will be described in detail later in Chapter 5, here we present their main idea to aid the understanding of the results obtained from these tests.

Figure 2.3: Scalability of StarSS with the CD benchmark using the optimal configuration.

First, the CD (Complex Dependencies) benchmark is introduced which embodies the most complex dependency pattern in the benchmarks studied, where each block depends
on its left and top-right neighbor block. The theoretical maximum attainable scalability for this benchmark is 14.5 when using 16 cores, due to the existing dependencies among the tasks. This is especially essential at the beginning and the end of the benchmark where little parallelism is available. Under these circumstances, several configurations were used in order to study the scalability and overhead of the StarSS programming model, resulting in the graph illustrated in Figure 2.3. For the default configuration a poor scalability was obtained even for very large task sizes, because the run-time system tries to group several tasks to send to the same worker core. This is beneficial only when an ample parallelism is encountered, but not in every case. Therefore, a new configuration was set to spread the tasks once they were ready, reaching in this case a top scalability value of 14.2. As a drawback, this new setup implies a larger overhead for the run-time system, resulting in worse responses when decreasing the task size. Although a new optimal configuration was achieved, results showed that for small tasks the main and helper threads are the bottleneck. For such small tasks the management threads cannot continuously provide all SPUs with tasks to execute. Taking for example the task duration of 19µs, which approximates the task length in H.264 decoding, a maximum scalability of 4.8 was obtained.

For the SD (Simple Dependencies) benchmark, one of the dependencies from the previous case is removed, and only the dependency on the left neighbor remains. Consequently we only have one dependency present and the matrix is completely row-independent as links among rows do no longer exist. The expectations for this, in comparison with the previous one, were to obtain a significant improvement on the results since the creation of bundles is now more efficient due to the simplified dependency pattern. However, the resulting graph, depicted in Figure 2.4, demonstrates only slight
improvement in comparison with the CD benchmark, since the scalability for the 19µs tasks is only 5.6. After analyzing a trace, the conclusion was that the main bottleneck was the main thread, responsible for adding tasks. This fact implies that the maintenance and construction of the task dependency graph is not strongly dependent on the pattern of the dependency itself. On the other hand, it seems that helper thread is significantly relieved by the possibility of gathering blocks into bundles, as the overhead introduced now is much lower. This experiment could lead us to the thought that helper thread is scalable enough, but this does not hold because of the round robin nature of its synchronization mechanism. This policy plays an important role when a SPE might have finished execution, because the helper will not pay attention to its signal until all its predecessors have finished as well, meaning a worst case scenario of 34.56µs for 16 SPEs, when for an individual the time needed is only 2.16µs.

ND benchmark is characterized for not having any dependency between any two tasks, so both of them are independent and can be run in parallel. As a result, it is shown in Figure 2.5 that now the scalability reaches a value of 15.8, much higher than in the previous two cases. The reason behind this change is that since no ramping effect is found and as soon as tasks are added, they are ready for execution. The existence of a higher parallelism is translated in a better response of the system for shorter tasks lengths. When looking at the trace, it has been noticed that the time necessary to build and maintain the task is very similar to the one encountered in the SD benchmark, casting the conclusion that the dependency pattern does not impact in the time consumed, but instead it is the dependency checking the main reason for the delay.

Figure 2.5: Scalability of StarSS with the ND benchmark.
2.3 Conclusions

The results from StarSS show that a good scalability is reached for large task sizes, when for small values its performance drops due to the overhead introduced by the programming model. For these short tasks, results from the CD, SD, and ND benchmarks have shown three bottlenecks in the behavior of the StarsSS runtime system. First, the dependency resolution procedure is the main bottleneck which cannot provide tasks quickly enough and thus causes the SPEs to stall. Second, scheduling and adding of tasks performed by the runtime system is also a bottleneck, as they are not able to meet the speed requirements. The final bottleneck is the synchronization of the SPEs with the helper thread that checks the working cores with a round robin policy. In order to cope with the effects of these bottlenecks, a new dynamic management support system has been designed. Such system, explained in detail in Chapter 3, aims to offload the workload from the PPE to a hardware block embedded in the CBE platform. A simple runtime system adds tasks to the hardware module, by sending it task descriptors. These contain, among other, the memory blocks used by the task. This dynamic management support system is responsible for checking the dependencies among the tasks and handling the synchronization with the worker cores using its memory-mapped registers. With this centralized solution, it is expected to reach a better scalability for smaller task sizes.
As it has been explained in the previous section, although StarSS obtains a good scalability for large tasks, its performance decreases for fine-grained tasks. The Nexus system emerges as a hardware support intended to alleviate the bottlenecks that limit this performance. In this section, the goals of the Nexus design are explained and its proposal to overcome them. Next, the block overview of the system is depicted, explaining in detail each of its components. In the last section, the dependency resolution is described.

3.1 Nexus Design Goals

![Figure 3.1: Nexus hardware in Cell processor.](image)

The Nexus system is proposed as a hardware support targeted at overcoming the bottlenecks found in the StarSS programming model. This system tries to enhance the limited scalability of StarSS for fine-grained tasks, by placing a new hardware block in the processor architecture. The three bottlenecks described before are handled as follows.

The first factor limiting the scalability was the dependency resolution procedure, whose dependency resolution is done according to the addresses of the input and output blocks of each task. Regardless of the dependency pattern, the system could not keep
up with the SPEs executing tasks. In [17], it is been calculated that it is necessary to speed up this procedure by a factor of 3.6 for 8 cores, and by a factor of 7.3 for 16 cores. A higher speed-up would be much appreciated, as it would yield the possibility of using more worker cores efficiently. The proposal of Nexus is to carry out this process entirely in hardware.

The second bottleneck found is the scheduling, preparation and submission of the tasks from the PPU. This is performed in StarSS by the helper thread, which tries to reduce overhead and exploit locality by delivering groups of tasks rather than individual tasks. This scheme is insufficient for small task sizes however, due to the large overhead introduced by the system. For this bottleneck, a speed up of 3.9 is required for 16 cores[17]. Hence, Nexus proposes a very simple implementation of the task scheduling, expecting to be fast taking advantage of such simplicity.

The last issue to be confronted is the synchronization of the PPE with the SPEs. As explain in the previous chapter, StarSS uses a round robin polling mechanism for synchronizing with the worker cores that does not scale well. The StarSS runtime system uses mailboxes to send tasks to the PPE and receive finished tasks from them. Due to this round robin polling, SPEs may be finished while the PPE is still checking the other cores. Therefore, Nexus proposes a centralized hardware queue where the worker cores can fetch tasks and signal them back once they are finished. This procedure is done independently by every processor, and it is required that such synchronization process does not take longer than 1µs to be scalable.

To fulfill these requirements, Nexus initial design contains two different blocks, the Task Pool Unit (TPU) and the Task Controller (TC). The TC would be placed inside the SPEs, being responsible for fetching tasks from the TPU and signaling them back. It would use double-buffering to hide DMA latency while the core executes a previously fetched task. As the TC would imply a large additional implementation effort, as it would have to deal with memory allocation within the worker cores, in this work we only focus on the TPU. For this reason, the TC implementation is left for future work. Hence, the current implementation of Nexus consists of a TPU that offloads the PPE of the dynamic task management. In Figure 3.1 the architecture of the IBM Cell processor with the task pool is depicted, where the new block is drawn as a blue box. The new block is connected to the shared EIB, where it is accessible from any other block of the system. It is intended to work as a centralized unit that receives tasks from the PPE, resolves their dependencies and distributes them over the worker cores. The main design concern of Nexus is to achieve a high throughput capable of satisfying the requirements of an increasing number of SPEs. For this reason, the TPU is highly pipelined, where pipeline one stage is performed concurrently with the other stages, as it is shown later in this section. The PPE adds new tasks to the TPU by sending the metadata describing the task, shaped as task descriptors. Task descriptors are structures which contain the required information to execute the tasks, as number of dependencies and the parameters needed to handle each of them.

The PPE sends the task descriptor pointer in the main memory and the size such task descriptor to the TPU. Then, the TPU loads the descriptor from the main memory into the task storage, its memory mapped scratchpad memory. This memory acts as the local store of the task pool and holds the descriptors of every task until it is finished,
3.2 DESIGN OVERVIEW

preventing the SPEs from going to main memory every time the task descriptor needs to be accessed, which would yield larger delays. After receiving the descriptor, the task pool determines the dependencies of the new task with tasks previously added. The dependency resolution is performed by three table lookups, filling them with the required information, which can be done very fast. In case the task is dependency-free, its id is written to the ready queue together with the task descriptor pointer to TPU task storage, where it can be read from the SPEs.

The ready queue is memory mapped and hence can be accessed quickly from anywhere in the platform. When a SPE is free to execute a task, it tries to read one from the ready queue. In case there is an available task, the SPE reads the descriptor from the TPU; otherwise it is stalled waiting for a task to be added to the ready queue. Once the SPE receives the descriptor, it loads the data necessary to perform the task from main memory into the LS and executes the task according to the function included inside the task descriptor. The next step is to signal back the completion of the task to the TPU. This is done by writing another memory mapped register, the finish buffer register. As this step involves another memory mapped register, it is completed in a few cycles taking advantage of the low latency of the EIB. The last stage is for the TPU to receive the finished task and update the tables by performing lookups again. Furthermore, it checks for tasks that do not have dependencies, adding them to the ready queue.

As it has been explained, this approach addresses the bottlenecks found in the StarSS programming model. Such a centralized approach may become a bottleneck, as the number of cores in the same chip increases. This can be solved by building clusters of TPU with low latency communication among TPUs. The load of the system can be balanced by allowing task stealing. Next two sections present the life cycle of a task, explaining the Nexus design more in detail by going through each of its pipeline stages, and how it performs dependency resolution.

3.2 Design Overview

In Figure 3.2 the internal view of the Nexus TPU is depicted. The block is composed by the memory mapped registers, the task storage, the tables containing the dependency information, and the functional modules that handle this information. The TPU is pipelined and thus, the descriptor loader, the ready queue, and the finish handler operate in parallel. The descriptor handler and the finish handler perform lookups in the task table, the producer table, and the consumers table, modifying their entries in accordance with the current status of the dependency graph. The in-buffer, the ready queue, the finish buffer, and the status register can be accessed from anywhere in the system in a few cycles, due to the low latency of the EIB. The task storage contains the task descriptors of the tasks being processed, and its entries are also memory mapped.

The task life cycle begins on the PPE, which fills the task descriptor with the information regarding its input and output operands. These operands are used later to determine the dependencies of the task with previously added tasks. The PPE sends the memory pointer to where the task descriptor is allocated and its size to the in-buffer. The in-buffer has several entries to allow the PPE to add tasks when the descriptor loader is still busy with a previous request or has stalled. Once a new task descriptor pointer is
received, the descriptor loader builds a memory request to the descriptor pointer in the main memory. When the new task descriptor arrives, it is stored in the task storage.

After storing the descriptor, the descriptor handler assigns an id to the task and determines the dependencies of the new task, by two lookups in the producers table and the consumers table. How this is done is detailed in the next section. Once dependencies have been determined, the descriptor handler fills the information regarding the task in the task table that contains the task id, the descriptor pointer within the task storage and its number of dependencies. In case no dependency exists, the task id and its descriptor pointer are added to the ready queue.

The SPEs read the ready tasks directly from the ready queue. In case there is no task awaiting to be processed, the SPEs are stalled until new ready tasks arrive. After fetching a new task, SPEs request the tasks descriptor from the task storage, which will send the task descriptor directly to them. The SPE will then request the data necessary for the execution of the task from the main memory, and starts executing the tasks according to task descriptor information. Once this is done, they signal back the completion of the task by writing the task id into the finish buffer. The finish buffer has also several entries to be able to store the finished task ids sent from several SPEs.
When the finish handler detects a task id in the finish buffer, it starts processing it. This handler, like the descriptor handler, checks the consumer and producer tables. Next, the finish handler updates the tables by removing dependencies with the finished task, as they no longer exist. The task table entries containing tasks with dependencies that have been removed are also updated. In case any of them is no longer dependent on any other task, the finish handler adds its task id and descriptor pointer to the ready queue. The last operation of the finish handler is to remove the task table entry containing the information of the task being processed, and thus allowing one more task to be added to the system. The status register stores valuable information regarding the current status of the TPU. This information contained in the status register indicates whether the TPU is running or empty, and if it is allowed to add a new entry to the in-buffer.

The TPU pipeline can be stalled. For instance, when the task table is full, the descriptor handler will stop inserting new entries in the table. This condition is signaled to the descriptor loader, which stops processing task descriptor pointers from the in-buffer. As the last effect, the PPE cannot keep adding tasks to the TPU due to the full in-buffer. PPE detects the in-buffer full by reading the status register, and hence it does not perform a new write into the in-buffer.

3.3 Dependency Resolution

Building and maintaining a dependency graph that represents the task dependencies can be very slow in software. Instead, the TPU handles the dependencies by simple lookups in the producer and consumer tables, which can performed very fast as they do not need to check the entire list. Every entry of these lists contains the address allocated there and a kick off list consisting of task ids waiting for the address to be released. In addition, the consumers list entries also have a field deps that holds the number of dependencies for that address. The fields of a task table entry describe the task id, the pointer to the descriptor, the current status of the task within the system and its number of current dependencies. When a task id is no longer encountered in any kick off list, its number of dependencies is 0, and thus it can be added to the ready queue.

The producers table is the mechanism used to avoid Read-After-Write (RAW) hazards. Each item in this table contains an address that is being written by some task previously added. Whenever the descriptor handler detects a task depending on one of the elements in this table, it immediately subscribes the task to its kick off list. That task will not be allowed to run until it is released from the list. In accordance to that, when a task is finished, the finish handler checks its kick-off list in the producers list. If it finds any element in the kick off list, the finish handler removes all of them and decreases their dependency count in the task table. This mechanism is not sufficient to avoid every hazard, as it cannot handle the case of two tasks writing to the same address. This case is known as a Write-After-Write (WAW) hazard. If a task writes to the same address written too by another task, adding it to the kick off list as described before does not solve the problem. In such a case, when the task producing the address finishes, it would kick off every element in the list regardless of them being writers or readers. If two writer tasks, or one writer and one or more readers were kicked off at
the same time, it would yield corrupted data. To deal with that situation, an additional mechanism must be implemented to distinguish the type of task found on the kick off list. This is done by adding the negative task id instead of the real task id for the tasks writing the data. In such a way, when the finish handler finds a negative id, it will stop removing elements from the kick off list. Later in this section we will show an example of how this works.

Figure 3.3: (a) Reading and writing tasks example to same address A and (b) its dependency graph.

The consumers table is used to avoid Write-After-Read (WAR) hazards. The elements of this table store the addresses that are being read from tasks previously added. When a new task is processed by the descriptor handler, it checks the output operands operands of the task. In case any of them has an entry in the table, the handler subscribes its task id to the kick-off list of this entry. As it happens with the producers
3.3. DEPENDENCY RESOLUTION

Receive task descriptor pointer from the descriptor loader
Fetch the associated task descriptor
for each input $I$ in the task descriptor
  if $I$ present in producers table
    append $I$ to kick-off list
  if $I$ present in the consumers table
    if kick-off list is empty
      Increase number of readers
    else
      Set a barrier as the negated number of readers
  else
    Write a new entry in the consumers table
for each output $O$ in the task descriptor
  if $O$ present in consumers table
    append $O$ to kick-off list
  if $O$ present in producers table
    Set a barrier as the negative task id
  else
    Write a new entry in the producers table

Listing 3.1: Descriptor handler pseudo-code.

list, when the task completes execution, the finish handler performs a lookup in the consumers table for each of its input operands. If the deps field is larger than one, it is decremented by one, indicating that one task is no longer reading it. If only one task was reading that address, the number of readers is now 0 and the finish handler must check the kick off list. In case there are tasks in this list, the finish handler removes them and decreases their number of dependencies in their task table entries. Like the previous case, this scheme is not able to deal with all possible cases. It might happen that after adding a writer to the kick off list, one or more new readers from the same address are added to the system. As the elements in the kick-off lists of this table are writers, only one of them can be removed as we cannot have multiple writers concurrently writing in the same position. On the other hand, multiple readers can read from the same address concurrently. Hence, we cannot add the readers as regular new items to the kick off list, as multiple readers are allowed to be removed from the list at the same time. Thus, a new barrier is added consisting of the number of readers awaiting for the address, but represented as a negative value to distinguish it from a regular task id. Accordingly, if $n$ readers are waiting for the address with no writer in between them, the barrier would be $-n$. Later we will present an example to illustrate this process. In such case, when the finish handler is processing a finished task, it checks the kick off list. In case it detects a negative value, it will remove the element from the kick-off list and write the number of readers in the deps field accordingly. Notice that for this particular situation, it is not needed to decrease any dependency count in the task table, as they are already set as dependent in the producers list. An example of how this exactly works is depicted in Figure 3.3 where all the different possibilities are shown.
CHAPTER 3. NEXUS: CONCEPT DESCRIPTION

18

Listing 3.2: Finish handler pseudo-code.

In the example shown in Figure 3.3, tasks 1, 3, and 6 perform a write operation and tasks 2, 4, and 5 perform a read operation on data block A. In Figure 3.3(b), the existing hazards due to task addition are detailed. As it is shown, in this example every possible hazard is presented. In Figure 3.4 it is illustrated how the three tables, consumers list, producers list, and task table, are updated at the arrival of new tasks. For simplicity, it is assumed that the execution of the first task is slower than adding the rest of the tasks, and that the TPU never stalls. Thus, when the first task finishes the other tasks have already been processed by the descriptor handler. The pseudo-code of the descriptor handler and the finish handler are shown in Listing 3.1 and in Listing 3.2. We will go through every step and explain the addition and removal of tasks to or from the kick off lists.

The first task added is a writer and hence a data producer, since it has block A as its output. Once it is loaded into the task table, the descriptor handler checks whether A is present in the producers table without any success. Thus, it writes a new element in the
Figure 3.4: Dependency resolution process.

table and its dependency count remains 0. The next step is to check the consumers table, which does not have an entry for this address either. This time no entry is allocated in the consumers table as the current task is not consuming the data. Due to a zero dependency count, it is added to the ready queue to be executed. At arrival of the second task, the descriptor handler detects its dependency on block A in the producers list, and thus this task is added to the kick off list. Notice that this is the case where it can be kicked off together with other readers. The next step is to add a new entry in the consumers table with deps set to 1, indicating that one reader is locking the address. This field of the consumers table entries contains the number of readers currently accessing to the address of the entry. In this case, task 2 dependency count is 1, as it needs to
read the address task 1 is writing to.

