Design and implementation of an architecture-aware hardware runtime for heterogeneous systems

Master Thesis

for the award of the degree of

Master in Innovation and Research in Informatics (MIRI)

specialized in

High Performance Computing (HPC)

Juan Miguel de Haro Ruiz

(juan.deharoruiz@bsc.es)

Advisor

Carlos Álvarez Martínez

(calvarez@ac.upc.edu)

Co-advisor

Daniel Jiménez-González

(djimenez@ac.upc.edu)

Computer Architecture Department (DAC)

June 30, 2020

Abstract

In order to keep accelerating applications, it is a common trend to use heterogeneous systems with specialized hardware. They offer the best trade-off in performance and power consumption at the cost of programmability. Moreover, the number of cores in Symmetric Multiprocessors (SMP) architectures is increasing to keep up with the computation needs of emerging applications. As a result, handling such hardware accelerators and cores is becoming a challenge. Task-based programming models offer to the programmer an easy way to expose and exploit the parallelism of an application. A task is a unit of work which can be executed by a single thread on a processor core or an accelerator. The user can annotate tasks with input and output data requirements that can be used by the runtime to detect dependencies between tasks and establish a correct implicit task execution order. A software runtime is responsible to detect these dependencies to be able to ensure correctness and also exploit any existing parallelism based on the programmer's annotation in the application. The overhead introduced by this runtime becomes noticeable as the number of compute units increase or the task execution time becomes smaller.

To keep up with the number of cores/accelerators and speedup fine-grained parallelism in an efficient way, in this work we propose, design and implement Picos Daviu, a hardware dependence manager for task-based programming models. Picos Daviu proposal is able to handle task dependencies and determine which can be executed in parallel. First design has been implemented in SystemVerilog, and integrated to OmpSs@FPGA programming model, which provides a scheduler and a communication protocol to deliver tasks to hardware accelerators implemented in FPGAs. Picos Daviu has result in a mechanism to deal with distributed systems with FPGAs connected to the cloud and embedded FPGAs in a multicore chip. The autonomy of Picos Davius helps you to manage these systems without the need of a close and attached host.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Nowadays, single-thread performance in processors does not scale at the same rate as emerging applications need. To overcome this limitation, multicore architectures take advantage of the potential parallelism of applications. Symmetric MultiProcessor (SMP) architectures are a common solution in High Performance Computing (HPC), where multiple identical CPUs are connected together in a coherent platform with shared memory. Furthermore, heterogeneous systems are also used to reduce energy consumption. In such systems, multiple high performance but power hungry cores are put together with low power, reduced performance CPUs. Specialized hardware is also used in heterogeneous architectures, also called accelerators. They are used to perform faster operations than in the CPU, since they are designed to do a specific type of work. GPUs are an example, but also FPGAs or ASICs can be used to accelerate many kind of applications too. They offer the best performance and energy consumption trade-off at the cost of a higher difficulty and time to design such accelerators.

However, there are two main problems when handling SMPs or heterogeneous systems. On the one hand, the application needs to express in some way the parallelism that can be exploited by the hardware it runs on. This is a programmer work, and depending on the application it can be a difficult task. In addition, not all applications can use the full potential of the hardware due to dependencies between parts of the application. On the other hand, scaling the number of processors or accelerators to gain more performance is not a trivial task. For instance, as the number of cores increase in a SMP, it becomes harder to manage all these cores efficiently. Usually a thread inside a single core is responsible to decide where to send work. If there are too many cores, the throughput is not high enough and the time of each unit of work is low, the system is limited by this single thread and performance is degraded. Another problem is the communication latency, which increases also with the number of compute units in the system. In the typical master-slave approach, where a CPU sends work to other CPUs or accelerators, it may be a problem to communicate with such devices. For example, PCI express boards and cloud networks have a higher latency than cores inside a single chip. Therefore, to gain performance it is not viable to use the same communication protocol to exchange data or work between compute units.

1.1 Task based programming models

Parallel programming models are a solution to the first problem. In this work, we focus on task based programming models. They allow to express parallelism with units of work called tasks. These tasks contain code to be executed in a single compute unit like a CPU thread. Furthermore, dependencies can be attached to these tasks, allowing to establish an implicit execution order. These dependencies determine whether some data is produced or consumed by a task. Therefore, if a task produces some result and another task which was created after consumes it, the latter has to wait until the first finishes. This is calculated at runtime, which means that the programmer has only to declare the data dependencies of each task, and the runtime system will take care of discovering the parallelism. Any task which all its dependencies do not depend on any other task can be potentially executed in parallel. It's up to the runtime to exploit all the parallelism respecting the dependencies. Some of these programming models are OpenMP 5.0 [1] and OmpSs [2]. The first is an standard with tasking model support, whereas OmpSs is an academic initiative which serves as a forerunner of OpenMP.

Although these programming models solve one of the problems, the other remains unsolved. When there are many tasks which have a low execution time, a software runtime affects negatively to the performance. All dependencies among all running tasks have to be stored and compared (depending on the data structure some optimizations can be applied). The time it takes to solve the dependencies of a task and schedule it can be higher than the actual execution time of the task. Therefore, the only solution in this case is artificially making the task execution time higher, which is not always an option.

1.1.1 OmpSs

The OmpSs programming model is developed at the Barcelona Supercomputing Center (BSC). It integrates features from StarSs, another programming model, and it extends OpenMP by adding new directives. In OmpSs, there is a single thread which executes the main code and creates tasks. These are processed by a software runtime (Nanos++ [3]) and scheduled to another thread. Furthermore, it also supports the execution of tasks in heterogeneous architectures, such as GPUs (with CUDA), SMP clusters (with MPI) and FPGAs (with OmpSs@FPGA [4]).

Dependencies are declared over single memory pointers. With a directive the programmer can state that an address is an *in*, *out* or *inout* dependence. Tasks with *in* dependencies mean that the code will access at some point to that specific address for read-only operations. If the task performs write-only accesses, then it has to be declared as *out*, and *inout* in case it performs both. Nesting is supported, a task has the ability to create new tasks. This can help for some applications to have several threads creating tasks in parallel.

```
 1  #pragma omp task in(a[0], b[0]) inout(c[0])
 2  void matmulBlock(int bsize, float* a, float* b, float* c) { ... }
 3  void matmul(int msize, int bsize, float* a, float* b, float* c) {
 4      int b2size = bsize*bsize;
 5      for (int i = 0; i < msize/bsize; i++) {
 6          for (int j = 0; j < msize/bsize; j++) {
 7              int ci = j*b2size + i*bsize*msize;
 8              for (int k = 0; k < msize/bsize; k++) {
 9                  int ai = k*b2size + i*bsize*msize;
10                  int bi = j*b2size + k*bsize*msize;
11                  matmulBlock(bsize, &a[ai], &b[bi], &c[ci]);
12              }
13          }
14      }
15      #pragma omp taskwait
16  }
```

Listing 1: Matrix multiplication example with OmpSs tasks

Listing 1 shows a C code example of a matrix multiplication using OmpSs. The function *matmulBlock* implements a serial multiplication of a square matrix with fixed size *bsize*. The function *matmul* is executed only by the main thread, and it divides the input matrix in blocks and creates tasks which multiply those blocks. The *taskwait* pragma at line 15 is used to synchronize the main thread, which waits until all its created tasks have finished execution.

1.2 Hardware runtimes

The aim of this work is to design and implement Picos Daviu, a dependence management module of a hardware runtime for the OmpSs programming model. This design keeps track of all dependencies of the running application to determine which are ready and thus can be executed. By moving part of the work from a software program to a specialized hardware design, the runtime overheads are significantly reduced. Therefore, this design allows the system to increment the number of cores or accelerators, or reduce the task execution time without observing a degradation in performance.

This hardware runtime is intended to be used in multiple use cases. One of the main objectives is to attach it in a multicore chip, so it can manage all the CPUs to reduce runtime overheads and thread contention. Another objective is to implement it on a FPGA among other specialized accelerators. The runtime takes care of the tasks which go to the accelerators, and schedules them when there are multiple accelerators which can run the same task.

1.3 Objectives

- Integrate the hardware runtime Picos++ into the OmpSs@FPGA framework. As a consequence, add new features such as taskwait management, hardware task creation support and repeated dependencies per task (dependence merging) support.

- Develop a hardware dependence manager (Picos Daviu) based on Picos++ but with an improved design, prepared for different use cases. These are FPGA accelerators, SMP and distributed systems. In the last case, there are multiple instances of the hardware runtime which control the accelerators/cores of its own device.

- Integrate Picos Daviu into the Smart OmpSs Manager (SOM) runtime, which lacks the dependence management feature. Therefore, create Picos OmpSs Manager (POM), a hardware runtime with similar features as Picos++.

- Create a SMP simulator environment with 31 cores and one accelaror which creates tasks for the cores.

- Connect the simulated cores to Picos++ and POM and implement the system on a FPGA to get performance results.
- Develop a semi-perfect hardware runtime simulator to study the impact of the scheduling policy of both Picos++ and POM.
- Test POM in a SoC and discrete FPGA environments with different benchmarks, to compare against SOM or a software runtime.
- Test POM in a cloud environment where FPGAs are connected with a CPU through an internet network. Compare the performance over a software runtime and SMP-only impementations.
- Develop an alternative version of POM for embedded FPGAs with limited resources. Study the impact of the different configurable parameters of Picos Daviu using the SMP simulator.

1.4 Related publications

The work done in this thesis has lead to some related publications:

- J. M. de Haro Ruiz, J. B. Pons, D. Jiménez-González, and C. Álvarez Martínez, “Design and implementation of an architecture-aware hardware runtime for heterogeneous systems,”7th BSC SO Digital Doctoral Symposium, 2020. In this publication we introduce the Picos OmpSs Manager design and do a preliminary study and comparison with the Picos++ hardware runtime.
- C. González, J. Bosch, J. M. de Haro, A. Filgueras, M. Paolini, S. Balocco, C. Álvarez, and R. Pons, “Accelerating pp-distance algorithms on fpga using different strategies and runtime managers (under revision),”Future Generation Computer Systems, 2020. This work proposes and evaluates an FPGA implementation of the spectra application. In this study, they compare an OmpSs@FPGA implementation using Picos Daviu on a Xilinx Alveo board, and an Intel Stratix 10 FPGA with OpenCL. The results show that OmpSs@FPGA is more than two times faster than the OpenCL version.
- Juan Miguel de Haro, Daniel Jiménez González, Carlos Álvarez, ”Picos Daviu, a hardware task manager for the European Processor Inititative,” International Symposium on Field-

Programmable Gate Arrays (FPGA). Under Preparation. This paper summarizes the EPI evaluation results shown in section 5.3. It explains how to implement a hardware task manager that is able to manage up to 32 processors in a limited resources FPGA showing the trade-offs with different alternative designs.

1.5 Document organization

Chapter 2 introduces related work to this thesis, alongside a more detailed explanation of the OmpSs@FPGA framework and the Picos hardware runtime. Chapter 3 introduces Picos Daviu and explains extensively its internal design and functionality. Next chapter 4 presents the Picos OmpSs Manager, a hardware runtime using Picos Daviu and the OmpSs@FPGA scheduler. We also compare its internal structure with Picos++ and do a performance and area evaluation. In chapter 5 we do more performance studies of POM for different use cases, like SoC and discrete FPGAs, cloud FPGAs and embedded FPGAs in multiprocessor chips. For the last case, we also do a study to tune Picos Daviu configuration parameters and optimize area. Lastly, chapter 6 proposes some future work and concludes this thesis.

Chapter 2

Related work

2.1 OmpSs@FPGA

With OmpSs@FPGA [5][6] the programmer is able to execute OmpSs tasks (programmed in C/C++) on specialized hardware accelerators in FPGAs. The framework generates automatically hardware designs (in Verilog/VHDL) from the user code. This translation is possible thanks to High Level Synthesis (HLS) tools.

Currently, OmpSs@FPGA supports only Xilinx FPGAs. The accelerators are generated with Vivado HLS and C/C++ code generated by the Mercurium compiler from the original source code. Each task in the code is translated to a hardware accelerator, which can be replicated to execute multiple instances of the same task. To manage these accelerators, the OmpSs@FPGA tools include a hardware runtime, the Smart OmpSs Manager (SOM). It receives tasks from a queue filled by the CPU and transfers the necessary data to the accelerator. Listing 2 shows a C example of a matrix multiply implementation with OmpSs@FPGA. The pragma of line 1 specifies the device where to execute the task (FPGA accelerator), among other platform specific parameters. Line 2 shows the standard declaration of an OmpSs task with dependencies. In order to get an efficient hardware implementation, it is necessary to use the proper tools given by the vendor. Xilinx provides custom pragmas which allows to enhance the resulting hardware design, such as the *array_partition* or *pipeline* pragmas in lines 6,7,8 and 11.

In addition to manage accelerators, SOM supports hardware task creation [4]. Since it does not include a dependence management module, if the task created by an accelerator has dependencies, it is forwarded to the CPU. However, tasks without any dependencies can be executed immediately. In this case, SOM includes a scheduler which decides where to execute

```c
#pragma omp target device(fpga) num_instances(1) no_copy_deps \
    copy_in([B2SIZE]a, [B2SIZE]b) copy_inout([B2SIZE]c)
#pragma omp task inout(c[0])
void matmulBlock(const float *a, const float *b, float *c) {
#pragma HLS inline
#pragma HLS array_partition variable=a cyclic factor=FPGA_PWIDTH/64
#pragma HLS array_partition variable=b cyclic factor=BSIZE/MBLOCK_II
#pragma HLS array_partition variable=c cyclic factor=BSIZE/MBLOCK_II
   for (int k = 0; k < BSIZE; ++k) {
      for (int i = 0; i < BSIZE; ++i) {
         #pragma HLS pipeline II=MBLOCK_II
         for (int j = 0; j < BSIZE; ++j) {
            c[i*BSIZE + j] += a[i*BSIZE + k] * b[k*BSIZE + j];
         }
      }
   }
}
```

Listing 2: Matmul implementation with Vivado HLS and OmpSs@FPGA pragmas

that task. Figure 2.1 reveals the external interconnection with the accelerators and the CPU. Accelerators which create tasks have their own interconnection to send data when they are creating tasks, and another to receive a notification when all created tasks have finished. The CPU can send commands, which are used to execute ready tasks to a specific accelerator through the command in queue. When an accelerator finishes executing a task, it sends a notification to SOM, which will be forwarded to the CPU if needed through the command out queue. The spawn in and out queues have a similar behavior but with opposite directions. They are used to send tasks created inside the FPGA, either because it is a CPU task (it doesn't have an accelerator) or it has dependencies. Then the CPU sends a notification when a CPU task has finished, or when a FPGA task with dependencies is ready to be executed. In the last case SOM has to also notify back when the task finishes.

There is also related work to OmpSs@FPGA. Sommer et. al. [7] proposes a similar way to offload OpenMP tasks to hardware accelerators in FPGAs using HLS tools. However, hardware

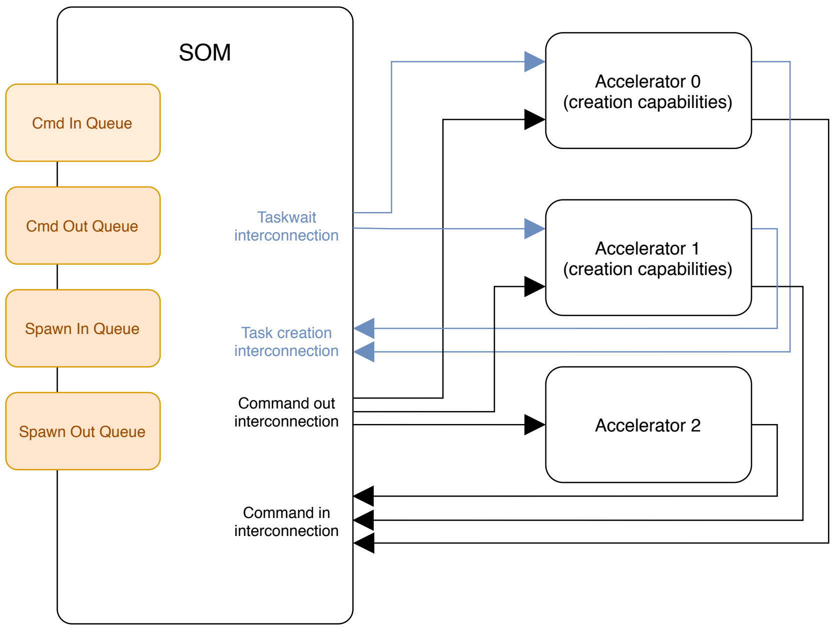

SOM
Cmd In Queue
Cmd Out Queue
Spawn In Queue
Spawn Out Queue
Taskwait interconnection
Task creation interconnection
Command out interconnection
Command in interconnection
Accelerator 0 (creation capabilities)
Accelerator 1 (creation capabilities)
Accelerator 2

Figure 2.1: Smart OmpSs Manager external interface

task creation is not supported. Xilinx supports implementation of OpenCL kernels in FPGA accelerators with the Vitis Platform [8]. The Vineyard project [9] offers a new ecosystem and programming model to program heterogeneous systems with FPGAs in data centers.

2.2 Hardware task managers

Carbon [10] is a hardware implementation of task queues. Its objective is to speedup dynamic task scheduling in a multiprocessor architecture, giving each thread a private queue. However, all tasks in the queues are assumed to be parallel.

Meenderick and Juurlink [11] performed a scalability study of the StarSs programming model for fine-grained parallelism in multicore architectures. They conclude that hardware acceleration of the runtime is necessary and propose the design of Nexus, a hardware support able to handle task dependencies. Etsion et al. [12] also propose a hardware design to speedup dependence management for the StarSs programming model.