Next, the writer task (task 3) arrives and, as it is a produce, it should add an entry to the producers list. However, there is an already existing entry for this address. Therefore, we must split the list with a barrier to indicate the presence of a second writer. This barrier is the negated task id. It also finds an existing entry in the consumers table, so its task id is also added here to the kick-off list, but now in a positive representation. The dependency count this task 3 is set to 2, as it depends on one address in the consumers table and one address in the producers table. When reader task 4 arrives, it must be added to the kick-off list of the producers table as it happened with task 2, being placed after the barrier. Now, the singular case is found for the consumers list, as this task must add a new item in the consumers list, but it finds that the element already exists. Thus, a -1 is written as a barrier to indicate now that there is one reader waiting for the address. Although new elements have been written into both kick off lists, notice that the dependency count is 1 this time. The case of task 5 is very similar to previously added task 4, as they both try to read the data block. In accordance, id 5 is added to the producers table kick off list right behind id 4. For the consumers table, instead of adding a new barrier, here the -1 barrier becomes a -2 barrier, indicating two readers waiting for the address. The dependency count is as was done for the previous task, set to 1. The last task seen in the example is the writer task 6, which case is the same as task 3, adding its negated task id in the producers kick off list entry, and its positive value in the consumers table after the -2 barrier. In its task table entry it has a dependency count of 2 as it happened for task 3.

Let us now see how the finalization of tasks affects the tables. According to the initial assumption, when task 1 finishes, the other tasks have been already added to the system. After each task finishes, and the finish handler has processed the dependencies, the task table entry associated with the finished task is removed. The finish handler processes task 1 by checking the producers table, as this task is a writer. It removes tasks from the kick off list until and including the barrier. In this case, tasks 2 and 3 will be removed. Furthermore, the finish handler will decrement the dependency count of the task table entries of both tasks, becoming 0 for task 2 and 1 for task 3. Notice that task 3 depends on task 2, as it is set in the consumers table. As task 2 has no dependencies left, it will be added to the ready queue and executed. Once it has completed the execution and since it is a reader, this time the consumers table is processed. The finish handler decreases the number of readers contained in the field nr deprs of the address read by task 2, which reaches 0 and thus, it must kick off elements from the list one by one. The reason they are removed one at a time is that elements here are writers and cannot write to the same address at the same time. The element removed now is task 3, which no longer has a positive dependency count and thus is moved to the ready queue. Since next element happens to be a barrier with the value -2, this element is removed as well and a value 2 is written to the nr deps field for this entry. Notice that readers 4 and 5 depend on task 3 in the consumers table and therefore they are not allowed to be executed. After task 3 finishes, since it is a writer, it must check the producers table, taking out every element until and including the barrier. This will produce tasks 4 and 5 to reach 0 dependencies and be moved to the ready queue at the same time. Since they do not modify the data in the block, they can run concurrently. As it occurred with task 3, task 6 is also removed.
but it has 1 more dependency in the consumers table, so it cannot run yet. Assuming task 5 finishes first, it will process the consumers table entry of address A, decreasing the number of readers from that address. Contrary to the previous case, it does not reach 0 so the handler does not remove any element from the kick off list until task 4 finishes. When this happens, the finish handler removes task id 6 from the list and decreases its dependency count in the task table. Since it has no more dependencies, task 6 is added to the ready queue and processed. At this time, no task is holding the entry A in the consumers table and hence it is removed. After execution of task 6, the finish handler cannot find any element depending on task 6 in the producers table and thus this entry is removed.

3.4 Conclusions

Nexus dependency resolution scheme described in this chapter tries to overcome the StarSS programming model bottlenecks. This is done by simple table lookups that resolve dynamically the tasks dependencies and determine tasks available for execution. In order to do that, it uses several concurrent pipeline stages as the handlers, the descriptor loader, or the register interface that performs the reads and writes from or to the memory mapped registers. These registers, which can be accessed from any block in the system, are used to build an efficient synchronization procedure, and should perform the read and write operations in a few cycles. To study the impact of the Nexus block on a heterogeneous multicore platform, the goal of this work is to simulate its behavior and compare against the software of CellSS. The next chapter explains the details of the chosen simulation framework UNISIM, and its current implementation of the Cell BE, named CellSim. In CellSim, Nexus is integrated and its functionality is simulated, after which the results will be obtained.
4.1 UNISIM

Several simulation environments have appeared last years trying to model multicore platforms with the purpose of letting researchers gain knowledge on their behavior [6] [15]. Nevertheless, the big majority of them lack generality, as they are designed for specific processor architectures. This is a hindrance for evolution due to the big complexity of porting one module used in one tool to another one, making the comparison among different chip designs cumbersome. Furthermore, most of the available simulation frameworks are proprietary, so besides being difficult to carry out, experiments are expensive as well. UNISIM [4] was born to overcome all these necessities by offering a modular and free simulator. This is the tool used to simulate the hardware support described in the previous section.

UNISIM is a structural simulation framework that divides the architecture in different blocks, each of them corresponding to one of the elements of the architecture. Its main characteristic is that it maps the hardware directly to the simulator, making the implementation very intuitive, besides improving its readability and simplicity. In addition, the simulator enables the re-usability of blocks and the interoperability among different environments, by using a solution pioneered by LSE [24] and the MicroLib environment [20]. This is accomplished by wrapping the simulators into UNISIM modules with a .sim extension, whereas links among these blocks composing the design are established in the .uni file. Such feature is achieved also by the definition of a rigorous and standardized communication interface between modules. The interface decentralizes the control logic, establishing instead a distributed control among the blocks that eases the allocation of a new module in the architecture, or modification of an existing unit. Furthermore, there

![Communication in UNISIM](image-url)

Figure 4.1: Communication in UNISIM.
is a wide set of full-system functional simulators available capable of booting an entire Operating System (OS), although in our case of study only a small piece of them will be used. UNISIM also allows including hooks in the modules to call the capabilities of the system, where we can find power modeling, sampling or debugging among others. As this is a free simulation environment, a repository with components and complete models is at disposal to shorten the research activity and raise cost savings.

There exist two different methodologies for building a simulator, depending on the purpose of the design. When the interest is mainly focused on the functional behavior of the system rather than measuring the performance, TLM (Transaction-Level Modeling) methodology is the best choice. It provides faster simulation speeds with the drawback of not having timing involved, so delays will be ignored. On the other hand, when system performance is the main concern, the CLM (Cycle-Level Modeling) methodology is the most suitable option since it does involve time issues in the simulation. Nonetheless, it incurs in larger simulation times, as every signal delay is taken into account. We will focus a bit more on this model and its correspondent communication scheme.

Communication interface in CLM takes the form of a simple three-signal handshake between two different modules as can be seen in Figure 4.1. In order to plug in a module, we must have defined this signal set present in the in- and out- ports. When the first block A needs to send something to any other block B, it first starts delivering the data through the datapath connecting them. Once the destination has received it, it can either accept or refuse the data, using the accept signal. Nevertheless, module B does not process the new values until the enable signal is asserted, as to prevent the acceptance of data previously handled. This last mechanism is the part of the control that serves as synchronization when a source is delivering data for several destinations. In order to process those signals, UNISIM enables the definition of sensitive methods. These are called whenever one of these signals changes.

4.2 CellSim

CellSim, developed by Barcelona Supercomputing Center (BSC), is a modular simulator intended to prototype heterogeneous multiprocessors built on the UNISIM framework. The objective of the system is to test the performance of different architecture schemes. This is accomplished through its high modularity, choosing the CLM methodology offered by UNISIM for the performance study. Even though it can be tuned to research different heterogeneous processor architectures, its first purpose is to model the IBM Cell processor. Here it is where our hardware support will be attached to evaluate its impact. Within the Cell implementation, some parameters can be adjusted in order to investigate the effect of such changes in processor’s behavior. Every hardware block of the Cell BE architecture has its own representation on the simulator. They are shaped as C++ classes and wrapped into UNISIM modules with a .sim file extension, following the requirements of the simulation framework. As it was shown before, these components must have the aforementioned communication signals data, accept and enable, as well as methods sensitive to them. In addition, according to the CLM methodology, it is also required that all modules share a common clock signal with an associated sensitive method, used by the simulator to perform the functionality of every module related to
the current cycle. The interconnection of signals among the modules is defined in the .uni.cxx file instead of the original .uni format, and the synchronization of the blocks is the responsibility of UNISIM. It handles the method calls when signals change. Although CellSim has been validated before to match the real processor, we have carried out our own validation as some changes in thread management have taken place as it will be explained later on this chapter.

Although CellSim closely matches the real processor, there are few differences that must be mentioned. First, the memory module is used to model both main memory and the different LSs using the same memory class, as seen in Figure 4.2. This is achieved due to the property that the access latency and the port count of each module can be configured to act as the SPEs Local Storage. The second difference affects the cache hierarchy. In contrast to the real hardware, which contains two cache levels, in the simulator only one cache level has been implemented. Third, the pipelines of the PPU and the SPU are modeled at a functional level instead of a cycle level. Instructions are not executed based on the issue-width and the availability of functional units. Instead, a parameter can be modified to determine a static instruction fetch rate. Fourth, although the simulated PPE can run multiple threads similar to the real PPE, this model is not very accurate. The imprecision is in the scheduling of the threads, which are scheduled in a TDM fashion. Therefore, they are all assigned an equal processor time slice regardless of these threads being idle or not. In the real processor instead, only active threads are given a processor time slice. The EIB has been replaced by the Interconnection Network (IN), composed by a K-Bus topology, as an attempt to utilize a more scalable approach. The original interconnection is very efficient but it is not very scalable. Although they
Table 4.1: Physical page range.

<table>
<thead>
<tr>
<th>Physical page range</th>
<th>Module</th>
<th>IN Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x0000 - 0x4000)</td>
<td>Memory</td>
<td>0</td>
</tr>
<tr>
<td>(0x4000 - 0x4100)</td>
<td>LS0</td>
<td>NUM_PPEs+1</td>
</tr>
<tr>
<td>(0x4100 - 0x4200)</td>
<td>LS1</td>
<td>NUM_PPEs+2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(0xB000 - 0xB001)</td>
<td>PPE0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(0xC000 - 0xC001)</td>
<td>MFC0</td>
<td>NUM_PPEs+1</td>
</tr>
<tr>
<td>(0xC001 - 0xC002)</td>
<td>MFC1</td>
<td>NUM_PPEs+2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

are different, CellSim parameters allow us to match the performance of the IN with the EIB.

For the communication interface within the simulator, a shared scheme has been chosen in order to connect every module to the IN, in the form of the MemoryAccess class. Every input and output module port contains such MemoryAccess class as the data signal, together with the accept and enable signals used in the handshake. Inside this data container some valuable information is included, regarding the entities involved in the communication. Thus, within this information, there are fields that determine whether the packet is a load or a store, and the source and destination addresses. In case it is a store, the data transmitted is also included on the packet. Every module of the processor is mapped to a physical address range, as seen in Section 2. Therefore, CellSim also needs to perform the memory mapping of its modules to different address ranges in the global address space. The CellSim memory mapping allows us to connect a variable number of processing blocks, both SPEs and PPEs, and a memory no bigger than 1GB, seen in Table 4.1. The variable NUM_PPEs denotes the number of PPEs present in the current design, which in this case is 1, as only one PPE is connected. For the LS and MFC cases, the indexes attached to their names indicate the element number. Thus, for these two cases, the elements with the same number attached to the module name belong to the same SPE. For instance, LS1 and MFC1 are both inside the same SPE.

This flexible mapping is intended to investigate different architectures. Such flexibility is accomplished by the definition of address ranges for each group of elements. Hence, PPE addresses begin at $0 \times B0000000$ and finish at $0 \times C0000000$, where the MFCs range begins. Notice that only the two most significant bytes of the 32-bit addresses are written (the two least significant bytes are $0 \times 0000$). The first column of the table sets the physical memory mapped to the module found in the second column in the same row. The last column indicates the interconnection network port associated to the module of the same row. Each of those EIB ports consists on an input port and an output port. When a module needs to sent data to any other module, it sends the data with the target address, according to the table, to the EIB. The EIB receives the data from one of it input ports. Once it is received, the interconnection will send the
4.2. CELLSIM

MemoryAccess package towards the block containing the targeted address through the output port assigned on this table. Since a new hardware block is being defined, in this thesis a new physical memory range will have to be reserved in order to be able to address our hardware support, and so a new item would be included in the table taken from [8].

4.2.1 Fixes in CellSim

When the validation process was being carried out, it was noticed that DMA transfers did not match the real processor in terms of latency. As DMA latency is a parameter used by the simulator to model these DMA transfers, different DMA latencies were run on the simulator in order to match the real performance. DMA test results always showed smaller latencies than the real platform even though large latencies were tested, without any meaningful improvement. Therefore, the code belonging to the MFC was checked in order to understand the reason behind this behavior.

Listing 4.1: Changed lines in \texttt{mfc.sim}.

```
1  // if (_current_command->getType() == DMACommand::PUT)
2  _current_cmd_delay = dmacmd_delay;
3  // else
4  // _current_cmd_delay = 0;
```

The result of the code analysis was that the DMA latency parameter was not assigned to some type of DMA commands. The two main DMA commands are \texttt{PUT}, used to deliver the data to any destination; and \texttt{GET}, used for requesting the data from any source. The original code distinguished when to apply the latency depending on the command type, using for that purpose an if statement. Concretely, DMA latencies were disallowed for commands different than the \texttt{PUT} type. To solve that, those lines were commented, as showed in Listing 4.1 and a more accurate model for the time spent on a DMA was achieved.

4.2.2 Block Attachment to the Existing Architecture

As explained above, this work studies the performance impact of a new hardware block that relieves the PPE of the bottleneck found in CellSS study. Therefore, the definition of such new hardware and its connection to the interconnection network are needed. As we are working in the UNISIM environment, its implementation must follow the rules of the simulator. According to Table 4.1, it is also necessary to assign a physical page range so it can be addressed from other modules: the PPE and the SPEs. The main core adds the tasks to the task pool, and the worker cores retrieve ready tasks to be executed, signaling them back once they have been completed. In the following paragraphs a step by step procedure is detailed and it can be used as a guideline for future work.
template< int TPU_PORTS= 2 >
class TPU : public module
{
public:
/* Port declaration */
inport<MemoryAccess_bus> in [TPU_PORTS];
outport<MemoryAccess_bus> out [TPU_PORTS];
in clock clock;
/* Create a new TPU */
TPU(char* name) : module(name)
{
    /* Naming the interface to enhance debugging */
    class_name = "TPU";
    for(int i=0; i<TPU_PORTS; i++)
    {
        in [i].set_unisim_name(this,"in",i);
        out [i].set_unisim_name(this,"out",i);
    }
/* Sensitive methods */
sensitive_pos_method(start_of_cycle) << clock;
sensitive_neg_method(end_of_cycle) << clock;
sensitive_method(on_enable_0) << in [0].enable;
sensitive_method(on_enable_1) << in [1].enable;
for(unsigned int i = 0;i<TPU_PORTS;i++)
{
    sensitive_method(on_data) << in [i].data;
    sensitive_method(on_accept) << out [i].accept;
}
void init()
{
}
void start_of_cycle()
{
}
void end_of_cycle()
{
}
void on_data()
{
}
void on_enable_0()
{
}
void on_enable_1()
{
}
void on_accept()
{

Listing 4.2: Initial skeleton of TaskPool.sim.
The first step was the definition of an empty box compliant to the UNISIM rules. Similar the rest of the elements of CellSim, the task pool is written as a C++ class, wrapped into an UNISIM module with a .sim file extension. The new file was created and named TaskPool.sim, and allocated on the processor folder, as it is where all the modules are located with the exception of those forming the SPEs (accelerators). The next step was to define the input and output ports containing the data signal MemoryAccess_bus, which is nothing but an instantiation of the MemoryAccess class described before. Within this port definition, the handshake signals enable and accept are also included to carry out the data transfer with the rest of the entities present on the simulator. This is done only by the used of UNISIM reserved words inport and outport that we can see in Listing 4.2. Notice that also the common clock signal mentioned in the previous section has been added. In accordance with Nexus design, there are two input and two output ports, one for the MMIO registers and one for the task storage, being independent of each other. Once signals have been defined, it is required to assign each input signal to a sensitive method, which is called whenever a signal change is detected. Theses methods are:

- **start_of_cycle**, for the rising edge of the clock signal; and **end_of_cycle**, for its falling edge. The main functionality of the componet will be distributed over these methods.

- **on_enable_0** and **on_enable_1**, for the enable signals of ports 0 and 1 respectively.

- **on_data** and **on_accept**, for any port data and accept signals. Both data signals were grouped in the same sensitive method, as they performed the same operations, but only distinguishing the target of such operations. Similar was the case of the accept signals.

```plaintext
/*--------------------------------------TPUs--------------------------------------*/
// Definition of number of TPUs present in the architecture and
// number of ports of each TPU
#define NUM_TPUs 1
const int TPU_PORTS = 2;

Listing 4.3: New lines in default_configuration_parameters.h.
```

The init() function is used to initialize the block at the beginning of the simulation. Such initialization is called from the file global_definitions.cxx, where a new line s.tpu->init(); is added to the function cellsim_init(GeneratedSimulator& s). This function starts all the blocks contained in the simulator similar to a global reset signal, with s representing the entire simulator. Each element of this record is an element of the architecture. The last file involved in the task pool definition is the file default_configuration_parameters.h, which contains the default configuration of the blocks. Here, two lines were added: the
number of TPUs in the design and the number of ports present on each of them. The number of TPUs is left for further study on clusters of task pools. Both lines can be seen in Listing 4.3.

```cpp
#include "processor/TaskPool.sim"

typed EIB<NUM_PPEs+NUM_SPEs+2*NUM_TPUs+1> MyEIB; // This line has been changed due to the definition of a new module TPU
typed EIB<NUM_PPEs+NUM_SPEs+1> MyEIB;
typed TPU<TPU_PORTS> MyTPU; // This line has been added to connect TPU

MyTPU* tpu; // TPU definition

/* This block has been added to wire up the TPU to the EIB*/
/*------------------------------------------------------*/
sprintf(module_name,"TPU");
tpu = new MyTPU(module_name);
tpu->clock(global_clock);

/* Block added to connect the TPU to the Interconnection Network*/
/*------------------------------------------------------*/
// Here we connect both port 0 and 1 of TPU to their corresponding spot in the EIB
for (int id=0; id<TPU_PORTS; id++)
{
    tpu->out[id] >> eib->in_port[NUM_PPEs + NUM_SPEs + 1 +id ];
    eib->out_port[NUM_PPEs + NUM_SPEs + 1 +id ] >> tpu->in[id];
}

delete tpu; // This line has been added to delete tpu when the rest of the block in the architecture are deleted
```

Listing 4.4: New lines in cellsim.uni.cxx.