Nexus# [13] is an improvement of the Nexus++ hardware runtime [14]. It is a VHDL implementation of a hardware dependence manager. Its main target is to work as an accelerator in a multicore platform, resolving dependencies of tasks created by the CPUs. It works with the OmpSs programming model, and it replaces the Nanos++ software runtime used by default.

The main improvement of Nexus# over Nexus and Nexus++ is the capacity of processing more than one dependence in parallel. It achieves this by storing multiple dependence graphs and distributing the incoming dependencies. Therefore, two dependencies with different addresses can be processed in parallel, reducing the time to process each task. However, it was only evaluated on a simulated environment.

2.2.1 Picos++

Picos [15] is another hardware runtime for task based programming models including dependence managing and implemented in VHDL. Its first objective was to manage dependencies of cores in a SMP. The main difference with Nexus# is that it was tested on a real system (with a FPGA) besides the simulator. However, although it was also designed to support storing many task graphs, the implemented version and its successors only have one. Picos is also used as a runtime for the OmpSs programming model, and was evaluated on a FPGA inside a SoC with ARM cores [16]. In addition, it was implemented on a RISC-V multicore environment on a FPGA [17]. In the latter case, Picos is attached directly to the CPUs and they communicate through special instructions extending the RISC-V ISA.

The final version, Picos++ [18] is implemented as a hardware runtime for accelerators. Like SOM, it manages hardware tasks with the OmpSs programming model. The main difference with OmpSs@FPGA is that all dependencies are managed by Picos on the FPGA, and it schedules these tasks in the accelerators and even on the SMP (if the task has both hardware and software implementations). However, task creation on hardware accelerators is not supported in Picos++. It does however support nesting, which is not supported in the first version of Picos. In that case only the main thread can create tasks.

The internal components and its connection to the CPU are shown in figure 2.2. Tasks created by the application running on the cores are captured by a software runtime (a modified version of Nanos++). This runtime is much lighter than the default of OmpSs, since it just puts the task in a shared queue stored in main memory, called new task buffer. Picos fetches this queue, processes the dependencies and executes tasks on the accelerators or puts them in the ready task buffer if they are scheduled for the SMP. This queue is also used to notify the software runtime that a hardware task has finished execution. For SMP tasks, the CPU puts the notification in the finished task buffer. The bypass buffer serves two purposes. First, it allows the software to control the accelerators when the tasks have no dependencies, or they are handled by a software runtime. Lastly, it solves a deadlock problem caused by the limited

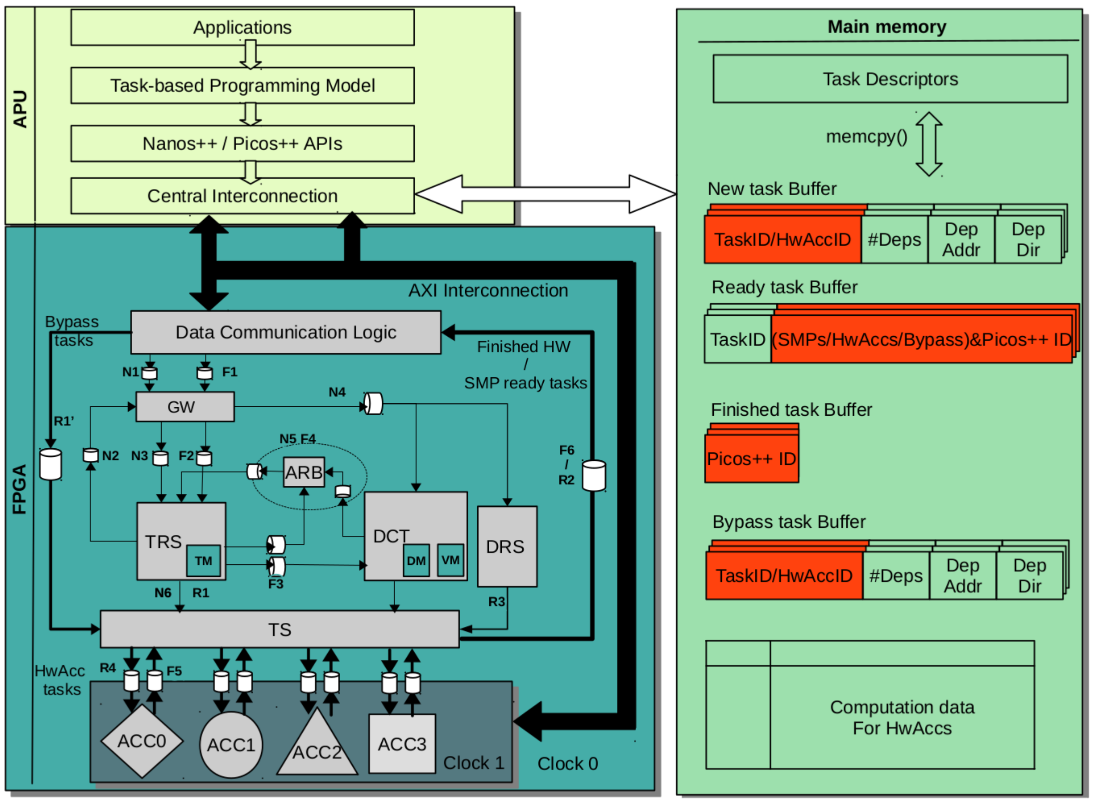


Figure 2.2: Picos++ system overview

internal memory of Picos++. This scenario is explained in section 4.1.1.

2.2.2 Picos++ integration with OmpSs@FPGA

One of the main drawbacks of the Picos++ ecosystem is the specialized software it needs to make it work with OmpSs. Even though it uses a similar architecture of the OmpSs@FPGA model, Picos relies on a custom API which communicates with the Nanos++ software runtime. It was developed in our OmpSs@FPGA group, which already maintains other custom APIs like Xtasks and Xdma to communicate with SOM. Furthermore, OmpSs@FPGA uses an automatic tool, AIT, to generate the FPGA bitstream directly from the C/C++ code. Nevertheless, Picos++ projects have to be handcrafted, which is a time consuming work and limits the amount of applications running with it.

Therefore, in order to take advantage of the OmpSs@FPGA automatic toolchain and enhance maintability of the code, we adapted the Picos++ external interface to match SOM's. This is the first contribution to this work prior to the development of Picos Daviu.

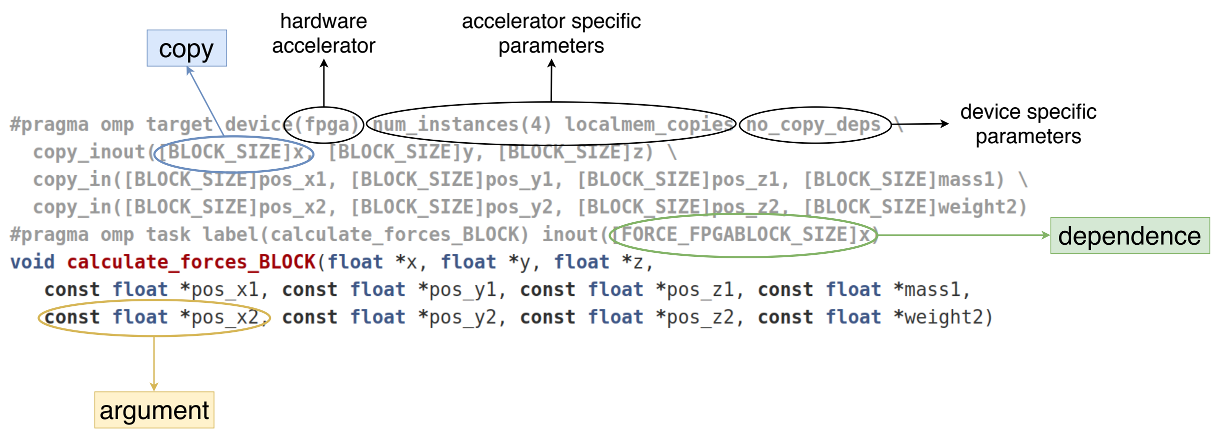
External interface

As figure 2.2 reveals, all external queues of Picos++ are stored in main memory. However, the memory requirements of these buffers is not very high, so moving them to local SRAMs (Xilinx Block RAMs or BRAMs) improves the performance at a very low area cost. Since SOM uses this system too, Picos++ can use the same protocol to exchange data with the CPU.

Another aspect to adapt is the format of the data inside the buffers. The main difference is the width of each data word, Picos++ uses 32-bit whereas SOM uses 64-bit words. Another important change is how a task is described. Picos++ task descriptros are formed by:

- Task ID: A unique identifier generated by the software runtime or Picos.
- Architecture mask: A 16-bit mask, each bit determines if the task can be executed by a specific accelerator. Bit zero represents the SMP, so a total of 15 hardware accelerators are supported in Picos++.
- Dependencies: Memory addresses pointing to a data element used by the task.
- Dependencies direction: Specifies how the data element pointed by a dependence is used by the task. It can be either *in*, *out* or *inout*.