Once the block has been defined as a component of the architecture, the next step is to build its link with the interconnection network. In CellSiM this must be done in the cellsim.uni.cxx file, where classes containing the modules are called and linked. Therefore, in this file several changes, detailed in Listing 4.4 have been made in order to call the new block and link it with the existing design. The first line added was the include of the file containing the TaskPool class, located in the processor folder. As the new block with two ports is plugged to the EIB, the port count of the interconnection is now raised by two for each TPU in the design, so this parameter is changed. The
next two lines added correspond to the instantiation of the new module, indicating the number of ports defined in default_configuration_parameters.h. The following three lines are to name the module for debugging, to connect it to the shared global clock, and for the class instantiation. Since we had already defined the block and connected to the clock, we had to truly build the wires with the network. This is done in the loop, where each output port of the TPU is wired to its associated input port of the EIB, doing the same for the input ports of the TPU with its correspondent output port in the EIB. This establishes a physical link between our new module and the rest of the architecture. The last line means to remove the class once the simulator is finished to free memory space. However, besides physical links, it is also needed to set a memory range to be able to address it, as it is explained in the next paragraph.

Since the new module has to be reached from other modules, it needs to have a valid range within the shared address space. Looking at Table 4.1, it is visible that addresses below 0xD0000000 are occupied by the main memory, the PPE, and the SPEs. Even though for the original configuration of CellSim no more than 16 SPEs and 1 PPEs are expected, it has been avoided to use intermediate range for the sake of future work. Thus, the available range for new components ranges from 0xD0000000 to 0xFFFFFFFF. The decision made was to start from the first available address, as it was the simplest solution and there was no reason to pick another one. The size of the task pool is intended not to be larger than 1MB, thus the final address is 0xD0100000. Notice that for future work, when more than one TPU might be present, they could be placed immediately after the previous one, in a consecutive fashion similar to the SPEs or the PPEs. The TPU module has two ports, one for the register interface and the other for the task storage, and consequently two different address sets must be assigned. Task storage block is much larger than register interface block, as it has to hold the tasks descriptors of every task contained on the pool. The register interface holds queues of elements that are no longer 2 integers (4-byte each) per each item on the queue, a buffer and the status register. In addition, this register interface also has the control logic involved in the synchronization with the rest of the modules and the task management. Accordingly, it was decided to set the register interface range as 128KB (0x20000), which leaves enough room for the queues, the registers, and the control logic. Task storage range was set to 896KB (0xE0000), as it is where all the information regarding the tasks is loaded. This address range should be also enough, as no more than few thousands of task descriptors are intended to be stored in the TPU. As a consequence, Table 4.1 is modified resulting in Table 4.2.

Being already defined the address range; it has to be added in the routing tables used by the interconnection network to determine the destination of the data. The routing table is given as a parameter when the CellSim simulator is called. Such file is obtained calling the program generateRT.cxx located in the routing tables folder. When executing this program, we are asked about the architecture of the system, this is the number of PPEs, SPEs, memories, and, through a small modification in the code, the number of TPUs. The file produced has information about the initial- and end- address of each block, the port associated to each one of them and the existence of a cache, which is not the case for the task pool.
At this point, the new component has a physical link with the interconnection network and a valid physical address range assigned. However, it is not possible to reach the TPU from any module yet. The reason is that has a physical address, but not a virtual address, and the PPU and SPUs operate on virtual addresses instead physical. Accordingly, the application running on the processor must build the relationship between real address and physical address. This establishment of a link between the virtual address and the physical address is responsibility of the operating system running on the PPE. Thus, it is done by calling the function `init_tpu`, which returns the virtual address assigned to the TPU. This function performs two system calls. The first system call was `open`, which opens a file descriptor pointing to the TPU given the name of the module as a parameter, in this case ‘taskpool’. The value returned by this function is used to relate the physical address to the virtual address. However, this cannot be done with the initial system call set. The file `open.cpp`, where the functionality of the `open` system call is implemented, does not include the option of creating a file descriptor for a task pool, and thus it has been added. Within these lines, it calls the `FileDescriptor` class to open the file descriptor, using for that the `FrameMap` class. This class is used to request the physical address associated the block as well as its size. According to this, it is also necessary to change `FrameMap`, writing the function that returns the beginning of the physical address range, and the size of the block.

Once the file descriptor has been opened, next step is to perform the physical memory mapping, performing the `mmap` system call. This function receives as parameters the file descriptor pointer and the size of the virtual address requested. This system call requests a free virtual address range given the size value, returning the first virtual address of such range. Within this function, the virtual address is assigned to the physical address. The value returned by last system call is used by the system to refer the TPU and needs to be sent to the SPEs.

<table>
<thead>
<tr>
<th>Physical page range</th>
<th>Module</th>
<th>IN Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 - 0x4000</td>
<td>Memory</td>
<td>0</td>
</tr>
<tr>
<td>0x4000 - 0x4100</td>
<td>LS0</td>
<td>NUM_PPEs+1</td>
</tr>
<tr>
<td>0x4100 - 0x4200</td>
<td>LS1</td>
<td>NUM_PPEs+2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xB000 - 0xB001</td>
<td>PPE0</td>
<td>1</td>
</tr>
<tr>
<td>0xC000 - 0xC001</td>
<td>MFC0</td>
<td>NUM_PPEs+1</td>
</tr>
<tr>
<td>0xC001 - 0xC002</td>
<td>MFC1</td>
<td>NUM_PPEs+2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xD000 - 0xD002</td>
<td>RegisterTPU0</td>
<td>NUM_PPEs + NUM_SPEs + 1</td>
</tr>
<tr>
<td>0xD002 - 0xD010</td>
<td>StorageTPU0</td>
<td>NUM_PPEs + NUM_SPEs + 2</td>
</tr>
</tbody>
</table>

Table 4.2: Physical page range with TPU.
4.2. CELLSIM

4.2.3 Simulator Validation

When running a simulation, it is essential to have the simulator validated in order to obtain reliable results. It enables us to compare our results with the targeted platform, as it implies an accurate approach to the real hardware. In this case, CellSim has been validated against the Cell Blade located in the Barcelona Supercomputing Center (BSC). Even though the Cell Blade has 16 SPEs available and CellSim can simulate also 16 SPEs, only 8 are used for this validation. The reason is that both systems differ in terms of architecture, as Cell Blade has two different chips containing each 8 SPEs and inter-chip communication is needed. Instead, the CellSim interconnection communicates the 16 cores without any extra inter-chip communication. The other major difference is the absence of a second level cache in the PPE of the simulator. This fact produces a mismatch between delays for level 1 cache misses in the real processor and the simulator. While in the real processor the PPU tries to obtain the data from the second level cache, in the simulator it fetches the data directly from main memory. Accessing directly to main memory incurs in larger delays than to the second level cache and hence, this effect needs to be balanced. In order to do that, the memory latency has been reduced as an attempt to model the average delay produced by those first level cache misses. However, reducing memory latency also has an influence on the memory latency concerning the SPEs. As SPEs memory accesses are all DMA transfers, DMA latency has been increased to compensate the effect of a shorter memory delay.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem-latency</td>
<td>45</td>
</tr>
<tr>
<td>cl-latency</td>
<td>11</td>
</tr>
<tr>
<td>ppu-issue-width</td>
<td>1.4</td>
</tr>
<tr>
<td>spu-issue-width</td>
<td>1.8</td>
</tr>
<tr>
<td>eib-bus-latency</td>
<td>8</td>
</tr>
<tr>
<td>eib-bus-bw</td>
<td>8</td>
</tr>
<tr>
<td>mfc-dmacmd-latency</td>
<td>2600</td>
</tr>
</tbody>
</table>

Table 4.3: Validated parameters.

The parameters used for the validation of the simulator are shown in Table 4.3. The three values at the top are related to the PPE, as they are the latency for each access to main memory, the level 1 cache latency and the issue width of the PPU. The next parameter in the table is the spu-issue-width, which modifies the instruction fetch rate of the SPUs but not the issue width. Nonetheless, even though these parameters do modify the IPC of the cores, this is not equal to the real value. Real IPC is measured from the execution and it depends on the code run in the core. This situation is due to a simulator inaccuracy in the pipeline of the cores. In the real processor, within the cores pipeline two different stages are distinguished, the even- and odd- pipeline. However, this is not implemented in the simulator, where a number of instructions are fetched and executed, regardless of the pipeline stage the instruction belongs to. In this manner, by setting this parameter to two, more instructions than in the real processor would be executed.
Thus, it is necessary to lower this parameter and fewer instructions are executed as an attempt to match the average IPC of the core. Then next two are EIB parameters, as they are its correspondent latency and bandwidth. The last item of the table represents the delay of DMA transfers.

To obtain such set of configuration parameters, several tests were run on both the Cell Blade in BSC and CellSim, executing the same code. These tests have measured several phases of a normal execution, trying to isolate the effect of each of them to obtain an accurate approach. The first four experiments were useful to estimate a valid range for every configuration parameter, that later were refined in the last two experiments. The procedure followed was to go step by step over the following tests:

- **Task PPE.** This measurement tried to tune the PPE issue width and the level 1 cache latency, by executing only one task located on the cache. Memory. This benchmark consisted on running the serial code in the PPE. Once level 1 cache latency and PPE issue width were optimized, here the memory latency can be determined.

- **DMA transfers.** For this test, only the DMA transfers were performed and measured in both cases, sending and receiving the data. This test was executed to study the DMA latency parameter.

- **SPE task.** In this case, only one task was executed on one SPE to determine the best SPU issue width approach.

- **Total execution time.** The entire matrix decoding was executed using 8 SPEs in addition with the serial processing.

- **Scalability.** This is the final test where the relationship between 1 and 8 cores executing is determined.

<table>
<thead>
<tr>
<th>Test</th>
<th>Cell Blade</th>
<th>CellSim</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task PPE</td>
<td>33.072 cycles</td>
<td>34.361 cycles</td>
<td>+ 3.90 %</td>
</tr>
<tr>
<td>Memory</td>
<td>168,740.509 cycles</td>
<td>170,491.780 cycles</td>
<td>+ 1.01%</td>
</tr>
<tr>
<td>DMA transfers</td>
<td>55,927.075 cycles</td>
<td>55,031.064 cycles</td>
<td>-1.6 %</td>
</tr>
<tr>
<td>SPE task</td>
<td>44252 cycles</td>
<td>44253 cycles</td>
<td>0 %</td>
</tr>
<tr>
<td>Execution time</td>
<td>215,420.701 cycles</td>
<td>213,306.584 cycles</td>
<td>- 0.99%</td>
</tr>
<tr>
<td>Scalability</td>
<td>7.35</td>
<td>7.67</td>
<td>+ 4.35%</td>
</tr>
</tbody>
</table>

Table 4.4: Validation results.

As seen in Table 4.4, four of the results are within a precision of 1.6%, which means a very good approach. For the other two tests, also a good accuracy has been obtained, where these results are the best we obtained during the procedure.
4.3 Conclusions

In this chapter, the simulation environment and the simulator used for the Nexus implementation have been presented. CellSim is the implementation of the Cell BE for the UNISIM simulator environment. As our goal is to implement Nexus in CellSim, the UNISIM rules must be followed. Therefore, the UNISIM wrapper for the TPU has been defined and connected to the existing simulator. Additionally, the UNISIM communication protocol has been implemented, containing the signals and its sensitive methods. To be able to address the new block from the other blocks in the architecture, an address range has been assigned and introduced in the routing tables of the simulator. As we are working with a simulator instead of the real platform, the simulator has been validated in order to be as accurate as possible.
In Chapter 4 it has been shown how the TPU is added to the architecture of CellSim. This block communicates with the rest of the architecture modules by using the input and output ports, and sensitive methods that react to changes in the signals present on these ports. In this chapter, we explain how the functionality of the system is spread over these sensitive methods.

## 5.1 Block Overview

The functionality has been split in two main blocks, the dependency manager, containing the control logic and the tables involved in the dependency resolution, and a second part responsible to handle the memory mapped register interface.

The main functional core of the task pool is programmed inside the main file, `TaskPool.sim`. Together with the functionality, the queues of the design, as depicted in Figure 5.1, are defined: the in-buffer queue, the ready queue, the finish buffer queue, and some internal queues. Among this internal queues, the LoaderToHandler queue is used by the descriptor loader to indicate to the descriptor handler the presence of a new task descriptor that must be processed. Next internal queue is the queue where SPE requests remain until being attended. When an SPE tries to retrieve a ready task but there are not any element in the ready queue, the SPE request is stored into this internal queue. Last internal queues are those keeping track of the free elements in the task storage and the task table. Since the task table and the task storage have a fixed size, it is not possible to allocate more than a certain number of entries in each of them, as it has been stated before. After reaching their limit, it is needed to wait for a finished task that release an entry in order to place new entries. Therefore, it is necessary to keep track of free elements on the lists that will indicate whether it is possible to add new items or not. This has been implemented as a queue that contains the number of free elements in the task table. In the case of the consumer and producer tables instead, a hash function with collision resolution has been used, as these two lists need a quick access to its elements. For these two last tables, the occupation of each entry is determined in the table entry itself. The similar case of the task table is applicable to the task storage, which also has a fixed size.

Due to the task descriptors being fixed size as well, as explained later, tasks descriptors are written into the indexes of the task storage as it has been described before. Accordingly, we must define an extra queue with the indexes of those addresses in the memory mapped task storage. If there is not any entry available in either the task table of the task storage, it is not allowed to continue processing the new task. This implies that the items of the in-buffer are not processed neither removed, causing the PPE to stop adding tasks. At the beginning of the execution, these lists have all the entries
Figure 5.1: Block overview of the Implementation of Nexus, divided in register interface and dependency manager.

present in the task table and the task storage. Whenever a new task arrives, an item in both the task table and the task storage is assigned, holding them until the task has finished. These ids in the task table and task storage are released after the task execution has been finished in the SPEs and processed in the TPU. TaskPool.sim also contains the code of the descriptor loader, the descriptor handler, the finish handler, and the additional functions necessary to complement their functionality. Additionally, within this file there is the initialization function, called at the beginning of the simulator. Such function resets the internal variables and builds up all the queues of the design with their
specific sizes. As this is the initialization, the in-buffer queue, the ready queue, the finish buffer queue, the LoaderToHandler queue, and the queue where SPU s remain until being attended, are empty. On the contrary, the queues containing the free elements in the task table and the task storage are full, as no element is occupied. The status register is configured to indicate that it does not have any task and it is available to start receiving them.

As was shown in Section 4.2.2, this file also has the ports of the module and the sensitive methods that are called when any signal of the ports are modified from the outside. These methods cover the three handshake signals data, accept, and enable involved in the communication of the TPU with the interconnection network. In case the data is modified in any of the TPU ports, its sensitive method is called. It performs a simple loop that checks if there is something on the ports, asserting the accept signal for the ports where something has arrived. Following the UNISIM rules, we cannot process the data until it is enabled from the source. If the accept signal is asserted from the outside of the TPU in any of the ports, the assigned method enables the data sent along that port to finish the handshake. When the accept signal is no longer active, the enable signal is set to 0 again. Despite having one unique method for the enable signals, two methods are distinguished, as each port must perform different operations with the incoming data. In both cases, the register interface is called according to the type of operation included in the data, either a load or a store. However, if the data comes from the first port, it targets the memory mapped registers. On the contrary, if the data comes from the second port, then the operation must be performed on the task storage. If the type of operation is a read, the TPU should not respond immediately as it incurs some delay. Therefore, after reading the required register of task storage address, the response is stored in an output buffer together with its associated delay. This delay is calculated depending on the amount of data read.

The methods sensitive to the rising and falling edge of the clock are also involved in the data transfer. Thus, for the case of the rising edge, if the output buffer is not empty, the output data is written into the output port when its correspondent delay has expired. In the falling edge case, when data has been transmitted from the TPU, this sensitive method removes the data from the output port if it has been already accepted. Notice that if the accept signal is asserted, it has been already enabled right after the accept signal arrived. Among these two functions, sensitive to the global clock, the main functionality of the system is called. Hence, the descriptor loader and the descriptor handler are called at the beginning of the clock cycle, and the finish handler is called at the end of the clock cycle. At the end of the clock cycle, if there has been any change in the status register, the status memory mapped register is updated by calling the register interface.

The register interface serves as the communication of the block with the rest of the modules in the architecture, and it is packed into the following files: Register_TPL Interface.cxx, Register_TPL Interface.hxx and TPU_MMU.hxx. As shown in Figure 5.1 this interface is placed between the dependency mananger and the interconnection network of the platform. Every access to, or from, the TPU must travel through this module, which contains the different registers used for synchronization with the PPE and the SPEs. Once the data is enabled in one of the input ports of the TPU it is
### Register Interface Memory Mapping

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>In buffer (low byte)</td>
<td>0x000000</td>
</tr>
<tr>
<td>In buffer (high byte)</td>
<td>0x00004</td>
</tr>
<tr>
<td>Status</td>
<td>0x00200</td>
</tr>
<tr>
<td>Ready queue (low byte)</td>
<td>0x00204</td>
</tr>
<tr>
<td>Ready queue (higher byte)</td>
<td>0x00208</td>
</tr>
<tr>
<td>Finish buffer</td>
<td>0x00404</td>
</tr>
<tr>
<td>Task storage</td>
<td>0x20000</td>
</tr>
</tbody>
</table>

Table 5.1: TPU register interface memory mapping. Offsets applied to the initial address of the TPU

processed. This incoming data can be either a load or a store, including or demanding the data, respectively. Once the data arrives, the Memory Management UNIT (MMU) is called, to check if the target address of the incoming data is correct. Depending on the port where the data comes from, the correspondent operation is performed either in the memory mapped registers or in the task storage. The register interface scheme has been implemented similar to the MFC register interface to keep the homogeneity of the simulator.

The MMU is implemented in `TPU_MMU.hxx` and it has the functions necessary to handle the physical addresses received from the EIB. The MMU, called from TaskPool.sim, receives as its initialization inputs parameters the physical addresses related to the TPU registers and task storage and, storing them in internal variables. The main functionality of this block is to perform the translation between the incoming physical address to TPU addresses. At the initialization of the TPU, this module performs the memory mapping of the physical addresses into register index values. During the execution of the simulator, this module is called to check the destination of the incoming data, distinguishing between task storage and registers. Additionally, given a physical address, it returns the register index related to such address. This is used in the `Register_TP_Interface.cxx` when it access to the register bank. The main functions of the TPU MMU are:

- **mapRegs.** This function executes the memory mapping of physical address to register indexes. This function is called at the beginning of the simulation as soon as the TPU is running. The memory mapping as done as in Table 5.1. The addresses in the table denote the least significant bits from the address.

- **isReg and isStorage.** These two functions are called when the TPU receives data from its ports to ensure that destination of such data is received in the right port.

- **getRegNum.** This function receives a physical address and subtracts it with the start address of the TPU. The result of the subtraction is the index of the mapped table in Table 5.1. The function returns the index contained in the checked entry of the table.
Both components of the TPU, the dependency manager and the register interface, share the access to the in-buffer, the ready queue, the finish buffer and the LoaderTo-
Handler queue. The register interface allows writing to and reading from items to and from the memory mapped registers. The in-buffer register and the finish buffer register act as the bottom edges of the in-buffer and finish buffer queues. These two queues are only accessible to write elements to the queue. Instead, the ready queue register behaves as the top edge of the ready queue and it can be only read from the outside of the TPU.

When accessing from the control logic to these queue, the operation mode is the reverse: in-buffer queue and finish buffer queue are read and ready queue is written. The status register does not include any queue. It stores valuable information of the TPU, i.e., whether it is running or not, the number of free items in the in-buffer, and the number of elements present in the ready queue. This register can be read from any module in the simulator, and it is updated at the falling edge of the clock cycle, as long as there has been any change on these values. This is done by the control logic of the TPU, which performs a register call to update the register whenever a change is detected. The register interface also includes the task storage used as centralized memory that can be accessed quickly from the SPEs. Besides being accessed from outside the module, it is as well used from inside the TPU. Furthermore, the control logic contained in TaskPool.sim may also perform different calls to the register interface to perform reads and writes. These two operations can be done both in the register interface to update the status of the TPU or to read a descriptor from the task storage. This last feature would correspond with the right line connecting the two main entities of the block diagram. The communication of the dependency manager and the register interface is entirely depicted in Figure 5.1.

The task storage, as explained in Chapter 3, stores the tasks descriptors of the tasks that are currently being processed in the TPU. Although the initial design contemplated task descriptors with a variable size, during the implementation process it was decided to set a fixed size for them. The reason is that having a variable size task descriptor would lead to more complex control logic in the TPU. If the TPU handles different sizes for the descriptors, it would be necessary to address memory allocation issues. Thus, when handling the arrival of task descriptors, it would be required to reserve memory in the task storage. In addition, the free memory in the task storage does not have to be in a consecutive fashion. Therefore, a task descriptor could be stored in separated task storage addresses. Dealing with the memory allocation would indeed slow down the task pool and increase the area of the module, due to a more complex control logic. On the contrary, having a fixed size task descriptor allows us to keep a simple mechanism for dealing with task descriptor allocation. Due to the fixed size of the task descriptors, the task storage can be seen as a table which entries would be such descriptors. Therefore, the indexes to the entries are addresses containing a multiple of the task descriptor size, starting at the task storage base address. Furthermore, the initial design of Nexus included a field for the size of the descriptor in the in-buffer, and it has been removed as every entry has the same size, making this info unnecessary.

Notice that having a fixed length for task descriptor also affects to the size of the consumers table and the producers table. Having the maximum number of operands limited allows the design to know its worst-case scenario. This worst-case scenario is produced when every tasks contained in the task table has the maximum number possible
of dependencies, and no other task has any dependency in common. This means that each address is only needed by one of the tasks contained in the task table. In the initial design of Nexus, these two tables and the task table were also meant to be lists with a variable size, as the TPU was intended to dynamically allocate memory. Hence, the TPU would stop adding tasks when it ran out of memory to place new tasks. Therefore, a variable number of tasks would be contained inside the TPU. Similar to the task descriptor, this fact would imply larger area consumed as well as more computations due to memory allocation of each entry. Instead, the task table has a fixed size that can be changed to study its impact on the system performance. Furthermore, the length of these consumers table and the producers table is determined by the maximum number of parameters in the task descriptor and the commented length of the task table. These two tables contain the addresses involved in the execution of the tasks.

```
typedef struct {
    uint8_t task_func;
    uint8_t no_params;
    uint8_t p1_io_type;
    uint8_t p1_x_length;
    uint32_t p1_start_ptr;
    uint32_t p1_y_stride;
    uint8_t p1_y_length;
    uint8_t pad1;
    uint8_t p2_io_type;
    uint8_t p2_x_length;
    uint32_t p2_start_ptr;
    uint32_t p2_y_stride;
    uint8_t p2_y_length;
    uint8_t pad2;
    uint8_t p3_io_type;
    uint8_t p3_x_length;
    uint32_t p3_start_ptr;
    uint32_t p3_y_stride;
    uint8_t p3_y_length;
    uint8_t pad3;
    uint8_t p4_io_type;
    uint8_t p4_x_length;
    uint32_t p4_start_ptr;
    uint32_t p4_y_stride;
    uint8_t p4_y_length;
    uint8_t pad4;
    uint8_t p5_io_type;
    uint8_t p5_x_length;
    uint32_t p5_start_ptr;
    uint32_t p5_y_stride;
    uint8_t p5_y_length;
    uint8_t pad5;
    uint8_t pad6;
    uint8_t pad7;
} taskdescriptor_t;
```
5.1. BLOCK OVERVIEW

Listing 5.1: Task descriptor structure.

The final descriptor format is shown in Listing 5.1. The maximum number of parameters within the task descriptor has been limited to 5, resulting in a structure of 64 bytes. The first two fields of the task descriptor are used to set the function that must be performed by the SPEs, and the number of parameters involved in the execution. For instance, when the second field has a value of 3, only the first three parameters are valid, disregarding the two of the end. Next, for each of the parameters, the type of operand (IN, OUT or INOUT) is indicated, the length in the x and y axis, the pointer to the initial address, and the stride in the y axis of the matrix. Notice that there are some unused fields in between each parameter and at the end, as the 32-bit values must be word-aligned. Since they follow this rule, these fields were defined in case in a later step they were needed.

According to the task descriptor format, each block processed may depend on four additional blocks. This is a very reasonable number, as, for instance, H.264 decoding algorithm only involves 3 blocks in the decoding procedure. For H.264, used for testing Nexus, the most complex dependency pattern only depends on three blocks. In the case of the consumers and producers tables, limiting to five the maximum number of parameters also has its impact. As no more than five parameters can be used by the same task, having in total N tasks in the system would yield in no more than $5 \times N$ different parameters. This would be the case where every task in the system had a different set of dependencies than any other task in the system. For our case of study, the H.264 decoder, and its most complex dependency pattern, it is visible that each task can hold only 3 dependencies in the tables. The size of the consumers and producers table has been chosen to be four times the size of the task table, although this relation may be modified. Choosing a smaller value however, could lead to corrupt results, as the case where there is no space in this tables is not handled. This size allows us to place the extreme case for H.264. Although this size cannot handle the worst case for the general case with five dependencies, this is very unlikely to happen as it would imply that every task is independent of all the others tasks, holding no dependencies among tasks. Nevertheless, the size of the three tables can be easily modified, but with its length should be a power of 2, as we shall see later. Within the TPU definitions, the width of consumer table and producer table size is defined as well, and only that width is taken into account in the current implementation. In addition, the task descriptor structure can be changed easily to allow including more parameters and hence more dependencies.

All these modifications can be done in the file `tpu_defs.h`. In this file, the aforementioned sizes for the tables and the task descriptor format are defined. In addition, this file also includes the type definitions of the queue and table elements. Furthermore, it is possible also to tune the delay of the TPU processing the tasks, by changing the latencies involved in the lookups and the register interface communication. This file also contains the flags used by the control logic.

In the following, the life cycle of a task is shown, including all the elements that will be introduced in this chapter. Next, the overall design of the Nexus implementation is shown, explaining in detail the code encountered in the `TaskPool.sim` file and the register
interface. Later in this same chapter, we explain the methodology used for allocating entries in the dependency tables, the producers table and the consumers table.

## 5.2 The Task Life Cycle

Figure 5.2: The task life cycle observed from the PPE (a) and from the SPE (b).
In this section we review the life cycle of the tasks explained in Chapter 3, including the internal elements introduced during the implementation time. Notice that the different steps have been colored to distinguish among the modules that intervene on each of them. There are three modules that take part of the entire process, that in order of appearance are:

- **PPE colored in blue.** It is the responsible to add the tasks to the TPU by indicating the addresses used for every task execution. After being signaled that the TPU has finished the execution, it is aware of the application being finished.

- **TPU colored in green.** It handles the dependency resolution of tasks added from the *ready queue*, and synchronizes the communication with the SPEs through its memory mapped registers.

- **SPEs colored in red.** These cores execute the tasks signaled from the TPU, loading the data from the main memory and storing the results after they are processed.

The entire life cycle of a task is depicted in Figure 5.2. First, the PPE prepares the task descriptor shown in Listing 5.1 by writing its parameters concerning the function that must be performed and the dependencies involved in the operation. Once the descriptor is ready, the PPE checks whether there is any room in the *in-buffer* by reading the status register of the TPU. The PPE keeps reading this register until it detects a free element of the *in-buffer* of the TPU. Once it is granted the free element, the PPE writes the main memory pointer of the task descriptor in the *in-buffer* of the TPU. After this has been completed, PPE starts preparing a new task descriptor to submit to the TPU.

The TPU detects new data on its memory mapped registers targeted to the register interface. Therefore, the register interface module is called having the new data as a parameter. The register interface module recognizes the load into the *in-buffer* and performs the write by adding a new element at the bottom of the correspondent queue. After the new element has been added to the *in-buffer* queue, at the beginning of the next cycle and if the *descriptor loader* is not busy processing any other previously added task, it processes the new arrival. Notice that since the *in-buffer* is a queue, all the elements are treated in order FIFO. The *descriptor loader* detects the new task by checking the size of the *in-buffer* queue. When this queue has a size larger than zero, it is attended. Before loading the descriptor, we must first ensure that there is enough room in the task storage. This is done by checking the queue that contains the free entries in the task storage. When TPU has guaranteed this, it retrieves the descriptor from the main memory. When this data arrives to the TPU, the register interface module issues the data to task storage to write the targeted address. Once the descriptor is loaded into the task storage, the internal queue *LoaderToHandler* is used to signal the *descriptor handler* the presence of a new task to be processed.

The *descriptor handler* checks the *LoaderToHandler* queue and detects that there is an unattended task. The next step is to check the availability of free items in the task table, similar to the case of the *descriptor loader* and the task storage. Also similar to the *in-buffer* queue, the elements of the *LoaderToHandler* queue are processed in order FIFO. When there is not any free item in the task table, the *descriptor handler*
does not process the task. Instead, it check every cycle whether there is a free entry in the task table. As no element from the \textit{LoaderToHandler} queue is being processed, the \textit{descriptor loader} continues adding tasks until this queue is full. As a consequence, the \textit{descriptor loader} reaches a state where it cannot process tasks from the \textit{in-buffer}, which also gets full, forcing the PPE to stop adding tasks. If otherwise, the \textit{descriptor handler} succeeds at finding an empty entry, it continues its execution, assigning the free position of the table to the new task. This position in the table is also the task id that identifies the task inside the TPU and will be sent to the SPEs. Next, the \textit{descriptor handler} performs the lookups in the producers and consumers tables. This determines the current dependencies of the new task with the previously added tasks. After this procedure, the \textit{descriptor handler} fills the required information in the assigned entry of the task table. In case the new task is dependency-free, the \textit{descriptor handler} adds the new item to the \textit{ready queue}. Notice this time it is not needed to use the register interface as the \textit{descriptor handler} is accessing the \textit{ready queue} from inside of the TPU. It might happen that the \textit{ready queue} is already full of tasks waiting to be served. In this case, the task id is stored in an internal variable and will be added to the \textit{ready queue} the next clock cycle where there is a free item in the \textit{ready queue}. While this is happening, no other task is processed until the ready task is finally added to the queue. At this point, regardless of having added the task to the \textit{ready queue}, the \textit{descriptor handler} exits waiting for new tasks to issue.

After the SPEs are initialized from the PPE, they perform a blocking read on the \textit{ready queue} waiting for tasks to execute. Due to its blocking nature, SPEs remain idle until they are assigned a task. The TPU receives a request on the register interface with a target address pointing to the \textit{ready queue}. Therefore, the TPU calls the register interface module to read the targeted register. If no task can be send to the SPE, the request is enqueued in an internal queue. Whenever there is an available task, either the first time the SPE requests it or after being enqueued, the \textit{ready task} must be sent to the worker core. The \textit{ready task id} and the task storage pointer of the task descriptor are packed and delivered inside the \textit{MemoryAccess} class received from the SPE. After receiving the \textit{task id} and the descriptor pointer, the SPEs must read now the task descriptor from the task storage. Thus, a new request is done to the register interface, but this time pointing to the task descriptor address. Again, the TPU receives this request and calls the register interface targeting the task storage this time. However, this read is performed immediately as no checking is involved for this operation as it is ensured the availability of the descriptor. The task descriptor structure is sent contained in the same \textit{MemoryAccess} class of the request. The SPE receives the descriptor and loads the data needed, according to the descriptor, from the main memory through a DMA command. After receiving the data, the SPE performs the operation indicated in the task descriptor. Once this has been done, it must signal to the TPU the finalization of the task. This is done by writing into the \textit{finish buffer} the task id received before. Similar to the case of the \textit{in-buffer}, the register interface is called, this time trying to write the finished id into the \textit{finish buffer}. The register interface adds the new item to the finish queue waiting to be processed by the \textit{finish handler}. After completing the communication with the TPU, the SPE tries to read another task from the \textit{ready queue} of the TPU.
Notice that both handlers, the descriptor handler and the finish handler, share the access to the consumers table and the producers tables. In the real platform, a race condition could happen if they were both performing an operation concurrently in the same table entry. Since this work has been carried out in a simulator, no race conditions ever happen here, as every handler operation is atomic. Therefore, their executions are finished in the same clock edge, the descriptor handler in the raising edge and the finish handler in the falling edge. However, this is an issue to take care of in the real platform. Thus, a mutual exclusion mechanism must be implemented in the real hardware to avoid race conditions, implying an extra delay in every execution of the handlers. One possible solution is to use an atomic test-and-set hardware instruction, and a flag on each entry of the consumers and producers tables. Therefore, each time either of these handlers needed to access a particular table entry, it would first need to execute such instruction on its correspondent flag. When the access is granted, the handler is allowed to perform any operation in the entry, otherwise the handler must keep waiting until the entry is no longer accessed from the other handler. This will prevent from both handlers accessing to the same entry at the same time. It is necessary to apply the mutual exclusion to each table entry separately. If applied to the entire table, the performance of the system could drop significantly because both handlers could not operate concurrently, as they generally access to the tables every execution.

As it has been explained, both the register interface and the dependency manager share the access to different queues of the task pool, which is illustrated in Figure 5.1. These queues serve to communicate both entities, as the register interface entails the communication with the rest of the blocks of the architecture, and the control logic carries out the main functionality. Next two sections explain more in detail the implementation features of both components of the block.

5.3 Control Logic

5.3.1 Descriptor Loader

The descriptor loader, whose interactions are depicted in Figure 5.3, is responsible for taking the task descriptor pointers from the in-buffer queue(1), and load the task descriptors from the main memory. After loading the descriptor to the task storage, it must signal to the descriptor handler the presence of task arrivals. This is done by means of the internal LoaderToHandler queue. This queue contains the addresses in the task storage of task that have arrived recently to the TPU and which dependencies must be processed yet. Since this block needs to request some data from the main memory, it cannot be sure when the descriptor has arrived. As a result, the register interface is the one adding the pointer to the task descriptor in the LoaderToHandler queue. In this manner, this block builds the memory request and indicates in the internal queue that there is one in-flight transfer. Such in-flight transfer cannot be processed by the descriptor handler, as its task descriptor is not in the task storage yet. However, from the descriptor loader, in-flight transfers are considered as occupying elements in the internal queue. Otherwise, we would not be able to know how many elements were being requested to main memory, being possible to exceed the capacity of the queue when
they arrived. For instance, let the *LoaderToHandler* queue be full with task descriptors already loaded to the task storage. If a new task descriptor arrives and we tried to add it to the queue, it would overwrite a queue element causing a corrupt result.

![Diagram](image)

Figure 5.3: Descriptor loader and its interactions within the TPU.

After checking the available size in the *LoaderToHandler* queue and in the task storage(2), the descriptor loader starts building the request of the task descriptor(3). The request is a *MemoryAccess* load targeted to the address of the descriptor. However, the pointer sent from the PPU is not a physical address but a virtual address. Accordingly, it is necessary to call the Translation Lookaside Buffer (TLB), which maintains the relationship between physical and virtual addresses. Having the virtual address as an input parameter, the TLB returns the correspondent physical address. The source address of the memory request is set as the task storage address previously taken from the queue. As the main memory responds to the source address(4), the response travels towards the task storage in the register interface, where it is written. Notice that when the register interface receives the task descriptor, it immediately writes it to the targeted position, as it is guaranteed to be free. Similar to the case where an external entity wanted to read the task storage, the memory request is enqueued in the output buffer of the register port with its associated delay. After performing the request, the descriptor loader increments the number of in-flight transfers, that will later occupy one position in the queue. As it is seen in the register interface section, after writing the task storage, the task descriptor address in the task storage is added to the *LoaderToHandler* queue(5). This signals the descriptor handler that it needs to read the new descriptor and process it.

The descriptor loader consumes some time performing the operations mentioned
before. Until this time has passed the loader is busy and therefore it cannot process anything else. To simulate such behavior, a variable has been defined containing the time the loader cannot run again. Accordingly, after executing the loader this variable is set to a fixed latency, which can be tuned in the \texttt{tpudefs.h}. We assume its latency is stationary as this block is doing the same operations and we are not capable of determining the time consumed by the TLB. The next cycles after the loader has been executed, this variable is decreased once every cycle until it reaches zero. When this happens, the \textit{descriptor loader} is ready again to process new tasks.

### 5.3.2 Descriptor Handler

The purpose of the \textit{descriptor handler}, illustrated in Figure 5.4 is to process the tasks submitted from the PPE, after the \textit{descriptor loader} has received the task descriptor of these tasks. As it was explained before, the \textit{descriptor loader} signals the presence of new tasks through the internal queue intended to communicate both entities. The \textit{descriptor handler} fetches the descriptor pointers, processes the dependencies, and introduces new entries in the task table, filling their fields after processing the tasks. In addition, if the new task has no dependencies, the \textit{descriptor handler} must add them to the \textit{ready queue}.