This limits the capabilities of the programming model, since it implies that each argument of a task needs to be a dependence. For instance, literal or scalar values are not supported in Picos++. On the contrary, OmpSs@FPGA task descriptors do not have this limitation. There is a distinction between three types of information inside a task. They are dependencies, arguments and copies. Figure 2.3 shows a code example of how they are specified in C code, among other specific parameters of the programming model. Dependencies have the same meaning as in Picos++. However, they can be decoupled from the arguments, which are the function parameters of a task declaration. Copies are a new concept not present in SMP architectures. In systems with external devices, which have their own memory, data accessed by a task targeted in the device has to be present in its memory before executing the task. Copies allow to specify which regions of memory have to be transferred to device memory prior task execution. For FPGA SoCs, where the memory is shared, this concept of copies is also required. Instead of moving data from one memory to another, it has to be copied in the same memory. This is done to ensure the data is contiguous and pinned in physical memory, since the OS could allocate physical pages in non-contiguous segments of memory.



```
#pragma omp target device(fpga) num_instances(4) localmem_copies no_copy_deps \
  copy_inout([BLOCK_SIZE]x, [BLOCK_SIZE]y, [BLOCK_SIZE]z) \
  copy_in([BLOCK_SIZE]pos_x1, [BLOCK_SIZE]pos_y1, [BLOCK_SIZE]pos_z1, [BLOCK_SIZE]mass1) \
  copy_in([BLOCK_SIZE]pos_x2, [BLOCK_SIZE]pos_y2, [BLOCK_SIZE]pos_z2, [BLOCK_SIZE]weight2)
#pragma omp task label(calculate_forces_BLOCK) inout([FORCE_FPGABLOCK_SIZE]x)
void calculate_forces_BLOCK(float *x, float *y, float *z,
   const float *pos_x1, const float *pos_y1, const float *pos_z1, const float *mass1,
   const float *pos_x2, const float *pos_y2, const float *pos_z2, const float *weight2)
```

Figure 2.3: OmpSs@FPGA task declaration

In order to integrate arguments and copies in Picos++, we had to modify the original design. One of the main modules inside Picos is the Data Reservation Station (DRS), shown in figure 2.2. It includes a local SRAM to store dependencies, so they can be forwarded to the accelerators fast. It was adapted to store arguments instead. Furthermore, a new memory was required to store copy information. This data is not needed for hardware tasks, since it is assumed that when a new task reaches Picos, all the task data is already in memory. However, tasks with SMP implementatin require copy information to send back data from FPGA memory to CPU memory. This introduced a new problem due to the extra area required to implement these memories. In Xilinx FPGAs, they consume more than 50% of the BRAMs of Picos++. They are designed to have enough space to store the maximum number of copies for all tasks. Moreover, they need three times more space than arguments. A copy is made of a 64-bit address plus the region size, offset, and other control flags. The solution was to add a parameter to control the maximum number of copies and arguments per task.

Hardware task creation

To fully integrate Picos in OmpSs@FPGA, it has to be adapted to accept task creation in hardware accelerators. To simplify the system, we decided to delete the new task buffer and replace it with the task creation interconnection of SOM. The objective is to be able to replace an accelerator with any device, like the SMP. This change also implied to add the taskwait mechanism in Picos++. To do that, there is a register file which stores the number of created and finished tasks for each accelerator. When the accelerator asks for a taskwait, it blocks until the two registers match. In that case, Picos++ sends a notification to resume execution.

Chapter 3

Picos Daviu

3.1 Origin

There are three main problems with Picos++. The first one is its limited capability to be configured. Although for the tested FPGAs its size is small enough, on the one hand it does not contemplate a more extreme case where the resources are limited (small FPGAs). On the other hand, it sets a hard limit on how many accelerators/cores the runtime can handle. Currently Picos++ can only manage up to 15 devices, and its internal memories are sized to match this number. This is a clear limitation which does not allow the runtime to scale as the number of compute units increases. Furthermore, it could help to reduce unnecessary resources and replace them with other hardware which does useful work.

The second problem is related to its internal design. The internal connectivity and resources prevented designs to reach its theoretical peak performance. This means that if Picos++ is instantiated in the system, the route and place algorithms struggle to find a solution with the expected performance. Therefore, designs without Picos++ could reach higher resource usage by adding more hardware accelerators or increasing frequency. We believe this is caused by the complexity of the internal connectivity of Picos++. So, for some applications where Picos++ is supposed to gain performance, in the end it is not true since it cannot make use of the same amount of accelerators with the same frequency.

Lastly, there is a problem to develop and maintain its code. Fixing bugs or adapting the design to our needs is difficult, and takes more time than necessary because the original developer is no longer in our group.

Our decision was to make another version of Picos, avoiding these problems. First, we would make use of parameterizable memories and other features from the beginning. Also, another key idea was to make it as simple as possible. Picos++ uses many queues to send data between the internal modules and to the accelerators. While they can be useful in some cases to avoid blocking, they use too many resources. Also, Picos++ uses a complex search system for address matching. It is capable of finding a dependence address in its internal memory in a fixed amount of cycles. However, this design requires a lot of complex logic, auxiliary memories and it is very susceptible to stalls due to lack of memory. When this happens, the runtime waits for several tasks to finish, which can lead to significant performance degradation. We decided to use a more simple data structure to implement the search algorithm, which requires more cycles to find a match but it doesn't stall as much. Moreover, when some internal memory is full, Picos Daviu only has to wait for the first task to finish, while Picos++ would wait for an undefined amount of tasks, potentially all of them in the worst case.

To ease maintainability and have a more unified ecosystem inside OmpSs@FPGA framework, we decided to implement only the dependence management module. The scheduler, taskwait system and communication queues are already implemented in SOM and are simple enough to be integrated with Picos. This is the main contribution to this work, called Picos Daviu, which is an evolution of the previous Picos++ system. Its internal architecture is inspired in the latter, but it was designed from scratch. This new architecture is not functional by itself and it needs features of SOM. Therefore, we created an alternative version of the OmpSs@FPGA hardware runtime which has the same capabilities as SOM with dependencies management. We called it Picos OmpSs Manager (POM).

3.2 Internal design

Picos Daviu is build with several independent modules coded in SystemVerilog. Each one serves a different purpose and communicates with other modules through different types of interfaces. These are illustrated in figure 3.1, which shows the internal structure of Picos Daviu. There are four main modules, the gateway (GW), task reservation station (TRS), dependence chain tracker (DCT) and ready task dispatcher (RTD). The entry point for new tasks is the gateway. It receives the task descriptor and splits the data between memories to store temporal data, the TRS and the DCT. The first handles and stores task related information, whereas the latter treats only dependencies and determines when they are ready. The DCT notifies the TRS for each dependence of every task when it becomes free. The TRS notifies the RTD when a task is ready through a FIFO, and it sends the required data through an external interface. When this task finishes executing, it is notified directly to the TRS through the finish task external interface. For each dependence of this task, the TRS notifies the DCT so it can free resources related to that dependence and potentially free dependencies of other tasks.

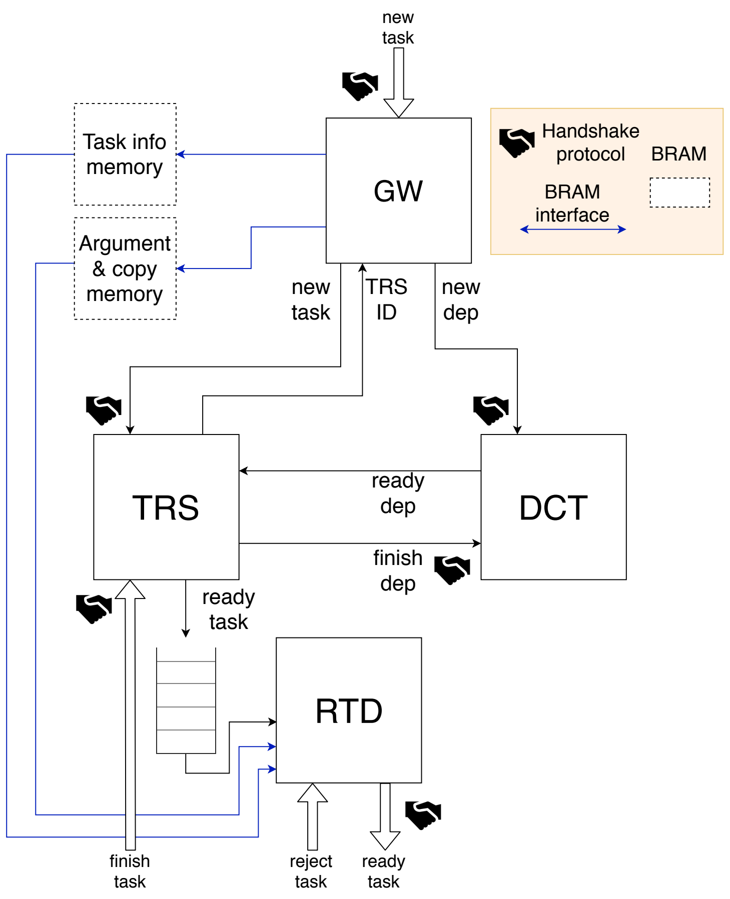


Figure 3.1: Picos Daviu internal modules

3.2.1 Gateway

The gateway takes care of splitting a task descriptor and sending the relevant data to each module. A task descriptor is formed by:

- Number of arguments, dependencies and copies.
- Parent task identifier: Unique identifier of the task that created this task, or 0 if it wasn't created by any other task (created directly from the main thread).
- Task type: A unique identifier of the task type, which represents the function that was called in the source code. Any two tasks with the same task type have the same number of

arguments, dependencies and copies since they do the same computation. It also includes the architecture were this task can be executed, currently being only CPU, FPGA or both.