![Figure 5.4: Descriptor handler and its interactions within the TPU.](image)

Every time the \textit{descriptor handler} detects a new item in the \textit{LoaderToHandler queue}(1), it tries to process it. However, similar to the case with the \textit{descriptor loader} and the space in the task storage, the \textit{descriptor handler} must check the availability of free item in the task table. It uses the same mechanism as the \textit{descriptor loader}, a queue containing the free elements in the task table(2). If there is not any available entry in the task table, the TPU must wait until a task is finished releasing an entry. If that is the case, it may happen that the \textit{descriptor loader} keeps adding tasks to the internal queue,
but these are not attended. As a consequence, this queue gets full and the in-buffer later too. This is the case previously explained, where the PPE is stalled waiting to add new tasks. On the other hand, if there is an available entry, the descriptor handler starts determining the dependencies of the new task with the previously added ones.

The first steps are to assign a task id to the new task and load its task descriptor. Such task id is the index of the task in the task table that serves to distinguish the task within the TPU(2). Once the task has the unique id, the descriptor handler loads the task descriptor from the storage(3) to start processing its dependencies. This is done calling the register interface function that returns the task descriptor given its index. Such index is the element obtained from the LoaderToHandler queue used to communicate this handler with the descriptor loader. Thereby, it handles the dependencies of each task operand(4) following the scheme presented in Section 3.2. Since the range of possible addresses is very large and it could not fit in the tables, we cannot assign a different entry for each address. Therefore, a mechanism is needed to map each address to an entry in the tables, trying to minimize the number of mappings to the same item. To do that, a function called find_list_id has been implemented, consisting of a hash function together with a collision resolution algorithm. This mechanism is explained in detail later in Section 5.3. The function takes an address as input parameter and returns the mapped entry in the table. Here two cases are distinguished, when the handler wants to add the task id to the kick-off list of an existing element, or when it wants to write a new entry or set a barrier. For the first case, if the searched address is found in the table, this function returns the address where it is located. Hereby, the descriptor handler adds the new item to the kick-off list. If no entry is found for that address, the function returns an invalid entry and the handler continues the execution without adding the task to any kick-off list. When the handler wants to introduce a new item or set a barrier, this function necessarily must return an index. Once that index is returned, the handler must check if the entry is empty or not. If it is empty, the new item is introduced, changing the status of the entry to taken. Otherwise, it means that the address is already being processed by another task and therefore the handler sets the barrier.

After all the dependencies concerning every parameter have been resolved, the descriptor handler fills in the entry in the task table(5). This entry contains the task descriptor pointer in the task storage, the number of current dependencies, and the state of the task, which is 1 when it has no dependencies and thus it is ready. If the task does not have any dependency, it means that it can run on the SPEs and therefore, the descriptor handler must add it to the ready queue(6). If the ready queue is not yet full, we simply push the new ready task id to the bottom of the queue, together with its descriptor pointer. Nevertheless, it might happen that the ready queue is already full of tasks waiting to be executed. In such a case, the descriptor handler stores the id and descriptor pointers in a temporary variable, and sets to 1 a flag indicating that there is one element to be added. After these two conditions, the descriptor handler exits the execution. After its correspondent delay, the descriptor handler is called again at the rising edge of the clock. If it finds an element ready but not added to the queue, this handler gives priority for this case and tries to add the element to the queue. In this manner, no new task is processed until this ready element is finally introduced in the queue.
5.3. CONTROL LOGIC

The latency of the descriptor handler is calculated similar to that of the descriptor loader. Due to this latency, in case any block must be written into the ready queue, its write is delayed and the block stalls. Additionally, the handler is not executed again also until the latency has expired. At the beginning of any execution of the descriptor handler, the variable containing its latency is zero. During the execution, this value is increased every time we need to read or write at any position of the task storage or in the dependency tables. The hash function and its collision resolution mechanism, that are explained later, are also taken into account. The hash function and the collision resolution are used every time an element is searched on the consumers or producers tables. Moreover, this procedure may read and write several positions in the table. Also the latency added by the loops is taken into account, and it depends on their number of iterations. The last step affecting the latency is the addition of the ready element to the ready queue, which also consumes some time. The weight of each action in the total delay incurred by the descriptor handler can be tuned in tpu_def.h. This allows us to study the impact of the handler delay in the overall behavior by setting higher or lower latencies.

5.3.3 Finish Handler

When the SPEs finish the execution of a task, it is the responsibility of the finish handler to process its dependencies. The SPEs signal the execution end of a task by writing its id into the finish buffer queue. The finish handler is continuously checking this queue to detect new arrivals. Once a new task id is detected, it processes its operands and tries to add new tasks to the ready queue. After processing all the dependencies in the consumers and producers tables, the finish handler removes the entry assigned to the task in the task table and in the task storage. This memory release allows new tasks to be added to the TPU from the PPE.

![Figure 5.5: Finish handler and its interactions within the TPU.](image)

The finish handler, depicted in Figure 5.5, is active every falling edge of the clock
and when it is not busy, it checks the *finish buffer* queue waiting for finished tasks (1). If a new element is encountered in the queue, it immediately starts handling the id. For this handler there are not any memory limit issues, as the finished task are already allocated in the memory, and no extra memory is necessary. The first step of the *finish handler* is to read the task descriptor pointer associated to the id (2). The descriptor, as stated before, is written into the task table entry indexed by this task id. Therefore, it is necessary to call the register interface to load the task descriptor structure (3). Once the descriptor is loaded, the *finish handler* starts processing the operands related to the finished task. The dependency resolution (4) has been already explained in Section 3.2. In a similar case to the *descriptor handler*, the *finish handler* needs to obtain the entries in the producers and consumers tables associated to the current address operand, using the function *find_list_id*. In contrast to the *descriptor handler*, this handler only needs to check one table for each operand. This means that for the input operands, only the consumers table is checked; while for the output operands, the producers table is consulted. For every case, the *kick-off* list of the correspondent entry is processed. For every element removed from this list, the *finish handler* calls the *kick_off* function with the task id removed as the input parameter. Within this function, the dependency count of the id removed from the *kick-off* list is decremented on its task table entry (5). In case the dependency count of the task handled is zero, the function tries to add them to the *ready queue* (6). If this is not successful, the status of the task table entry is changed to 2, meaning that it is ready but not added yet to the *ready queue*. The *finish handler* keeps track of the number of ready elements not added to the queue, and hence this number is incremented. When the *finish handler* is idle, it checks the status of the elements in the task table. This is done once per clock cycle the handler would be idle. To do it, the *finish handler* also keeps an index containing the last item checked in order not to check the same indexes.

If the *kick-off* list is empty after removing its suitable elements by shifting the list, the *finish handler* must remove the entry. The emptiness of the *kick-off* list is determined by a barrier that denotes the end of the list. When this is the case, the *finish handler* calls the *find_list_id* function that issues the deletion, as is explained later. The last step of the *finish handler* is to release the elements occupied by the task and its descriptor in the task table and in the task storage (8), respectively. The indexes in both tables are pushed at the bottom of the queues that contain the free indexes in the tables. In this manner, the finished task is considered no longer as present in the TPU, and the handler finishes its execution.

The latency consumed by the *finish handler* is calculated similar to the latency of the *descriptor handler*. Its value is determined by taking into account the reads and writes to the task storage and the tables of the dependency manager, besides the access to the *ready queue* in case it is needed. Also similar to the *descriptor handler* case, the *find_list_id* function returns the number of cycles consumed on its execution. However, in this handler, several tasks may be added to the *ready queue* during the same handler execution. Since they are processed sequentially, two tasks added to the *ready queue* should not be become accessible at the same time. This is because while the first task is ready, the other must be yet handled, and thus have an additional delay. To cope with that, an internal queue has been defined, which contains the elements to be added to
the ready queue and their associated latencies. After the finish handler has finished the execution, the next cycles this internal queue is checked looking for elements that must be added to the ready queue. The elements of this internal queue are sorted according to the time they must be added to the ready queue. As the latency associated to each task is the time when the finish handler is done processing such task, the maximum latency associated to a task its the final latency of the execution of the finish handler.

5.4 Register Interface

The main functionality of the register interface is quite simple, as it must take care of the reads and writes concerning the memory mapped registers and task storage. This must be a low-latency system as it is supposed to perform a very fast communication with the rest of the architecture. This interface must take care of some conditions that could harm the system. The register interface, as the dependency manager, has an initialization function that starts the module by resetting the register and setting the permissions aforementioned. The other function contained here is the update of the status register, which must be called from the control logic whenever an important event has happened. These events concern whether there are tasks in the TPU and thus it is active, and the number of elements in the in-buffer and the ready queue. The main functions of the register interface are explained in the following sections.

5.4.1 Read Register

This function is called when an external entity wants to read any of the two registers allowed to be read, the status register and the ready queue register. In both cases, the function should return the MemoryAccess class received, but now including the requested data. For the status register, this is always the case, as it is always possible to read the status of the TPU. When trying to read the first element of the ready queue, the register must take care of some possible situations.

First, depending on the source of the request, this access can be done in one or two data transfers. If it is the PPE who is retrieving the element, its access is performed in two steps, as in CellSim the accesses from the PPE can only fetch 32-bit values. A ready queue element has two 32-bit values, the task id and the descriptor pointer. Thus, for the first access, the first element of the queue is taken and the task id is sent. The descriptor pointer is stored in an internal variable, which is sent the second time the PPE performs the read. Notice that this solution is only valid for our case where only 1 PPE is present in the architecture. For the case of more than 1 PPE requesting elements of the ready queue, the accesses of the PPEs could be interleaved, and thus would received incorrect values. When accessing from the SPEs, the ready element is read at once, because these cores perform the read to the entire register, instead the two-step access from the PPE.

In the previous paragraph, it has been assumed that there are elements in the queue. It may happen, however, that the ready queue is empty and no task can be sent. To avoid the finish buffer to be able to be full, another condition has been taken here into account. Therefore, when the number of elements in the finish buffer is higher than a certain threshold, the ready elements are not sent either. Otherwise, the finish buffer
could be full at some point, and a new finished task arrival would overwrite the queue. Hence, this mechanism prevents the SPEs or the PPE to keep adding finished tasks when the previously finished ones have not been handled yet. The threshold can be also changed in the `tpu_defs.h` file. When reading the element in the queue is not allowed, this case is handled depending on the element that has perform the read. If the PPE is the source, the response is a 0 in the data container. This allows the PPE to decide whether to keep asking for tasks or to anything else. In the case of the worker cores, their unique function is to execute tasks and hence, their requests are enqueued. In this manner, SPE request are blocking. A function also included in this file is responsible for checking the existence of elements in this queue. In case the `ready queue` is not empty, this function attends the request and removes the petition and the ready element from their respective queues. Such function is called from the dependency manager at the beginning of every cycle.

5.4.2 Write Register

This function performs the write operations to the memory mapped register. As the read register case, only two register are allowed to be written, the `in-buffer` register and the `finish buffer` register. After the register interface writes the data, the `MemoryAccess` class is destroyed. Although initial design of the TPU envisioned the `in-buffer` to have elements containing the descriptor pointer and its size, the current implementation only includes the descriptor pointer field. The reason is the already commented fixed size for the descriptor size. Thus, in this case it is not necessary to distinguish the source of the access, as PPE and SPEs all write 32 bits.

The code of this function is very simple, as this time there are not special cases and the values are written immediately. In the `in-buffer` register, the responsibility of not exceeding the queue size is left to the PPE, which checks the size of the queue before adding new values. In the case of the `finish buffer`, it has been shown in last section that after a certain number of elements in the `finish buffer`, it is not possible to read items from the `ready queue`, preventing to go beyond the queue size. Therefore, in both cases, when a new value comes, and adding a new element exceeds the queue size, the first element of the queue is overwritten.

Writing to any of the register modifies the number of tasks present in the TPU. If a new task is written into the `in-buffer`, it means that a new task must be processed, and the count of tasks present in the TPU is incremented. Writing to the `finish buffer` decreases the number of task processed in the TPU. After the finished task is issued by the `descriptor handler`, its entry in the task table and in the task storage is removed. This task counter may modify the state of the TPU and therefore the update state function must be called. When it is zero, the TPU is done processing tasks and is no longer active, while the addition of a new task means that the TPU is active, as it has at least one element task being handled.

5.4.3 Task Storage Operations

The functions related to the task storage are divided in two groups. The first two presented here are called when the storage is accessed from outside the TPU. The next
two are used by the control logic to read the descriptors associated to the task they are processing. The functions are:

- **ReadStorage**. This function is called whenever an external entity wants to read a task descriptor. Since this is allowed in any case, this function only fetches the descriptor from the task storage and returns it.

- **WriteStorage**. In this case, the function performs a write operation of the input data on the address determined in the input MemoryAccess. In this case, this function must signal the descriptor handler that there are new tasks to be handled. As shown before, the index of the descriptor in the task storage is added into the LoaderToHandler queue. Notice that the availability of space in this queue has been already checked in the descriptor loader. Additionally, since the task descriptor is already loaded, the number of in-flight memory transfers is decremented.

- **GetTaskDescriptorIndex**. Given the address of the task descriptor requested, this function returns the index of that address in the task storage. Remember that the task storage is treated as a table containing task descriptor as the entries.

- **ReadDescriptor**. This function is called from the control logic and returns the task descriptor requested given its index as the input parameter.

## 5.5 Hash Function and Collision Resolution

The producers table and the consumers table have a fixed length, limiting their number of entries to a few thousands. Each of these entries should be suitable to allocate an address that is being processed by a task. Considering the data divided in $16 \times 16$ byte blocks in our benchmarks, and blocks are referenced by its start address, and that the possible address range is 32-bit wide, this leads to $2^{24}$ possible entries in the tables. Obviously, this is an unsustainable situation, as it would require a vast amount of data occupied by the tables. Therefore, we need a mechanism that transforms from a range of 32-bit to a few thousands entries.

Hash tables are data structures that distribute a large set of input parameters, known as keys, amongst a set of slots by using a hash function. The requirements for the hash function are that it must have low cost in terms of speed, and it has to be deterministic and uniform. Since our hash function is intended for table lookups and these happen quite often, it must be performed fast and therefore it cannot include complex operations. Hash functions are deterministic, as for the same input they always produce the same output. This is a fundamental requirement for our design, as operands with the same address must be mapped to the same address to recognize a dependency between them. Last, the hash table should distribute its inputs as evenly as possible over its outputs in order not to overload some slots, yielding in many collisions to the same entry. In our study case, although it is possible to receive any address range within the 32-bit value, the data treated is normally allocated within a short consecutive address range. Thereby, the hash function selected should perform well for small consecutive ranges. The hash function selected [2], detailed in Listing 5.2, consists of simple shifts, additions and
subtractions that perform the conversion quickly. It is characterized by being a \textit{half-avalanche} hash function, where every input bit affects itself and the higher bits of the output. This implies that higher input bits do not have any impact in lower output bits, being the main effect produced by lower input bits. This is the optimal case for our table lookups, where due to consecutive memory addresses, the higher input bits barely change. Accordingly, the output produced is mainly affected by lower input bit changes. Several hash functions of this type were tested receiving a range of legal memory addresses. The picked one has been the one that offered the best results in terms of collisions.

```c
uint32_t hash_function(uint32_t key)
{
    uint32_t a = 0;
    a = key;
    a = (a+0x479ab41d) + (a<<8);
    a = (a‘0xe4aa10ce) ‘(a>>5);
    a = (a+0xa942fa6) − (a<<14);
    a = (a‘0x5aded67d) ‘(a>>3);
    a = (a+0x17bea992) + (a<<7);
    a = a‘(32–DEPENDENCIES_LIST_WIDTH);
    a = a>(32–DEPENDENCIES_LIST_WIDTH);
    return a;
}
```

Listing 5.2: Hash function.

Although the hash function tries to avoid collisions to the same table entry, it is inevitable that two different addresses are eventually mapped to the same entry. Thus, a collision resolution mechanism is needed to solve this problem. To handle the situation where two different addresses are mapped into the same table entry, two main approaches seem to be the solution: separated chaining and open addressing [10]. The separate chaining scheme is based on building a list of elements mapped to the same entry. In accordance, when after hashing some input it produces an occupied slot in the table; separate chaining allocates the new element and links it to the resulting entry. If this happens again, the new element is allocated after the last element of the list. This would imply to dynamically allocate elements in the memory. However, as the tables in the TPU have been designed to have a fixed size, this is not possible. Instead, open addressing works on tables with fixed length. For the case of two inputs mapped to the same item, the idea of this method is to allocate the new element in the next item free in the table. This is, once the desired item is occupied, open addressing travels down the table trying to find a new item. If the bottom of the list is reached, it continues from the top of the table down until it finds an empty position or the first item checked. If it reaches again the hashed value, it means there is no free entry in the table. Notice that in separate chaining this case did not happen because of the memory is dynamically allocated.
Both mechanisms have their advantages and disadvantages. When we need to access one specific item of the table, separate chaining performs it very quickly as only the list linked to the hashed entry must be checked. On the other hand, in the case of open addressing, if the item is not in the table, it checks every item and thus involves a large latency for the worst-case scenario. Separate chaining entails a quick search engine at the cost of dynamically allocated memory, when open addressing offers a slower alternative with a fixed table size. Neither dynamic memory nor huge latencies are suitable for the TPU design. Instead, a combination of both schemes has been chosen, i.e., the coalesced hashing [25]. The idea of the coalesced hashing is to build lists of elements mapped to the same entry, given a fixed table size. The elements of the linked lists are the different slots of the table rather than structures dynamically allocated. Therefore, a new field is needed in the table elements in order to be able to link two items, as illustrated in Figure 5.6. This field, called Next, points to the next element of the list. In case of dealing with the last element of the list, this field contains an end marker indicating the end of the list. Additionally, a field indicating whether the position is empty and thus
a new element can be written there. The other necessary field is the address currently mapped to that position, which allows determining which the owner of the entry is.

To keep track of the free elements of the table, an available-space list is used, as depicted in Figure 5.7. Every slot of this list must be doubly linked to be able to be removed quickly from this list. The fields Address and Next serve as the predecessor and successor pointers for this doubly linked list, respectively. The first slot of the table remains always unused and serves as the pointer to the first element in the available-space list. Accordingly, the Next field of slot 0 points to the first element, and the Address field points to the last element of the list. Similarly, the first element of the list has slot 0 as its predecessor and the last element of the list has it as its successor.

At the initialization of the function, this list is formed, where the first element of the available-space list is the element at the bottom of the table, and the last element of this list is the top element of the table. In the initial available-space list, all the elements are properly doubly linked and the list length is assumed N. The three main operations to be performed are: insert an element, search for an element, and delete an element.

The insertion procedure tries to find an empty slot where to write a new element. An example of this procedure is depicted in Figure 5.8. In this case, the function returns the empty slot to be written when the operation is successful, or the overflow signal when there are no more available empty items. The insertion operation starts by performing the hash function on the input address, that in our case correspond to the values below the lists. In case this slot is empty, the next pointer is set to the slot 0, and the current address is written into its address field, that in our example is the case for Address 1.

Additionally, the current position is removed from the list containing the free elements and marked as taken. To remove an item from the available-space list it is necessary to modify the fields of the previous free item and the next free item. Thus, the successor of the previous free item is changed to the next free item. In accordance, the predecessor of the next free item is set to be the previous free item. In our case, position 4 is linked to position 2, as the item occupied is 3. Accordingly, the address field of 2 points to element 4, as it is its predecessor. On the other hand, the next field of position 4 points to 2, as it is its new successor. In the last addition of an element to the table, although
5.5. **HASH FUNCTION AND COLLISION RESOLUTION**

The address is not mapped to index 3, it is linked to its list, and we have two lists built in one. This feature gives the names to the collision resolution, the coalesced hashing, as several lists can be melted in one. When the first element checked is already occupied, we must reach the end of its linked list. Once it has been reached, the first element of the available-space list is taken and removed from that list. Afterwards, this element is linked to the list starting at the initial hashed position, and the field \textit{next} is set to point to slot 0. The removal in the available-space list is done as the previous case, with the particularity of the previous element being slot 0. For this particular case, the first element of the available space list becomes N-2 and thus, the \textit{next} field of position 0 becomes N-2. In addition, now position N-2 must point backwards to its new predecessor using the field \textit{address}, which now becomes 0.

The searching operation is quite simple, as it is only necessary to check the linked elements of the list starting at the index obtained from the hash function. Accordingly, once the input has been transformed with the hash function, the resulting hashed position is checked. In case its field \textit{address} matches the desired address, it returns the item meaning that the search has been successful. Instead, when the element is occupied by another address it continues checking the element contained in the field \textit{next}. Similarly to the first case, when the search is successful, the function returns the index to the entry where the desired element is encountered. In the case the first element is empty, or when the last item checked contains the slot 0, the function returns a value indicating an unsuccessful search. Notice that slot 0 indicates the beginning of the available-space list, meaning that there are no more linked elements in the list.

To delete an element, the first step is to find that element in the table. In case it is already not within the list, the algorithm finishes as it does not have anything else to do. Instead, when there is a match, the algorithm must remove the element from the linked list, by setting the \textit{next} field of the previous element to the slot 0. At this point, the element has been removed from the list but not added to the available-space list. The procedure of removing the item from the linked list is more complex, as it implies the reordering of the elements of that list. An example of this procedure is illustrated in Figure 5.9. Once the element has been removed, next elements must be processed. It may be possible to have several lists linked in the same list. For instance, when a new element is inserted, there might be the case where its desired position is already taken.

![Figure 5.9: Deletion of address 2 from the table.](image-url)
from any other element forming a list. Therefore, this new item is linked to the existing list. In this example, address 2 needs to be removed from the table. It is visible that there are two different lists: one beginning in position 3, and the other one beginning in position N. In this case, the deletion of address 2 will split the existing linked list in two. Hence, after removing an element from a list, next elements must be processed to check whether the list may be split. Therefore, addresses from 3 to 5 are processed after the deletion of address 2. Thus, for each of these following elements, the algorithm checks if they belong to another list with a different initial entry. For the next element, we calculate the hash function given the address contained on it. If it returns the index of the removed item, it is copied to the hole left after the removal. This is the case of address 3, as its initial hased value is N, which is empty now. At this point, the new hole is the slot left by the last processed item, currently in N-1. Otherwise, in case the hash function returns a value that is been already occupied, that item is linked to the list starting at that position. This happens to the rest of the processed elements in the example. In this case, address 4 is linked to the list starting in position N, and address 5 is linked to the list starting in position 3. This process is done sequentially on all the elements that followed the initially removed item. At the end of the execution, a hole is left and must be added to the available-space list. That operation is done by modifying the address field of the last free item to the new free item, the next field of the new free item to the last element of the available-space list, and the address item of the new free item to slot 0. After this, slot 0 points to the newly inserted free item, which points to the previous last item of the list. In the example given in Figure 5.9, the final hole is position N-1, which is added to the bottom of the available-space list. Thus, field next of position 1 points now to N-1, indicating its successor; and field address of position N-1 is set to 1, which it is its predecessor. The next field of position N-1 is set to 0, indicating that this is the last element of the available-space list.

The latency of the function is calculated according to its accesses to the elements of the tables. As it was done before in the handlers and the descriptor loader, every access to the table element, either read or write, incurs in a certain delay. According to the algorithms explained above, several reads and writes may take place as they can travel across the tables several times. The delays associated to every read and write is the same as for the handlers, and can be tuned in tpu_defs.h. The total latency is the sum of those accesses and it is returned to the handler that called the function.

5.6 TPU Types and Configuration

The file tpu_defs.h holds the types of the different TPU components, as the table elements or the queue items. For instance the task descriptor explained before is defined, and here is where it can be modified if more operands are needed. In addition, within this file some values can be changed to configure the behavior of the TPU. Thus, the sizes of the different tables and queues of the task pool may be changed here, to study their influence in the system’s performance. It is as well possible to change several parameters regarding the latencies involved in the execution of the descriptor loader, the descriptor handler and finish handler. The element types and the queues of the task pool are shown in Listing 5.3 and Listing 5.4, ordered as:
5.6. **TPU TYPES AND CONFIGURATION**

- **Status type.** This is the type of element of the status register. This register may be read from the outside of the TPU and it is updated whenever a change in its fields occurs. The *in* field stores the number of space left in the *in-buffer* queue. When this value is 0, it means that PPE must stop adding tasks. The *state* field indicates whether the TPU holds any task or not, by setting this value to 1 when it is running, or 0 when it is not. The *ready* field has the number of elements present in the *ready queue*. Last byte of the status type is unused.

- **In buffer type.** This is the type of the elements encountered in the *in-buffer* queue. As seen in the block overview of the TPU, it has a field with the pointer to a task descriptor and the size of such descriptor. Although the *size* field is not used in this implementation, this field is left for the study of variable size descriptors.

- **Ready queue type.** According to the TPU design, this element is used for the *ready queue*, including the id and the descriptor pointer of the ready task.

- **Ready queue interal queue type.** It has been mentioned that when *finish handler* needs to add elements to the *ready queue*, it first stores them in an internal queue. This is the element type of that internal queue, that stores the element that must be added to the *ready queue*, and the latency associated to such item.

- **Task table item type.** This type is exactly implemented as the original design, containing the descriptor pointer, the status of the task and the number of dependencies. The *status* field can contain a 0 in case it still has dependencies, a 1 indicating that it is ready and added to the *ready queue*, or a 2 that means it is ready but not added to the queue. As seen in the *finish handler* implementation, this status is checked when the *finish handler* is not processing finished tasks. If a 2 is found and the *ready queue* is not full, the task having a status set to 2 is added to the queue.

- **Producers table item type.** This type contains the fields described in the initial design of Nexus, and two additional fields used to implement the hash collision resolution explain in the last section. The *kick_off* field is an array which size can be also modified with the *KICKOFF_SIZE* parameter, where elements subscribed to the address are submitted to wait until the dependency no longer exists. *Empty* indicates if the element is already written and next is used to link the element to the list. For the initial available-space list, the address field is used to link the element to the previous free element.

- **Consumers table item type.** This type is exactly as the producers table item type, with the only difference that this table does not have the field *nr_deps*.

- **Queue types.** The following type definitions are related to the queues found in the TPU. Each of these queues is defined as a standard C++ queue, and an integer that stores the maximum size of the queue. This field is checked in order to to allow more elements to enter the queue if the current size of the queue is equal to this number. Additionally, for the *LoaderToHandler* queue, the field *pending* is
included. This field contains the number of in-flight task descriptor requests to the main memory from the descriptor loader.

```c
1 // Status data type
2 typedef struct status {
3   unsigned char in;
4   unsigned char state;
5   unsigned char ready;
6   unsigned char reserved;
7 } status_t;
8 // In Buffer data type
9 typedef struct in_buffer {
10   uint32_t ptr;
11   uint32_t size;
12 } in_buffer_t;
13 // Ready queue data type
14 typedef struct ready_queue {
15   uint32_t id;
16   uint32_t descriptor;
17 } ready_queue_t;
18 // Auxiliary queue for Ready queue data type
19 typedef struct ready_queue_finish {
20   uint32_t id;
21   uint32_t descriptor;
22   uint32_t latency;
23 } ready_queue_finish_t;
24 // Task list data type
25 typedef struct task_table_item {
26   uint32_t descriptor;
27   int32_t status;
28   int32_t nr_deps;
29 } task_table_item_t;
30 // Producers list data type
31 typedef struct producer_item {
32   bool empty;
33   uint32_t address;
34   int32_t kick_off[KICKOFF_SIZE];
35   int32_t next;
36 } producer_item_t;
37 // Consumers list data type
38 typedef struct table_item {
39   bool empty;
40   uint32_t address;
41   int32_t nr_deps;
42   int32_t kick_off[KICKOFF_SIZE];
43   int32_t next;
44 } table_item_t;
```

Listing 5.3: TPU types.
5.6. TPU TYPES AND CONFIGURATION

```
// In Buffer queue
typedef struct InBuffer
{
    std::queue<in_buffer_t> _InQueue;
    uint32_t _QueueSize;
} InBuffer_t;

// Finish Buffer queue
typedef struct FinishBuffer
{
    std::queue<uint32_t> _FinishQueue;
    uint32_t _QueueSize;
} FinishBuffer_t;

// Ready queue
typedef struct ReadyQueue
{
    std::queue<ready_queue_t> _ReadyQueue;
    uint32_t _QueueSize;
} ReadyQueue_t;

// Queue of integers
typedef struct tpu_queue
{
    std::queue<uint32_t> _Queue;
    uint32_t _QueueSize;
} tpu_queue_t;

// Queue of integers for Loader to Handler Queue
typedef struct tpu_load_to_hand_queue
{
    std::queue<uint32_t> _Queue;
    uint32_t _QueueSize;
    uint32_t _pending;
} tpu_load_to_hand_queue_t;
```

Listing 5.4: TPU queues.

The length of the tables and the queues contained in the TPU can be changed, affecting the area consumed by the module. In the case of the tables, it is possible to determine the length of the task table. This size will determine to the length of the consumers table and the producers table. It has been already explained that the length of the latter tables is set as 4 times the length of the task table due to the current task descriptor format. As it has been mentioned before, although their length could be
changed, we must be careful as the current implementation does not handle running out
of entries in these two tables. Although the size of these two tables could be anyone, it
is preferable to be a power of 2 because of the hash function, as it returns the indexes as
power of 2 outputs. Due to the current implementation of the hash function. Therefore,
the task table length should be a power of two as well. The value chosen for the task
table length may affect the maximum attainable scalability. For instance, if we only
allowed 8 tasks to be at the same time in the TPU, the scalability cannot be beyond 8,
even though we had 16 processors.

The number of elements allowed in the queues clearly is important for the perfor-
mance of the TPU. Notice that when there is not enough space in one of the queues,
it produces a pipeline effect that may either stall the PPE adding tasks or not to add
more tasks to the ready queue. We can tune the number of elements contained in the in-
buffer queue, the LoaderToHandler queue, the finish buffer queue and the ready queue.

Another restriction has been set in order to avoid the case where SPEs tried to write
finished tasks when this queue was already full. This would produce to overwrite one of
the positions of the queue, corrupting the resulting data and the dependency resolution.
Thus, the maximum number of elements in the finish buffer queue is defined, which must
be smaller than its maximum size. When the size of the finish buffer queue is larger
than its allowed number of tasks, the TPU stops sending ready tasks to the SPEs until
this number decreases below the allowed number.

The last configuration feature is the latency involved in the descriptor loader, the
descriptor handler and the finish handler. Since the descriptor loader does not perform
any read or write to the tables or the storage, its latency is assumed to be independent
and therefore is set apart as a base latency for the descriptor loader. The total latency of
descriptor handler and the finish handler is variable, as it is calculated as a base latency
plus a value depending on its accesses to the internal memory. For these two cases, each
time they perform a read or write in the tables or the storage, the latency consumed
by these operations is added. Additionally, the hash function called from both handlers
also consumes time due its mathematical operations. Last, whenever a new element is
added to the ready queue, it is assumed that this procedure consumes extra time as it
needs exclusive access to the shared resource.

5.7 Conclusions

In this chapter, the implementation of the Nexus task pool has been explained. Its func-
tionality has been split in two main blocks, the register interface and the dependency
manager. The register interface contains several functions to handle the communication
with the rest of the components of the architecture, allowing them to write and read
the memory mapped registers of the TPU. Additionally, the register interface also pro-
vides useful functions to the dependency to access the task storage or update the status
register. On the other hand, the control is the responsible of the dependency resolution
process and therefore is the component that processes the queue elements and the differ-
ent entries of the three tables. To communicate the register interface and the dependency
manager, several queues have been used, such as the LoaderToHandler queue. Due to the
wide range of possible addresses that must be allocated in the dependency tables, a hash
function with collision resolution has been implemented. The current implementation of the collision resolution mechanism builds linked lists of elements in the table, which allows efficient table lookups given that we have a fixed amount of space for the tables. As explained in the last section, different configuration parameters can be modified in this Nexus implementation to evaluate their impact in the system performance, such as the latencies and the sizes within the TPU.
Experimental Results and Analysis

6.1 Benchmarks

To evaluate the Nexus implementation, three synthetic benchmarks have been used, each of them characterized by a different dependency pattern. The dependency patterns of two of the synthetic benchmarks are shown in Figure 6.1, and the remaining benchmark has no dependencies. Using synthetic benchmarks instead or real applications allows us to study the impact of different parameters in the system response. Among these three benchmarks, the most complex one is the Complex Dependencies (CD) benchmark. This benchmark is very similar to the H.264, a state-of-the-art video coding algorithm. In this dependency pattern, every block of the matrix is dependent on its top-right and left neighbors, as illustrated in Figure 6.1(b). Due to this dependency pattern, this benchmark suffers from a ramping effect. At the beginning and at the end of the execution, only one task is available to be executed concurrently. The number of available tasks grows up to 16, and later this number decreases again. Thus, when using 16 cores, a scalability of 16 is not obtainable and therefore its maximum theoretical value is only 14.5. The dependency pattern of the Simple Dependency (SD) benchmark removes the dependency on the top-right neighbor. In this case, every row is independent from each other, as a task only depends its left block, which is in the same row. This removes the ramping effect and the maximum attainable scalability is 16 for 16 cores. The No-Dependencies (ND) benchmark is the simplest one, as every block is independent on each other. Therefore, all the blocks can be processed in parallel and the maximum scalability is 16 when 16 cores are used.

![Task dependency patterns of the SD and CD benchmarks. The ND benchmark has no task dependencies.](image)
Besides the different dependency patterns, several parameters are studied as well. Since it is possible to configure the length of the tasks due to the function `spend_some_time()`, the effect of several task lengths are measured. A parameter allows us to configure the time consumed in the computation of tasks. The total amount of time spent by each task is the sum of the DMA transfers and the task computation, and these tests range from 0\(\mu\)s up to about 245\(\mu\)s including both phases. All benchmarks process a 1024 \(\times\) 1024 matrix divided in blocks of 16 \(\times\) 16 and hence, the matrix contains 64 \(\times\) 64 blocks. As each of these blocks constitutes a task, the system must process 4096 tasks.

### 6.2 Nexus Evaluation

#### 6.2.1 Scalability of the CD, SD, and ND Benchmarks

In this section, the scalability of the Nexus hardware is evaluated for the three synthetic benchmarks described above. Each of these benchmarks is characterized by a different dependency pattern and thus involves different behaviors for the dependency resolution pipeline stages. All these tests are executed in the CellSim environment, which allows us to modify the number of cores available to execute the tasks. The TPU was configured to host up to 1024 tasks in the task table, and hence 4096 tasks in both the consumers table and the producers table. As the matrix is 1024 \(\times\) 1024 and the blocks 16 \(\times\) 16, this configuration can store 16 consecutive lines of the matrix. Thus, the maximum attainable scalability is 16 when using 16 SPEs. We have evaluated the system for varying task sizes in Section 6.2.1. The desirable behavior is that the scalability of the TPU is close to the theoretical value even for small task sizes. Nevertheless, having tasks with small sizes stresses the system, as it needs to handle the tasks quickly to feed the worker cores with tasks to execute. As explained in Chapter 2, the StarSS programming model does not scale well for small task sizes due to its three bottlenecks. The addition of the hardware support to the Cell architecture is meant to overcome these bottlenecks and provide a better scalability. The latencies of the TPU were set in accordance to the operations performed within each execution, as this test was not performed in the real processor but in the simulator. However, this is an approach that might differ from reality, and therefore the impact of different latencies in the TPU are studied in Section 6.2.2. Another determining feature of the task pool is the area consumed by the internal tables of the TPU. The size of the three tables can be modified, affecting the maximum number of tasks present in the TPU. With a smaller size, fewer tasks can be processed at the same time, and thus it might not be possible to exploit the total parallelism offered by the cores of the platform. The results of this evaluation are presented in Section 6.2.3.

#### 6.2.1.1 Scalability of the CD Benchmark

The CD benchmark case entails the most complex dependency pattern and thus, it is more demanding in terms of dependency resolution. Table 6.1 shows the average values of the different phases in the task life cycle: the submission of the task to the TPU, the descriptor loader and both stages involved in the dependency resolution, descriptor
6.2. NEXUS EVALUATION

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU add task</td>
<td>434 cycles/task</td>
<td>7.5 tasks/µs</td>
</tr>
<tr>
<td>Descriptor Loader</td>
<td>8 cycles/task</td>
<td>400 tasks/µs</td>
</tr>
<tr>
<td>Descriptor Handler</td>
<td>83 cycles/task</td>
<td>39 tasks/µs</td>
</tr>
<tr>
<td>Finish handler</td>
<td>76 cycles/task</td>
<td>42 tasks/µs</td>
</tr>
</tbody>
</table>

Table 6.1: Average throughputs of life cycle stages in the CD benchmark.

handler and the finish handler. The submission of the task to the TPU was obtained examining the execution trace with the Paraver tool [reference?]. The measurement of this time was done looking at the resulting trace from the execution. The other three were obtained from TPU inside the simulator, measuring the cycles consumed for the resolution of every task.