- List of dependencies.
- List of copies.
- List of arguments.

The task descriptor is transferred to the gateway through a 64-bit AXI-stream interface. The gateway does not know if the internal memories of Picos are full, so it relies instead on the module that sends the task. Before starting any transaction, the AXI master must check first if there is enough space to store it. To give this information, there is an output signal that is asserted when any of the memories inside Picos can't hold the biggest task possible (with the current configuration).

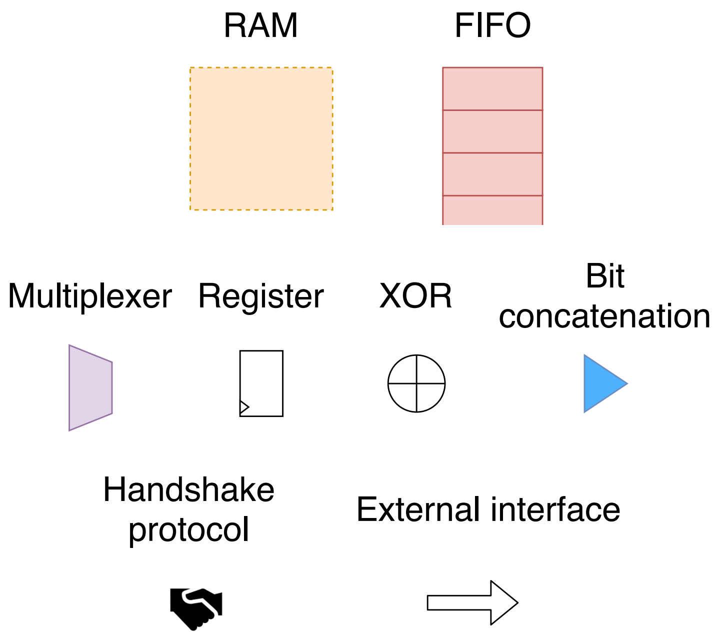


Figure 3.2: Legend for the figures of chapter 3

The number of arguments, dependencies and copies are used to know how many words have to be read for each list in the stream. They are stored in a memory called Task Info Memory, which also stores the task type and parent task identifier of the task. This information is not relevant for Picos, so it just stores this data until the task is ready to be executed. The arguments and copies are also stored in different memories to be forwarded outside later. In order to know in which address all this data is stored, the gateway asks for a unique identifier of the task to the Task Reservation Station (TRS ID). The range of this ID starts at 0 up to the number of tasks parameter. The size of the Task Info Memory depends only on the number of tasks parameter. On the other hand, the argument and copy memories need to have enough space to store all possible tasks with the maximum number of arguments and copies. Therefore, the size of these memories is the number of tasks times max number of arguments/copies. The latter is rounded to the next power of two to simplify address generation (concatenation of the TRS ID with the argument/copy ID), thus it is not viable to use a big number of arguments/copies per task as a default limit. To minimize memory usage, a good option is to adapt these parameters to the applications that are going to be run with Picos.

Another job of the gateway is to provide input to the Task Reservation Station (TRS) and the Dependence Chain Tracker (DCT). The first stores information for tasks that are not yet ready to execute or being executed. Specifically, it requires the number of dependencies. The DCT keeps track of the dependencies and notifies when a dependence is free. Therefore, its input consists on dependence addresses of a task. The gateway also provides the TRS ID and the dependence ID in order to uniquely identify that dependence. The dependence ID is used to distinguish all dependencies of the same task, which have the same TRS ID.

The internal design of the gateway is shown in figure 3.3. It only includes the data path and memory addresses. The control signals (memory and register enables, etc.) and the Finite State Machine (FSM) which controls the module are omitted for readability purposes. A legend for this figure and others in this chapter is available in figure 3.2.

CAM FIFO

One restriction of the DCT is that dependencies of the same task cannot be repeated. This is because the DCT does not check the TRS ID when comparing addresses. Furthermore, this restriction helps to simplify the logic dramatically. Even though the DCT does not allow repeated dependencies on the same task, the programming model does allow it, thus some preprocessing has to be done. The gateway counts with a CAM FIFO to do so. The size of this FIFO depends on the maximum number of dependencies, and it compares all its addresses with the one in the input stream. If the input dependence address matches with any other in the FIFO, the direction of both dependencies are merged. If both dependencies are in, the resulting dependence is in, if both are out, the resulting is also out. If one is in and the other out, the

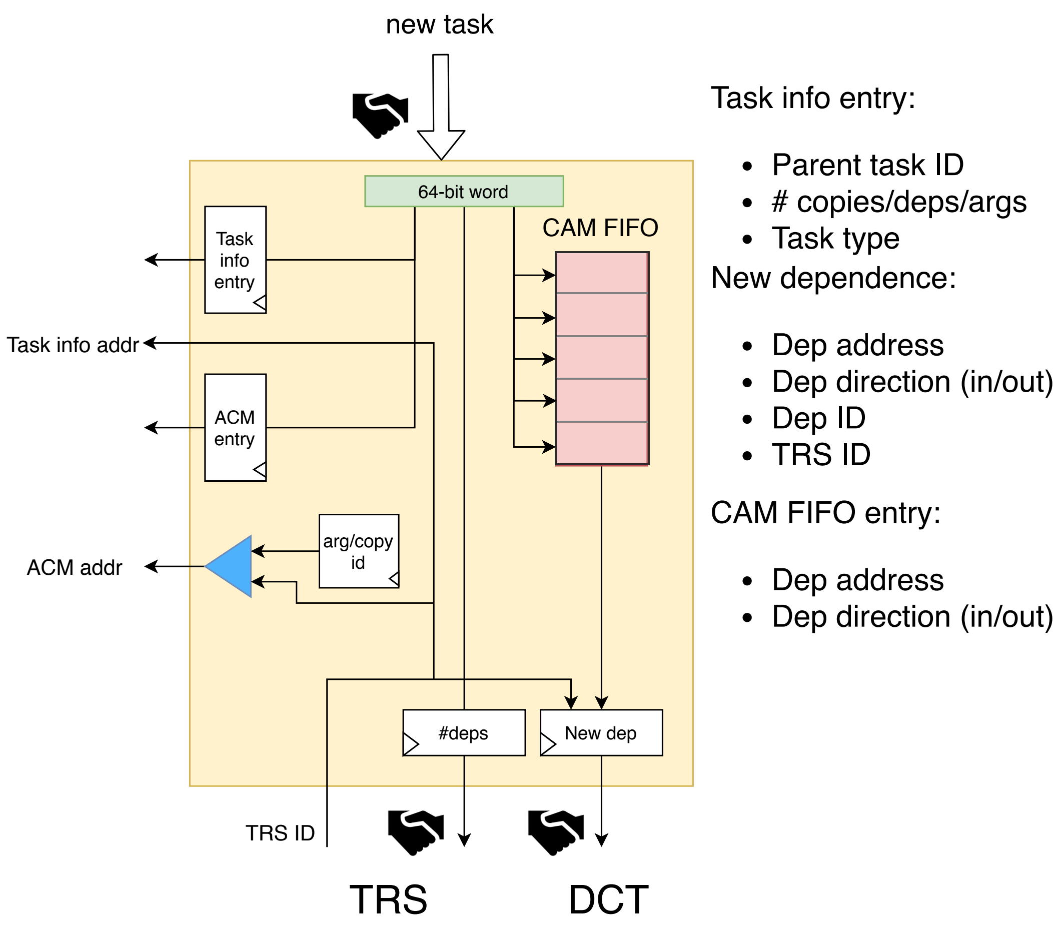


Figure 3.3: Gateway internal design

result is inout. On the other hand, if the new address does not match, it is pushed into the FIFO. This method allows to read new dependencies each cycle without introducing any delay to compare, at the cost of area. After doing this preprocessing the number of dependencies may be not the same as the task provided, so the gateway waits until all dependencies have been read to store the number of dependencies and sending it to the TRS.

3.2.2 Task Reservation Station (TRS)

The TRS stores task related data. It keeps track of the number of dependencies that are not ready, thus depending on at least one dependency of any other task stored in Picos. Initially this equals to the total number of dependencies provided by the gateway. The TRS also uses the same ID it sends to the gateway as address to access all its internal memories. The number of not ready dependencies is decremented each time the DCT notifies that a dependence is ready.

It provides the TRS ID of the task, so the TRS can update the correct counter. When it reaches zero, the task is ready to execute because none of its dependencies depend on any other task. In this case, it pushes the TRS ID of the ready task to a queue that can store all possible tasks.