As it is shown in the table, the submission of the tasks to the TPU is the slowest stage of the life cycle. PPE needs 0.1358µs between two tasks added to the TPU, and thus provides up to 7.5 tasks each µs. This stage seems to be the main bottleneck of the design, although considering that the H.264 decoding takes around 10µs executing each task, it performs quite well. On the other hand, the actions performed by the descriptor loader are quite simple, and hence this stage can process up to 400 tasks per µs, and thus it has the largest throughput. Since there is no variation in the operations performed by the descriptor loader, its latency is constant regardless of the dependency pattern. In the case of the handlers, they both have similar throughput values, even though the descriptor handler performs more computations as it needs to process the dependencies. The descriptor handler must resolve the dependencies among tasks, subscribe them to their correspondent kick-off lists, and write the information of the task into the task table. The finish handler needs to process the dependencies of the finished task, and removing the entry of the task in the task table. Resolving the dependencies of a finished task may imply to remove some entry of the consumers table or the producers table. In the previous chapter, it has been explained that this procedure needs to process the elements of the linked list that follow the removed item. The execution of this process may increase the latency of the descriptor handler due additional reads and writes to the task table, as well as hash calculations. Although this process means additional latency, splitting linked lists reduces search times for later executions of the descriptor handler and the finish handler. Therefore, both handlers may reduce the latency of future executions due to the extra time spent in the finish handler for deletions. However, the latency of the finish handler is also increased due to the deletion process, but it does not grow larger than the descriptor handler. As a result, the deletion process reduces the overall latency for the handlers, but does not make the finish handler become the bottleneck.

Figure 6.2 shows the scalability of the CD benchmark for different task sizes and different number of cores. The task sizes in the horizontal axis represent the sum of the time involved in the DMA transfers, and the execution of the task itself. The task durations were obtained from the traces generated with Paraver. The DMA transfers consume 4.5µs of the total time employed in the total execution of the task. Thus, the values associated to the 4.5µs task size in the graph represent only the time consumed
CHAPTER 6. EXPERIMENTAL RESULTS AND ANALYSIS

Figure 6.2: Scalability of Nexus for the CD benchmark.

by DMA transfers, and smaller task sizes were obtained reducing the number of DMA commands. The first task size including the execution of the task is the 6.42µs case, where the computational phase of the task execution is 1.92µs. To compare this graph to the one obtained from the StarSS programming model, we must take into account this difference. Thus, the first value of the StarSS graph, 1.92µs in Figure 2.3, corresponds with the task size of 6.42 in this figure. It is visible that the scalability increases with the task size. When the task size is larger, throughput requirements of the system are less demanding. In the last measured point with tasks of 71.2µs, the system should be able to processes 16 tasks in that time in order to obtain the scalability of 16. In the bottleneck of the system, the task addition can issue up to 7.36 tasks per µs. Therefore, in 71.2µs more than 524 tasks can be added in average. Accordingly, in this case the scalability of the system is slightly over 14.3, approaching the maximum theoretical value of 14.5. On the other hand, when each task only lasts 1.8µs, the PPE could only add up to 13.248 tasks in the duration of one task. In this case, the PPE cannot keep up the addition rate with the tasks being executed.

In comparison with the results obtained for the StarSS programming model without hardware support in Figure 2.3, the graphs show a large improvement in the scalability of the system, specially for very fine-grained tasks. For a task size of 6.42µs (4.5µs
6.2. NEXUS EVALUATION

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU add task</td>
<td>420 cycles/task</td>
<td>7.5 tasks/µs</td>
</tr>
<tr>
<td>Descriptor Loader</td>
<td>8 cycles/task</td>
<td>400 tasks/µs</td>
</tr>
<tr>
<td>Descriptor Handler</td>
<td>68 cycles/task</td>
<td>47 tasks/µs</td>
</tr>
<tr>
<td>Finish handler</td>
<td>70 cycles/task</td>
<td>46 tasks/µs</td>
</tr>
</tbody>
</table>

Table 6.2: Average throughputs of life cycle stages in the SD benchmark.

for DMA and 1.92µs of computations), Nexus performs 7 times better than StarSS, as Nexus scalability is 13.45, and the StarSS scalability is 1.92, when 16 cores are available to execute tasks. When the total execution of a task is 15.74µs, the scalability of the StarSS is 3.4 using 16 cores. Instead, the Nexus system obtains a scalability of 14.1 for a task size of 15.74µs, which is an improvement of 4.15 with respect to StarSS. For larger task sizes, the Nexus system approaches to the maximum attainable scalability of 14.5 more quickly. This is due to a better performance for fine-grained tasks, which leads to an earlier and faster growth towards the theoretical value of the scalability. When fewer cores are available to execute tasks, Nexus also reaches a more scalable approach. If working with 8 cores, Nexus provides a scalability that is over 7 for a task size of 1.8µs, where only a fraction of the DMA transfers take place. For a smaller number of cores, they all reach the maximum scalability in this first test. In the case of a task size of 6.42µs, using 8 cores the scalability grows up to 7.68, remaining flat for tasks larger than 15.74µs around a value of 7.8. Instead, in the graphs from the StarSS test, it is visible that they do not perform well for small task sizes even when using only 2 cores. In accordance, with the mentioned task of 15.74µs, Nexus achieves an improvement in the scalability in relation to the StarSS of 4.16 when using 8 cores, 2.14 when using 4 cores, and 1.51 when using 2 cores. The faster growth towards the maximum scalability holds here as well when fewer cores are available. Therefore, the scalability enhancement is especially important for small tasks, where the overhead introduced from the runtime system of StarSS limits the performance of the system.

6.2.1.2 Scalability of the SD Benchmark

The SD benchmark is characterized by a simplified dependency pattern with respect to the CD benchmark. Its dependency pattern only establishes row-dependencies among tasks and therefore different rows can run concurrently. Hence, the ramping effects of the CD pattern are removed, and a maximum scalability of 16 is attainable when using 16 cores. Accordingly, in this case the dependency resolution is less demanding and the handlers should perform this operation faster. Table 6.2 presents the average latencies involved in the different pipeline stages. Similar to the previous benchmark, the task addition to the PPU was extracted from the execution trace, and the latencies related the descriptor loader and the two handlers were measured inside the simulator.

For this dependency pattern, the submission of the task to the TPU is performed faster than in the CD benchmark. The reason for this difference is in the preparation of the task descriptor, since now it has to write one dependency less. To load the task
into the in buffer, there is no difference, as both write the pointer to the task descriptor, so the size of the data loader remains the same. In the descriptor loader, there is no difference as it performs the same operations to request the task descriptor, regardless of the information contained in there. However, there is a difference in the latencies of the descriptor handler and the finished handler. This is undoubtedly due to the fewer dependencies encountered within the task descriptor. Since fewer operands must be handled, these handlers need less time to process each task. As the previous case, they both exhibit very similar latencies; although for this case the finish handler is slightly slower than the descriptor handler. In this case, due to the fewer operations, the latencies of both handlers are smaller than for the CD case. However, the finish handler still needs to carry out the deletion of finished tasks from the TPU. Although fewer deletions are done, it appears to be affected to a lesser extent than the dependency resolution process. Yet, the task addition is the bottleneck of the task life cycle, even though its latency has been decreased.

The scalability of the SD benchmark is depicted in Figure 6.3, depending on the task size and the number of available worker cores. The system response is very similar to the CD benchmark for small sizes of the tasks, strengthening the idea of the task addition to the TPU as the main bottleneck. Until the tasks are not larger than 4.5µs, when only

Figure 6.3: Scalability of Nexus for the SD benchmark.
DMA transfers are done, the scalability of both benchmarks is very similar. For larger task sizes, the scalability of this SD benchmark reaches higher values. The maximum scalability is 15.1 for this case, and it is achieved when the task sizes are larger than 23.64µs, similar to the CD case. This shows that once the task addition is no longer the bottleneck, the scalability offered by Nexus grows quickly with the task size. A surprising result from the graph is that when 8 cores are available to execute tasks, a higher scalability is obtained for a task size of 1.8µs than with 4.5µs. This might happen due to a better alignment of the SPEs executing tasks. For tasks with a size of 4.5µs, the task executed are only DMA transfers, and the length of these transfers depends on the number of dependencies of the task. This is due to the fact that blocks on the left edge of the matrix cannot depend on any other block. Thus, they do not have left neighbor and they only request one block, instead of the two blocks the other blocks need. On the other hand, for task sizes of 1.8µs, they only perform a simplified version of the DMA transfers. This simplified version is the same for every task. As a result, every SPE needs the same time to perform a task, while for tasks of 4.5µs there are some differences.

In comparison with the SD benchmark tested with the StarSS programming model, depicted in Figure 2.4, similar conclusions may be taken to the case of the CD benchmark, in terms of scalability. Again, this graph shows a faster approach to the maximum scalability for fine-grained tasks. For a task size of 6.42µs, we can see that the scalability of Nexus is 13.95 when using 16 cores. If we take a look at the StarSS graphs, for 16 cores this only is able to achieve a scalability of 1.74. Thus, for this first case, Nexus scales around 8 times better than the StarSS programming model by itself. When the task size grows, the relative improvement of Nexus over StarSS decreases and they both approach their scalability limit. Using Nexus, the maximum attainable scalability is 15.12 when the task size is 245.25µs, although it already reaches a value of 15 when the tasks are larger than 23.64µs. Nevertheless, in the case of StarSS, the maximum scalability obtained is only 14.5, when tasks last longer than 2372.2µs. As it occurred in the CD benchmark, adding the TPU to the processor operates very efficiently and solves the bottlenecks of the StarSS programming model.

6.2.1.3 Scalability of the ND Benchmark

The ND benchmarks establishes no dependencies among tasks, and thus its makes the application fully parallel, as every block can be processed in parallel with any other block. Thus, as the previous case, no ramping effect is found and the theoretical maximum scalability is again 16. In Table 6.3, the average latencies and throughputs of the different pipeline stages are depicted.

Once more, time spent in the task addition procedure to the TPU has been decreased. As it already happened in the SD benchmark in comparison to the CD benchmark, now the difference is again in the task descriptor preparation. Now each task descriptor has only one operand, as the blocks are independent on each other and thus only the address of the block processed is loaded into the descriptor. To time expent in writing into the in-buffer has no variation, as only the pointer of the task descriptor is written. The latency of the descriptor loader, as it has been already explained, remains a constant 8 cycles per task. For this benchmark, the latency of the handlers is smaller due the absence of
Pipeline stage | Register | Address  
--- | --- | ---  
TPU add task | 400 cycles/task | 8 tasks/µs  
Descriptor Loader | 8 cycles/task | 400 tasks/µs  
Descriptor Handler | 47 cycles/task | 68 tasks/µs  
Finish handler | 50 cycles/task | 64 tasks/µs  

Table 6.3: Average throughputs of life cycle stages in the ND benchmark.

dependencies. Every task goes through the descriptor handler without being written to any kick off list, but always being added to the ready queue. The finish handler must process each finished task and removes its entries from the consumers and producers tables. As no dependency is found, no tasks are added neither removed from any kick off list nor added to the ready queue. However, it still happens that two elements are mapped to the same table entry and thus the deletion process must be done. In this particular case, the difference between the latencies of the descriptor handler and the finish handler is larger than the previous case. This is due to the deletion not being affected by the dependency resolution and the descriptor handler only writing the info related to the task and adding them to the ready queue. The absence of dependencies among tasks affects the dependency resolution of both the descriptor handler and the finish handler. However, the deletion process in the finish handler is not affected by the dependencies among tasks, as it only depends on the results from the hash function.

Figure 6.4 shows the scalability offered by Nexus for the ND benchmark. It is visible that the scalability grows at a higher rate than in the two previous benchmarks, due to the absence of dependencies. The maximum attainable scalability for this benchmark using Nexus is 15.994 when the task size is 245.25 µs including the DMA transfers. This happens when using 16 cores and it means a very close approach to the theoretical limit of 16. However, we can see that the graph already reaches 15.87 with a task size of 15.74 µs, remaining almost flat for larger tasks. Thus, the performance of the system is not strongly influenced by task enlargement after such size. Similar to the case of the SD benchmark, the graph for 8 cores shows an unexpected behavior when only part of the DMA transfers are performed. In that particular case, it obtains a better scalability than the entire DMA transfers are processed. As now every task performs the same amount of DMAs, this idea presented in the SD benchmark results is no longer correct.

In comparison with the Figure 2.5 from StarSS, once more Nexus scales better than StarSS, specially for fine-grained tasks. When working with a task size of 6.42 µs, the hardware support is able to provide a scalability of 15.45, while StarSS can offer 1.5. This means that Nexus performs 10.3 better than StarSS in terms of scalability for this task size. When tasks grow up to 15.74 µs, the difference between the two systems decreases to 5.27, as the overhead of StarSS has a lower impact on the system performance. In this case, the scalability obtained by Nexus is already very close to the maximum attainable value and thus it cannot grow much. The performance gap continues decreasing when the tasks are larger, as the StarSS overhead has a smaller impact. However, Nexus can obtain a good scalability much quicker than the StarSS programming model. Accordingly, its response remains flat for task sizes larger than 23.64 µs, as for the case without the
6.2. NEXUS EVALUATION

Figure 6.4: Scalability of Nexus for the ND benchmark.

hardware support, task sizes of $166.5\mu s$ are necessary to obtain the flat response. In addition Nexus reaches a higher maximum scalability, as its limit is 15.994 while the limit of StarSS is 15.84.

6.2.2 Scalability for a Variable Task Size

In the three previous benchmarks, the evaluated task sizes have been set to fixed values. The real H.264 decoding algorithm however, shows different task sizes, which depend on several factors such as the block type or the operations performed with the data. The task size variation of the H.264 decoder has been measured in the real processor, obtaining that the task size follows a gaussian distribution. The mean value for this gaussian distribution has been measured to be $8.1\mu s$, while its variance has been measured to be around $1.8\mu s$. In order to model this behavior in the simulator implementation, within the SPEs each task size has been calculated using a gaussian distribution function that returns its variable latency. This benchmark implementation has been done using C++ language using its random number generator. However, current random number generator from C++ does not follow a normal or gaussian distribution, but an uniform distribution. Such type of distribution assigns the same probability to each possible number, which is not the case for the H.264. Therefore, we have used the ‘Law of
large numbers’ [19] to build our own gaussian distribution function. This law establishes that a gaussian distribution can be obtained as a combination of several independent identically distributed random variables. Using this law, our Gaussian distribution has been composed by a set of uniformly distributed random numbers provided by the C++ language. The sum of them is set as the task size, which is determined at every task execution. Different seeds have been used in each of the SPEs in order to have different size variation pattern in each of the cores.

In Figure 6.5, the execution of several task sizes has been measured. We have used the CD benchmark, as it has the most similar dependency pattern to the H.264 decoding algorithm, from which the reference mean task size and variance have been obtained. The size variation may affect the available parallelism during the execution of the application. As this parallelism limitation is more noticeable when the number of cores is higher, for this experiment we use the maximum number of available cores, which is 16. Two lines are drawn, one representing a fixed task size and the other one representing a variable task size. For the variable task size case, each task size of the graph represents the mean task size. The variance of each measured task size has been modified with respect to the 8.1µs case, but maintaining the original mean/variance ratio. The evaluated task sizes only correspond to the cases where the task is truly executed. Notice that testing

Figure 6.5: Scalability of Nexus for the CD benchmark and a variable task size.
the cases where only DMA transfers are performed would not produce any meaningful results, as there cannot be any variation in this times, and therefore the results would be very similar to those obtained in the fixed task size case.

The resulting scalabilities of the variable task size and fixed task size are depicted in Figure 6.5. In the graph, it is visible that the scalability obtained in the variable task size case is slightly lower than the case with a fixed value, and the difference between the fixed and the variable case grows with the task size. The reason for this behavior is the task size variation, which may interfere in the execution of the tasks. If the task length is fixed, a perfect alignment of task executions takes place, while for a variable size this may not occur. Due to the size variation, a task may take longer than its mean value. When this occurs, tasks depending on that task cannot run until current task has finished its execution. This situation limits the current available parallelism and therefore the scalability of the system. In our case, when the mean task size increases, its variance increases as well, and therefore a task may need even larger times to be processed. Accordingly, larger values slow down even more the system as tasks depending on the currently executed one must wait longer, resulting in a lower scalability.

6.2.3 Performance Improvement by Adding the PPE to the Execution of Tasks

So far, it has been assumed that only the SPEs were available to execute tasks, and the scalability has been determined only considering SPEs. Meanwhile the SPEs are executing tasks, the PPE is responsible to submit those tasks to the TPU. However, when the task table is full, the PPE stalls until a task has finished, releasing a table entry. During the time the PPE is waiting for new available space in the task table, it could execute ready tasks. Therefore, when the PPE finds the TPU full, it tries to retrieve a task from the ready queue to execute it. This improves the performance of the entire system, as it saves time in executing available tasks, and at the same time releases a task table entry. After the PPE executes the task, it continues adding tasks to the task pool. Moreover, once the PPE is done adding tasks, it is stalled until all the tasks have been processed. We can take advantage of this by the PPE also joining to execute tasks once it has added all the tasks to the TPU. Notice that for this case, the task table size is an important parameter, as a larger task table size means that the PPE starts executing tasks earlier.

In this case, we cannot measure the scalability improvement by including the PPE adding tasks. First of all, the architecture of the PPE is different than the architecture of the SPEs, as they are optimized towards different objectives. In addition, the PPE is meant to be the most powerful core in the architecture, as it has to run the operating system and control the SPEs. Furthermore, the PPE is not executing tasks in the same conditions as the SPEs. PPE only executes tasks whenever the task table is full, while SPEs execute tasks as long as there are ready tasks in the TPU. When the PPE tries to retrieve a task from the TPU, it may happen that even though the task table is full, there are not any ready tasks to be executed. In this particular case, the TPU warns the PPE about this condition, which stalls until it can add a new task. To study the effect of this modification, the CD benchmark has been run with different task sizes and
a variable number of SPEs.

The speedup obtained by adding the PPE to execute tasks, with respect to the case where only SPEs execute tasks, is depicted in Figure 6.6 and in Table 6.4. In this test, only the cases with some computations have been measured. If there is no computation, and since the PPE does not process any DMA transfer, the PPE would do nothing, while the SPEs performed the DMAs. Hence, there would be a difference of up to 4.5\ µs in case of DMA transfers to 0\ µs, which would be completely unfair. The results show that the speed-up obtained by the PPE adding tasks increases with the task size and decreases with the number of cores. The time spent in the PPE on each task addition to the TPU is independent in the number of cores. Due to the PPE executing task when it cannot
6.2. NEXUS EVALUATION

add a new task, the total time spent in adding task is thus constant and independent on the number of cores. Therefore, the addition of tasks from the PPE to the TPU can be considered as a constant overhead in the PPE. When fewer cores or larger task sizes are used, the total execution time of the system is extended while the overhead remains constant and hence, is less important. Having a larger total execution time reduces the ratio between overhead and execution time, and allows the PPE to be able to execute a higher number of tasks. Therefore, the impact of the PPE executing tasks is higher. A speedup of three is achieved when the task size has its largest value and only 1 SPE executes tasks. By extending the task size, this speed up will not grow significantly, as it is saturated around this value. In this extreme case, the overhead introduced by the task addition is very small in comparison with the task execution. By using 1 SPE and the PPE, the performance is as if three SPEs were processing tasks. Thus, this speedup of three means that the PPE is at least two times faster executing tasks than a single SPE.

6.2.4 Impact of TPU Latency in the Resulting Scalability

The latencies within the TPU have been set in the configuration of the block. So far, they have been set according to the operation they represent, given we are dealing with a hardware block. However, the influence of the latency on the behavior of the system is expected not to be deciding. As it has been shown in the previous tests, the throughput of the task pool is higher than the one found for the PPE adding tasks. In the most complex dependency pattern, the CD benchmark, the lowest throughput within the task pool is the descriptor handler. This is the worst case given these benchmarks, as the more complex dependency pattern, the bigger number of operations performed in the handlers. On the other hand, the PPE adding tasks does not notice a meaningful change, as different dependency patterns only influence on the number of parameters written into the task descriptor. In any case, this pipeline stage is still around 5.3 faster than the PPE adding the tasks on average. The finish handler also has a very similar average latency to the one characterizing the descriptor handler. If the dependency pattern is simpler, as for the SD and ND benchmarks, we can see that this difference is even higher. Hence, the CD benchmark has been chosen to study the impact of the TPU latencies on the scalability of the system. The task size has been selected in such a way that the PPE adding tasks does not limit the scalability and thus we can study when the TPU truly becomes the bottleneck. Accordingly, the task size for this experiment is $23.63\mu s$, where the scalability of the CD benchmark starts being flat for the initial latency.

Figure 6.7 shows the attained scalability for different latencies within the TPU. Each of the points plotted in the graph represents a multiple of the original latency of the TPU. Thus, the first measured point represents the initial latency configured for the previous tests, and the following points represent the number that latency has been multiplied to. This graph shows that for the first tested delays, the scalability of Nexus is not affected by the different latencies consumed in the handlers. However, when the latency reaches 50 times its initial value, the system performance experiences an impoverishment on its performance. For higher latencies, the scalability offered by Nexus continues dropping. As its illustrated in the graph, it is only slightly over 2 when its delay is 500 times the
initial tested value. The reason for this behavior is that the handlers do not become the bottleneck until their throughput cannot satisfy the SPEs executing tasks. Given the task size of 23.63µs, both the descriptor handler and the finish handler can process task at a higher rate than the demanded from the SPEs. In fact, when the latency is 10 times the initial value, it is the first time that the descriptor handler is the bottleneck instead the PPE adding tasks. However, this handler can process, on average, up to 4.44 tasks/µs, which is sufficient given the current task size. Hence, during the task execution of one task, the descriptor handler could issue up to 104.9 tasks, fulfilling the requirements of the 16 SPEs. In the next case, where the latencies are 50 times larger, it is where we may notice a slight deterioration of the scalability. For this larger latencies, the descriptor handler can issue up to 21.5 tasks during the execution of a task, which is already very close to the necessary rate when using 16 cores. For larger latencies, the throughput offered by the descriptor handler is not enough, and thus the performance of the system falls. This condition keeps worsening as long as the latencies within the TPU are larger, until a scalability of one is the maximum attainable value. However, as it has been already stated, the chosen latencies have been set to be reasonable, given that the TPU only accesses to its internal memory. Therefore, although some error could have been made in the latency estimation, this will not affect to the behavior of the system.
6.2. NEXUS EVALUATION

Additionally, through these results it has been demonstrated that increasing the size of the TPU would not influence strongly in the expected scalability. Notice that if the block is bigger, it may take longer to fetch elements from the internal element, and also consume extra time traveling across the linked list, as they could contain more linked elements.

6.2.5 Impact of Task Table Size in the Resulting Scalability

So far, the effect of different task sizes and TPU latencies in the scalability offered by Nexus has been studied. Another important feature of the TPU implementation is the size of the three tables used to handle the dependencies. Each entry of these tables contains valuable information that is used by the descriptor handler and the finish handler. Nonetheless, storing all this information bring along some drawbacks, as storing information requires memory, and thus it consumes area from the platform. On the other hand, it must be taken into account that this is meant to be a hardware block placed inside a multicore platform and thus, it should not consume large amounts of internal memory. Therefore, one of the design goals is to be capable of building this block using the least amount of area possible.

The size of the task table is an important parameter of the TPU implementation. It has been already stated in Section 5 that the size of producers table and the consumers table depend on this size, as it is defined to be 4 times the task table size. Additionally, task storage can be considered as the memory where the descriptors belonging to the tasks within the task table are stored. Hence, the size of the task storage also depends on the number of elements that can fit into the task table. However, the number of
elements that can be concurrently stored in the task table also influences in the scalability. Consider the case of the CD benchmark where every block depends on the top-right block and the left block. As depicted in Figure 6.8, in order to have a maximum scalability of 16 using the same number of cores, it is necessary to have all the tasks within the task table. Thus, we can define the nominal size as the case where the maximum available parallelism can be always exploited. In case only the first 8 lines are stored, this limits the maximum scalability, as TPU would not have been processed the next dependency-free task at this point. Therefore, there is a trade-off between area consumed and maximum attainable scalability.

In this test, different sizes of the task table inside the TPU have been simulated to study their performance. These sizes range from a task table size of 2048 elements down to only 32 elements. The matrix used in the test is the aforementioned one, with a size of $1024 \times 1024$ divided in $16 \times 16$ blocks, which leaves a matrix of $64 \times 64$ blocks. Thus, in the first case it is ensured that the first 32 lines of the block can fit in the TPU. This case is used for comparison with the other tests, as it is not possible to obtain a better performance with a different task table size. In the case only 32 elements are allocated within the TPU, this means than only half of the first line is added at the beginning of the simulation, and thus only 1 task can be executed at the same time.
6.2. NEXUS EVALUATION

Figure 6.10 shows the resulting performance for different task table sizes and number of cores. The performance of every task table size is measured in comparison to the performance obtained when using a task table with 2048 entries. Thus, the presented results mean the percentage of the maximum reachable scalability. As expected, the performance of the system decreases with smaller task table sizes. It can be noticed that when using 1024 entries in the task table, the performance obtained is exactly the same as the case of 2048 entries. For this particular case, there exists a perfect match at the beginning of the execution, as it can contain exactly 16 full lines of the matrix and the number of worker cores is 16 as well. As it has been explained, fewer entries in the task table means that some tasks that could run in parallel are not executed, as they have not been processed yet. Nonetheless, these results show that the performance does not decrease linearly with the reduction of the task table size. For instance, when 16 cores are executing tasks, a task table size of 512 does not produce a relative performance of 0.5 with respect to the case of 1024 entries. Since only 512 tasks can be stored in the task table, and thus be processed, this is equivalent to 8 full lines of the matrix. However, when a task has finished its execution, it is removed from the task table and its task descriptor from the task storage. Accordingly, given the dependency pattern of the CD benchmark, when a task within a line is being executing, the tasks placed at its left side have already finished, and thus removed from the table. Notice that in the CD benchmark, the block placed at the left of a block that is being processed must have finished its execution. Thereby, all the blocks on the left have been already processed, leaving some space for new tasks. These new tasks are allocated in the entries left by the finished tasks. For instance, considering the beginning of the execution of the matrix, with a task table size of 512, the first element of the 9th line is added to the task table, when the first block of the matrix have been processed.

Figure 6.10 shows an example with 16 cores and a task table size of 512 entries, although the behavior of a smaller task table size and fewer worker cores is very similar. The red squares represent the tasks that have been completed and removed from the TPU, the green squares are the tasks that are currently being processed in parallel, the blue squares are the elements inside the TPU waiting to be executed, and the non-colored squares are the tasks that have not been written to the TPU due to a lack of
Due to the ramping effect encountered in the CD benchmark, the maximum current scalability grows with the addition of task to the task pool. It starts adding the first task, and thus only this task can be executed. When second row is added, the first element of the second row is executed, and so on. Now, this example shows the first case where 8 tasks can run in parallel as the first 8 rows have been added to the TPU. As it has been stated before, a task table with 512 entries can contain 8 entire rows of the matrix. Thus, if no task were executed until the task table was full, the task table would contain the first 8 rows. Nevertheless, this is not the case, as due to the pipelined TPU, the tasks are processed and executed concurrently with the addition of tasks. Therefore, at this point, it is shown that besides the first 8 rows of the matrix, due to the finished task removal the TPU starts also including elements from the 9th row. In the case of a task table with 1024 elements, it would be ensured that the first 16 lines fit in the task table. Next step, when 9 tasks were available to run in parallel, the scalability given the task table of 512 entries is the same as the one obtained from a task table of 1024 entries. In both cases, only 9 tasks can be running in parallel. Therefore, the task table of 512 entries performs equally to the 1024 entries case until one of the available tasks has not been added yet to the TPU. This event takes place when there are 11 available tasks to be executed, although a task table with 512 entries does not have any of the blocks from row 11th, as illustrated in Figure 6.11.

From this point, the maximum scalability of the smaller task table cannot meet the scalability of the larger task table. However, the scalability in both cases tends to grow up to 16, as 16 cores are being used. Using the task table of 512 entries, it will experience a slower growth, although it eventually reaches the maximum scalability of 16. Therefore, it can be considered that a task table with 512 entries suffers a modified ramping effect. Such ramping effect lasts longer due to the mentioned lower growth rate for more than 10 tasks being executed in parallel in this case. Nonetheless, having a smaller task table changes the first case where the new limitation on the scalability is encountered. At this point, the ramping effect is no longer the problem, but the size of the task table gains importance. This first case depends on the task table size and the number of worker cores available for executing tasks. When the task table size is too short for the number of used cores, it happens that a momentary scalability equal to the
number of cores never takes place. As an example, if the task table size is only 32, its maximum scalability will never be 16.

6.3 Conclusions

The results obtained for the three synthetic benchmarks and different task sizes have showed that Nexus obtains the targeted scalability improvement. The performance of Nexus for fine-grained tasks has shown a higher scalability due to its acceleration of the dependency resolution and the synchronization with the worker cores. Therefore, the three bottlenecks found in StarSS have been alleviated. The PPE adding tasks has become the current bottleneck of the system. The performance improvement due to the PPE executing task while is not adding them has been also studied. Although for a small number of cores the speedup due to the PPE executing tasks reaches even a value of three, its impact has shown diminishing results for an increasing number of cores. The evaluation of latencies within the TPU has shown that its response is insensitive to latency increases up to fifty times the original value. The evaluation of different sizes for the internal tables of the TPU has showed that reducing the table sizes reduces the scalability of the system. By reducing the number of tasks within the TPU, we limit the maximum number of tasks that could run concurrently, and therefore we limit the maximum obtainable scalability. However, working with tables reduced to half of their nominal size, the scalability still remains over 90% with respect to its maximum value.
7.1 Conclusions

The goal of this thesis has been to implement, and study the performance of Nexus, a hardware support system built to enhance the scalability of task-based systems. The design goal of Nexus was to overcome the bottlenecks found in the StarSS programming model. The current implementation contains a centralized task pool, where the tasks are added from the PPE and retrieved by the SPEs and the PPE as well. The functionality of this centralized task pool is to address the dependency resolution among tasks, and the synchronization with the cores of the architecture.

This new hardware support has been implemented in CellSim, a simulator of the heterogeneous Cell BE architecture. In order to add the new hardware, the new module has been built following the UNISIM requirements, the simulator environment used to build CellSim. This is a modular simulator environment and hence, we have been able to plug Nexus to the existing architecture by using the rigorous and standardized communication protocol of UNISIM. Building and plugging the new module has allowed us to study the performance of the Cell BE architecture including Nexus.

The functionality of Nexus has been split into the register interface and the dependency manager, which contains the control logic and the tables involved in the dependency resolution. The register interface has been defined as the module’s gateway. It has been given the responsibility of synchronizing the task pool with the cores of the architecture. The dependency manager has been implemented separately, containing the control logic that handles the incoming tasks and the finished tasks. Fixed sizes have been set for the tables of the design, and a hash function with a collision resolution algorithm has been programmed to issue the mapping of a wide input range to a limited number of entries in the tables.

The results have shown that Nexus has reduced the bottlenecks found in the StarSS programming model. It has succeeded meeting the requirements for the hardware support, by accelerating the dependency resolution and the synchronization with the cores present in the platform. The results have illustrated the raise in the scalability of the system, in comparison with those obtained from the StarSS programming model. This behavior has been noticed to be outstanding for fine-grained tasks, as Nexus has demonstrated to scale up to 7 times more than StarSS for the CD benchmark. For less demanding dependency patterns, as the SD and the ND benchmarks, the Nexus results have shown even higher improvements on the scalability in comparison to StarSS. Additionally, an extra speedup has been observed when the PPE has executed tasks while it was idle. The impact of the PPE joining the execution has been noticed to grow for a small number of cores and large tasks. However, when the number of cores grows and the tasks are fine-grained, its improvement has been shown to be very slight.
The analysis of different latencies within the TPU has shown that the scalability of Nexus is slightly sensitive to latencies shorter than 50 times its current value. These results suggest that Nexus is capable to offer the same scalability in case additional features were introduced. Furthermore, although current latencies have been set to be reasonable, an error of their estimation would not imply a weakening in the system's response.

Interesting is the trade-off between the area cost and the performance of the system. The results obtained with the variation of the sizes in the internal blocks of the TPU have shown a decrease in the scalability when smaller tables were used. However, these results have shown that such decrease is rather small when the tested size is half of the optimal size for the task table. The optimal value is defined as the one allowing the maximum attainable scalability at any time. With the current configuration of the TPU, dividing the size of the task table by two would mean also to divide the producers table and the consumers table by two, which are 4 times the size of the task table. In addition, task storage size is decreased too, meaning this component occupies the largest area within the TPU. Therefore, these results showed that a significant amount of area can be saved at the cost of a little loss in the scalability.

7.2 Future Work

Some decisions have been made at the time of implementing the Nexus system. These decisions affected its capabilities and performance, trying to keep it simple and efficient, as this has been the first implementation of Nexus. Thus, some features might be added to complement the functionality of the TPU. Hereby some suggestions that can extend the capabilities or fill in some incompleteness within the current implementation:

- **Variable-size task descriptor.** The task descriptors used for this implementation have been set to a fixed size, which is able to contain 5 operands as explained in Chapter 5. Such a task descriptor structure is sufficient for the dependency patterns tested here, where most of the blocks hold the same type of dependencies. However, there might be different dependency patterns for which this solution would not be optimal. Although it is possible to extend the task descriptor with additional operands, this is a hardware block and once the task descriptor format is set, it cannot be changed. Therefore, the optimal solution is to define the task descriptors as structures with variable lengths. This would imply to write the size of the descriptor submitted into the in buffer, and include memory allocation mechanism inside the TPU. These memory allocation mechanisms would have to deal with the available memory in the task storage, and also with the consumers table and the producers table. These two tables would not have a fixed size any more, as a variable number of dependencies could be determined by the task descriptor. Hence, the descriptor loader would have to check the available memory space before processing a task from the in buffer. Including this memory allocation would mean an additional control logic involved, implying a larger size for this block and extra latency. On the other hand, the memory inside the TPU would be handled efficiently.
7.2. FUTURE WORK

- **Mutual exclusion mechanism for the producers table and the consumers table.** Although no mutual exclusion has been implemented in the producers table and the consumers table, this will be necessary in the real hardware. Therefore, either the proposed, or any other efficient mutual exclusion mechanism must be implemented to avoid race conditions.

- **Implementation of a more efficient hash function.** Several hash functions have been tested and the best approach in terms of performance has been selected. However, it is possible to define a new hash function that could map the inputs more efficiently to the table entries, given that it is known that input addresses are consecutive in memory. This would imply to investigate on hash functions and propose a solution that could better avoid the collisions to the same entry.

- **Implementation of TC.** The initial design of Nexus contained the TPU and an additional module called Task Controller (TC), which would be embedded within the SPU architecture. Currently, the SPEs fetch tasks from the TPU, process the DMA transfers with main memory, execute the task and signal back to the TPU that the task has been finished. Nonetheless, the SPEs are not exploiting their double-buffering capability. The SPEs can be processing DMAs related to some task at the same time they are executing a different task. Hence, the performance of the system could be further improved by exploiting this available parallelism within the SPEs. The TCs would be responsible of communicating with the TPU to retrieve ready tasks and signal back the finished task, as well as processing the DMAs. Meanwhile, the SPEs could be executing previously retrieved tasks at the same time the TCs would be taking care of the mentioned operations.

The implementation of the TC would need to handle memory allocation issues, as it would perform the DMA transfers. The TC would have to load the blocks needed to process the next task from the main memory, and store the result of the last executed task in the main memory as well. The communication with the TPU would be simple, as it would need only to perform reads and writes in the memory mapped registers of the TPU.

- **Definition of TPU clusters.** As the TPU entails a centralized approach for the dependency resolution, an increase in the number of cores will eventually lead to this module being a bottleneck. The solution proposed in Chapter 3 is to build clusters of TPUs, assigning each TPU to a set of SPEs and allowing task stealing among TPUs. In order to implement several TPUs in CellSim, they must be defined as UNISIM modules and connected to the existing architecture. This requires to assign a different address range and unique id to distinguish each of them. Although several TPUs have not been implemented in this work, some work has been already. The number of TPUs in the system has been defined, and the unique id is assigned to each of them. For the interconnection of these additional TPUs, it is only necessary to modify the cellsim.uni.cxx file, where the current instructions for the TPU interconnection must be included inside a loop. The routing table containing several TPUs can be obtained using the current function
for building the routing tables, as one of its parameters is the number of TPUs within the architecture.