There are three main jobs the TRS is responsible to do. First, it has to allocate a new TRS ID and store the number of dependencies of a new task incoming from the gateway. It also has to update the counter when a dependence is ready, notified by the DCT. Finally, it has to free a TRS ID and send a notification to the DCT for each dependence of a task that has finished. To store this information it has a different memory which stores a DCT memory pointer for each possible dependence of a task. The meaning of this pointer is introduced in the next section. This memory and the one to store the number of dependencies form the Task Memory (TM), the size of which is configurable and determines how many in-flight tasks can Picos hold.

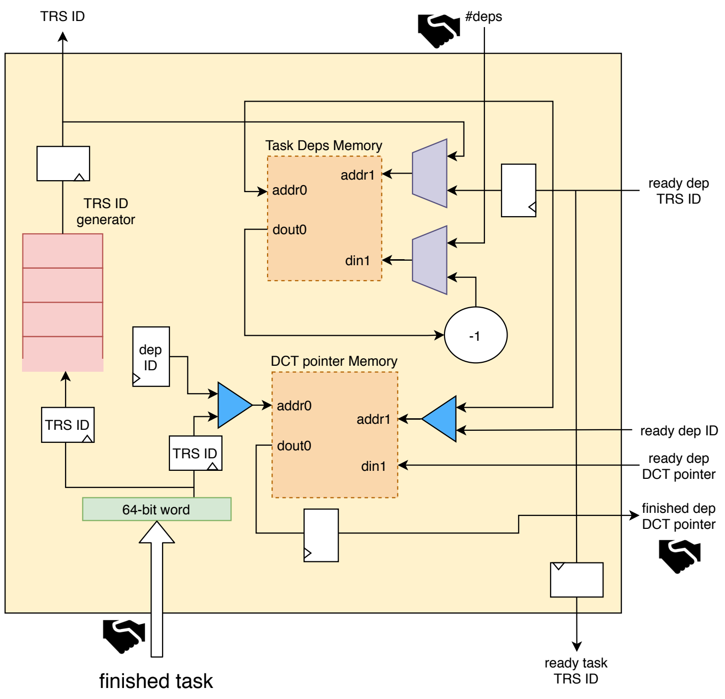


Figure 3.4: TRS internal design

Figure 3.4 introduces the internal design of the TRS. It does not show control related signals nor the FSMs for simplicity. The memories used for the design have two ports, but updating the dependence counter already needs both of them (one to read and the other to write the result updated). Therefore, there is a multiplexer in the write port to write also the number of dependencies of a new task. While the DCT is busy sending ready dependencies to the TRS, it cannot allow new dependencies inside, so the gateway waits. Thus, this is a critical part and it should be done fast. That's why there is no handshake between the DCT and the TRS. In return, the gateway and the TRS need one because the first has to wait until the write port is free from the DCT.

TRS ID generator

The generation of IDs is implemented with a combination of a FIFO and a counter. The FIFO stores the IDs that have been released because the task related to that ID finished. Since it is initially empty, the ID generator uses also a counter. It gives identifiers up to the configured number of tasks, and once it overflows the FIFO is used instead to give the IDs.

3.2.3 Dependence Chain Tracker (DCT)

The DCT is the most complex and critical component of Picos. It has to implement the main dependence tracking algorithm, and stores all dependencies from all tasks in Picos. To do that, it makes use of two memories:

- Dependence Memory (DM): Here is where the address matching takes place. This memory stores a single entry for each dependence address that appeared at least once in any task.
- Version Memory (VM): This memory can have several entries for the same address. It stores a dependence chain for each address in order to know the order in which they have to be marked as ready.

The DM is used to know whether an incoming address already appeared for any previous task. Since the objective is to have a relatively big memory, we have to use a data structure that allows to do a fast search. Picos++ uses a different structure to implement the tracking algorithm. It counts with a set associative cache-like structure, where each set contains eight ways. A hashing algorithm calculates the set to search the incoming dependence address. Then, the DCT compares each way in parallel to have the matching in only one cycle. Although this is fast, there is a risk. If more than eight dependencies are mapped to the same set, the DCT has to stall since none of the entries can be evicted. Therefore, this system relies on the capacity of the hashing function to distribute all addresses among all sets. To make a better design we decided to avoid this risk and use a data structure which permits utilizing the whole memory.

DM as a binary tree

The first idea for the data structure of the DM was a binary tree. This tree is sorted, on the right child are addresses bigger than the address on the root, and on the left child are the smaller addresses. The tree is not auto-balanced, which means that in the worst case (all incoming addresses are sorted) all nodes only have one child, thus behaving like a list. But in other cases, it helps to reduce the search space because only one of the children is chosen to explore the tree, and the other sub-tree is consequently discarded. In the mean case, the search time is logarithmic to the tree size. The auto-balancing feature is not implemented due to its high complexity and use of resources in hardware.

DM as a linked list

We also explored the possibility of a simpler data structure. This reduces design complexity and area, which can help to improve timing also in extreme cases. The simplest structure is a linked list, where each node has a pointer to its next and previous element. The downside of this structure is that searching in a list takes linear time, since the algorithm has to traverse the whole list until it finds (or not) the incoming address.

Hash table

In both proposed data structures, it is useful to use a hash table. This is an extra memory which stores pointers to a different binary tree or linked list. It is a way to distribute the addresses between multiple data structures. It implies applying a hash function to the incoming address to get an entry to the table and thus, the tree or list.

There are many hash functions, each providing a different set of properties. Our main requisite is that this function should distribute evenly the addresses between all the entries in the table. Also it should use low resources and be fast to compute. A function that fits these requirements is the Pearson hash, taken from Picos++.

This function uses a table T with all possible values of a given number of bits B, stored in random locations. Then, we apply the code in listing 3.1. The output h is a B-bit word, made up of a chain of XOR operations with elements of the table and a portion of the input. This method is really simple and since it has a random component it helps to distribute the addresses independently of their values.

The only problem of this method is that the loop is sequential because the address of the

Listing 3.1: Pearson Hash pseudocode

```
pearson_hash(C):
    h = 0
    for c in C:
        h = T[h xor c]
    return h
```

Listing 3.2: Modified Pearson Hash pseudocode

```
modified_pearson_hash(C):
    h = 0
    for c in C:
        h = h xor T[c]
    return h
```

table is calculated on the previous iteration. Since the memories in Picos have a one cycle latency to read, this implies that only one XOR can be done each cycle. In order to speed up the algorithm, we modified this method to remove the dependency of the address calculation. The modified function is in listing 3.2. With this small modification, the function can be applied in one cycle since all the XOR operations can be calculated in a single cycle. The implemented hardware design is in figure 3.5. Although the addresses inside Picos are 64-bit, we can't use all bits for the hashing function because the table has only 64 elements (6 address bits).

Version Memory

Before further explaining the role of the Version Memory, we first clarify some concepts about task and dependence states. Each task and dependence has three main states in its lifetime inside Picos:

- Not ready: The task/dependendence does depend on at least one other task/dependence.
- Ready: The task/dependence does not depend on anything else.
- Finished: The task finished its execution. In the case of dependencies, the related task of this dependence finished its execution.

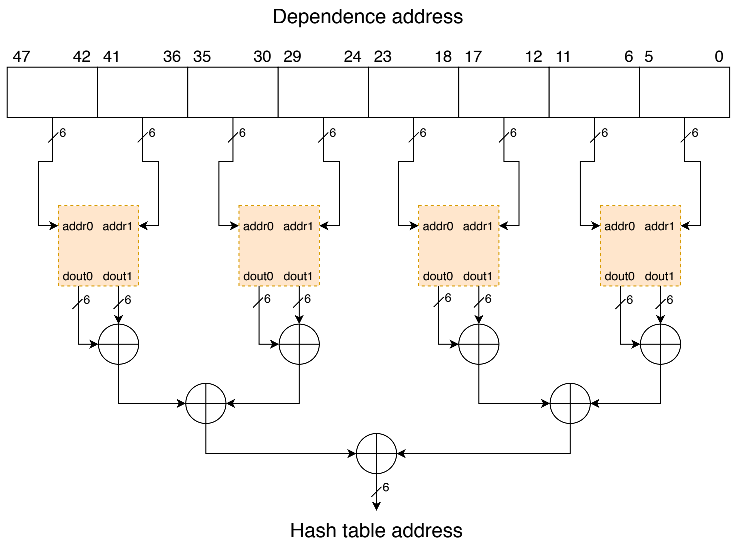


Figure 3.5: Modified Pearson hash function hardware design

This memory stores chains of dependencies with the same address either ready or not, depending on the order in which they arrived and their direction. The latter can be either *in* or *out*, *inout* dependencies are treated as *out* since they have the same effect in the DCT. We distinguish between three types of chains:

- *out-out*: This chain stores dependencies which are *out*. Only the first dependence in the chain may be ready.
- *in-in*: This chain stores dependencies which are *in*. All dependencies in this chain are ready.
- *out-in-in*: This chain is made of *in* dependencies and only one *out*. The *in* dependencies in this chain are never ready, although the *out* could either be ready or not.

When a new address appears for the first time in the DM, a new entry in the VM is allocated. If the dependence is *in*, an *in-in* chain is created and the dependence is sent immediately to the TRS and marked as ready. If the dependence is *out*, an *out-out* chain is created instead and sent to the TRS too.

If the address was already present in the DM, there are a few cases to consider. The DM always points to the last entry of an *out-out* chain, or an *in-in* chain in case there are no *out* dependencies. In the case of an *out-out*, if the incoming dependence is *in*, it is marked as not ready and inserted into the *out-in-in* chain started with the *out* dependence. In case it is the first one, a new *out-in-in* chain is created. If the new dependence is *out*, it is inserted in the *out-out* chain and marked as not ready. On the other hand, when the dependence found in the VM is part of an *in-in* chain, the new dependence is inserted into this chain and marked as ready if it is *in* (thus sent to the TRS). However, for *out* dependencies, a new *out-out* chain is created, but the dependence is not marked as ready since it has to wait for the *in-in* chain to finish its execution. In this peculiar case, the *in-in* chain points directly to the first element of the *out-out* chain.

When a task finishes, the TRS sends a notification for each dependence of that task. To identify the dependence, it sends a VM pointer to its related chain entry. This pointer is previously sent by the DCT when the dependence is marked as ready, and saved into the DCT pointer memory of the TRS. If the finished dependence was in an *out-out* chain, it has to be the first in the chain. Therefore, this entry is erased and if there is no *out-in-in* chain, the next dependence in the *out-out* chain is marked as ready. Nevertheless, if the finished entry is the first dependence in an *out-in-in* chain, all its dependencies are marked as ready and sent to the TRS. Then, all *in* entries in the chain are erased too, and a new *in-in* chain is created. This is possible because this type of chain can be compressed to a single slot in the VM. This slot is only a counter of the number of elements in the chain, with a pointer to a possible *out-out* chain. When an entry of an *in-in* chain finishes, its counter is decremented. If it reaches zero, the entire chain is deleted and the first element in a subsequent *out-out* chain is marked as ready, if existent. After deleting a dependence from the VM, if there are no more elements in the chain, the related DM entry can be released since this means that there is no task with a reference to that address. Section 3.3 explains in more detail the internal functionality of the DCT .

Figure 3.6 showcases a high-level schematic of the data structures and chains of the DCT. There is a new dependence address (0x03), which is used to apply the hash function to get an entry to the hash table. It contains a pointer to the top node of a binary tree. The address we are looking for is at the leaves of its right sub-tree. Just next to the tree an equivalent example is shown with a linked list structure, but the pointers to the VM and vice-versa are not shown. The node in the binary tree has a link to the last element in an *out-out* chain which is preceded by an *in-in* chain with three elements. There is also an *out-in-in* chain starting at the first element of the *out-out*. This specific structure explains that there are currently three dependencies which are ready and have an *in* direction. The rest are waiting until the tasks with these dependencies finish. After they do, the first element of the *out-out* chain will be marked

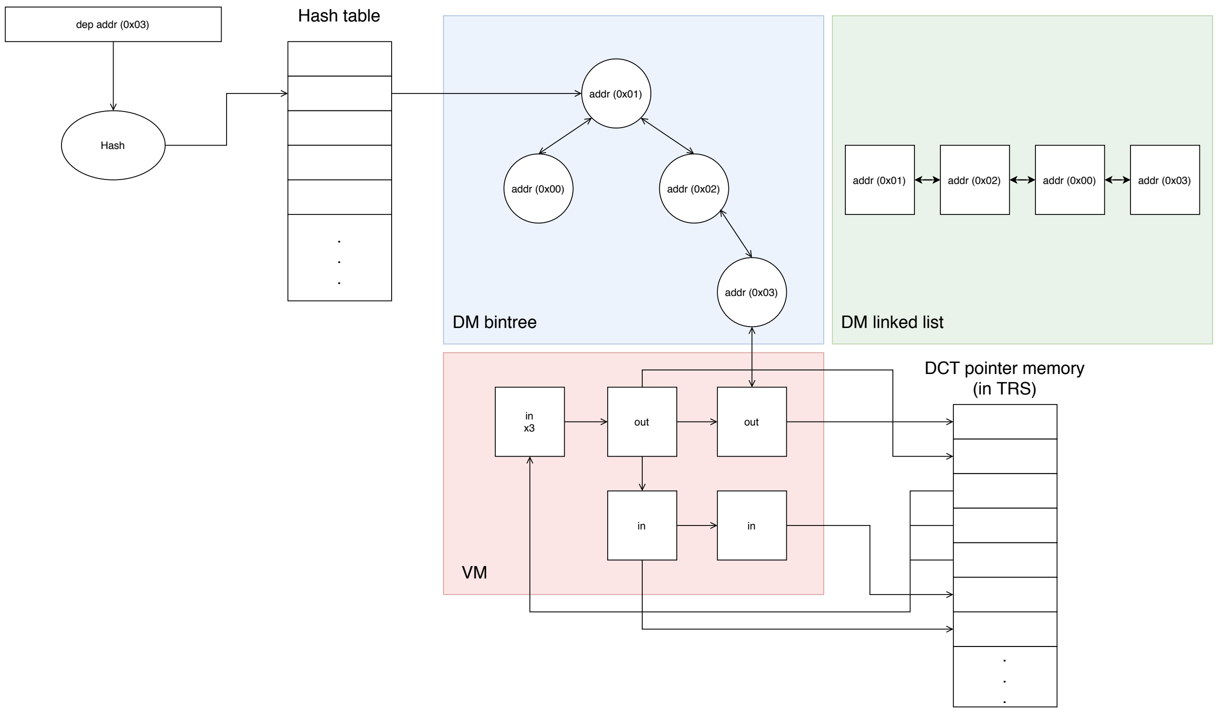


Figure 3.6: Schematic of the different data structures in the DCT and how they are linked

as ready. After it is finished, the two dependencies left in the *out-in-in* are ready and the chain becomes *in-in*. Then, the other *out* dependence is ready after the *in-in* finishes. After all chains are deleted, the entry in the DM is also erased.

3.2.4 Ready Task Dispatcher (RTD)

This module takes ready tasks and sends all the necessary data to an external 64-bit Axi-Stream interface. This data consists in:

- Header: It contains the number of arguments and copies of the task.
- Task ID: A 32-bit task ID generated by Picos. Is is made of the TRS ID and the number of dependencies. This is used later by the TRS so it doesn't have to read any memory to know the number of dependencies of a finished task.
- Parent task ID: The ID of the task which created the ready task, if any. This is taken directly from the parent task ID provided by the new task interface in the gateway.
- Task type: as with the parent task identifier, this is taken directly from the task type provided by the new task interface.

- List of arguments.
- List of copies.

A schematic with the internal components and data connections of the RTD is found in figure 3.7. The tasks sent via the Axi-Stream interface can be rejected by the slave attached to it. This can happen in the case that the processing element which should execute the task is busy. In that case, there is an additional interface, which is not handshake based, where the Axi-Stream slave has to send the task ID of the rejected task. The TRS ID of the task ID could be put back in the ready queue again to repeat the process, but this would imply an arbiter to control the write port of the queue. Since the TRS has to write to that queue too, it is simpler to make another queue internal of the RTD to put the rejected task TRS ID. When both queues are not empty, it chooses the ready queue to avoid sending always the same task when this one is rejected. This feature adds the possibility to send tasks to a different device (another FPGA or CPU). In this case, the external device may not have resources to store the task (e.g. the ready queue is full) so a rejection mechanism is needed. This feature extends the functionality of Picos++ and allows to build a decentralized system, where multiple instances of POM and/or software runtimes work together on different devices.

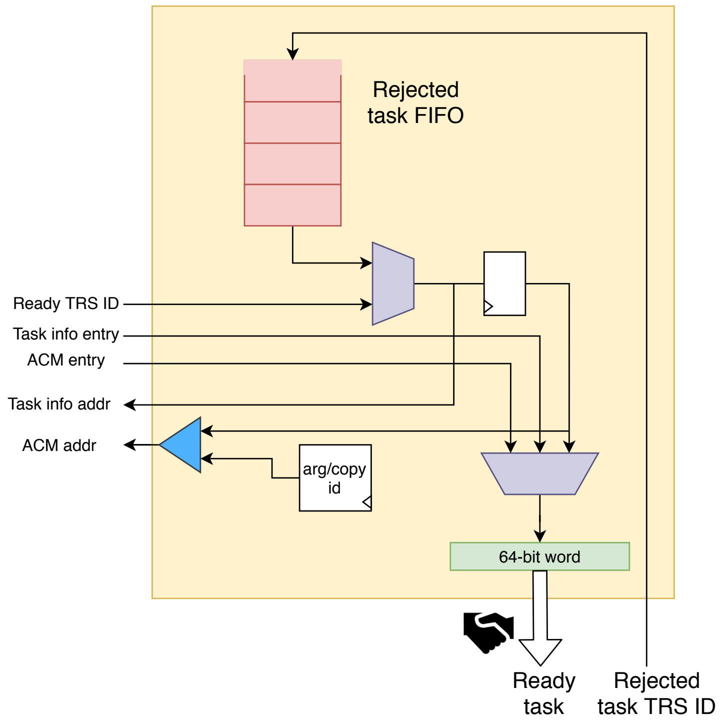


Figure 3.7: RTD internal design

3.3 Detailed example

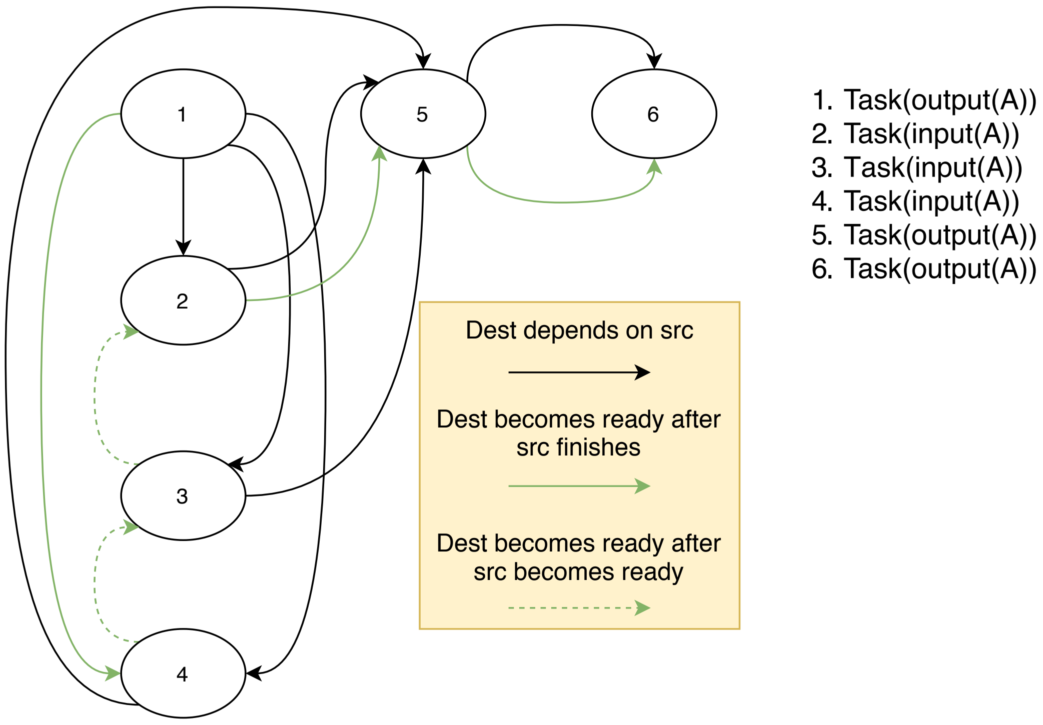


Figure 3.8: Dependence graph and task creation pseudocode of an example application.

In figure 3.8 there is a dependence graph of an example application. There are a total of six tasks, each one with only one dependence with the same address. For this example, we assume that Picos is initially empty, all tasks fit and they are stored before the first running task finishes execution. Since only a single address is referenced by all tasks, the hash table has only one slot used and the DM only accommodates one binary tree or linked list, this one having one node. However, the VM needs more slots to store the dependence chains. There is an *out-out* chain formed by tasks *1-5-6*, and an *out-in-in* chain formed by tasks *1-2-3-4*. After all tasks are stored inside Picos, only the first task is ready, which is notified immediately to the TRS by the DCT. The dependence counter in the TRS reaches zero as soon as it receives the notification, and proceeds to push the associated TRS ID into the ready queue. After this task finishes, the TRS notifies the DCT by sending a pointer to the VM where the dependence of task 1 is stored. Tasks 2 and 3 have an *in* dependence, and were created before tasks 4 and 5, so they depend only on task 1. Therefore, the DCT sends two notifications since both tasks can be executed in parallel. The DCT sends them in the inverse order in which they entered Picos. The *out-in-in* chain is transformed into an *in-in* with dependencies *2-3-4*. This chain is stored in the same entry as dependence 1, and slots for 2, 3 and 4 are deleted from the VM. After the first two tasks finish, the counter in the VM slot of the *in-in* chain is decremented. Then, when the last one finishes the slot is erased, destroying the chain, and dependence 5 is sent to the TRS. Finalization of this task triggers also a notification from the DCT to the TRS for task 6. When it is also finished, the last slot in the VM is deleted along with the related DM and hash table entries.

Chapter 4

Picos OmpSs Manager (POM)

Since the SOM hardware runtime already included an scheduler to assign tasks to the accelerators, we decided to integrate Picos Daviu within SOM. This makes maintainability of both runtimes easier because they share a big part of the internal structure and source code.

In this chapter we present the design solutions developed to obtain the integration. Some of the modifications have impacted also the original SOM design.

4.1 Cutoff manager

A restriction of Picos is that all tasks need to have at least one dependence. In SOM, the accelerator itself sends the task to the scheduler if it doesn't have any dependence, or to the CPU in the other case, by pushing it in a shared queue. The easy solution is to just replace this queue with Picos and let it handle all tasks with at least one dependence. The RTD of Picos is connected with the scheduler too. However, there are two main reasons to add a new module between the task creation interconnection, Picos and the scheduler. This module is called Cutoff manager.

4.1.1 Deadlock scenario

The limited capacity to store tasks inside Picos introduces a new type of deadlock not present in a software runtime. It can happen on applications with nesting tasks. The graph in figure 4.1 shows an example which could cause the deadlock. To simplify the example, we assume Picos has space to store only four tasks. When tasks 1, 2, 3 and 4 enter Picos, it triggers the full state in which it doesn't accept any more input. Then, since task 1 adds a level of nesting, when it tries to create task 1.1 a deadlock occurs. Picos cannot accept any task because its internal memories are full, but tasks 2, 3 and 4 are waiting for task 1 to finish because there is a dependence. Task 1 can't complete either because it is trying to create tasks 1.1, 1.2 and 1.3.

One solution is to execute tasks 1.1, 1.2 and 1.3 sequentially so they don't need to resolve any dependencies. The cutoff module implements a method to do this when Picos is full. Inside the header of a new task descriptor there is a sequence number since the last taskwait. The cutoff manager stores in a memory the number of finished tasks for each task with creation capabilities. When it detects that Picos is full and the incoming task is the first not executed task (if sequence number is equal to the number of finished tasks), it sends a special notification to the accelerator. This notification changes its behavior, making it remove the dependencies. This way, the next try will go directly to the scheduler. Once the task is created, the accelerator issues a taskwait before creating the next task to respect potential dependencies.

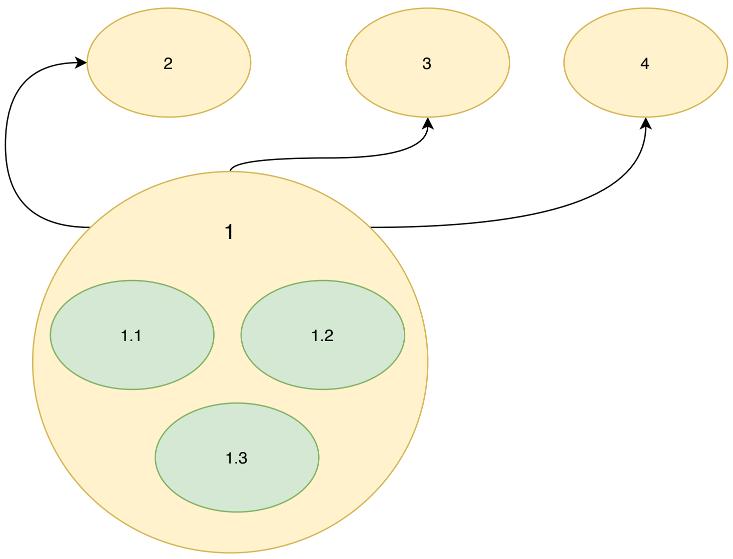


Figure 4.1: Task dependence graph causing the deadlock

4.1.2 Undefined number of task creators support

For the uses cases of SOM, each accelerator acts like a single thread in a CPU without context switch. They can only execute a specific code, and never start to execute a task without finishing the current. Therefore, at most there can only be as many task creators as accelerators supported by the runtime. There is a memory inside SOM to store data for each task creator (related to the taskwait), and it is guaranteed that will always have enough space. This is true also for POM when it is used with the OmpSs@FPGA programming model as SOM. Nevertheless, a use case of the original Picos and also POM is to act as the runtime of a multicore chip. In such environment, any number of threads can execute any code, therefore the number of contexts executing a task creation code is not known. This implies that the memory to store data for each task creator context could not have enough space. When this happens, the cutoff manager sends back a notification to reject the task. The module keeps rejecting all tasks incoming from new contexts until an existing context is waken up from a taskwait. Figure 4.2 shows the FSM governing the cutoff manager. The final mode triggers the behavior change in the task creator making it remove the dependencies of the task. The idle state is the starting point, and the final states (reject and accept task) always transition back to idle.

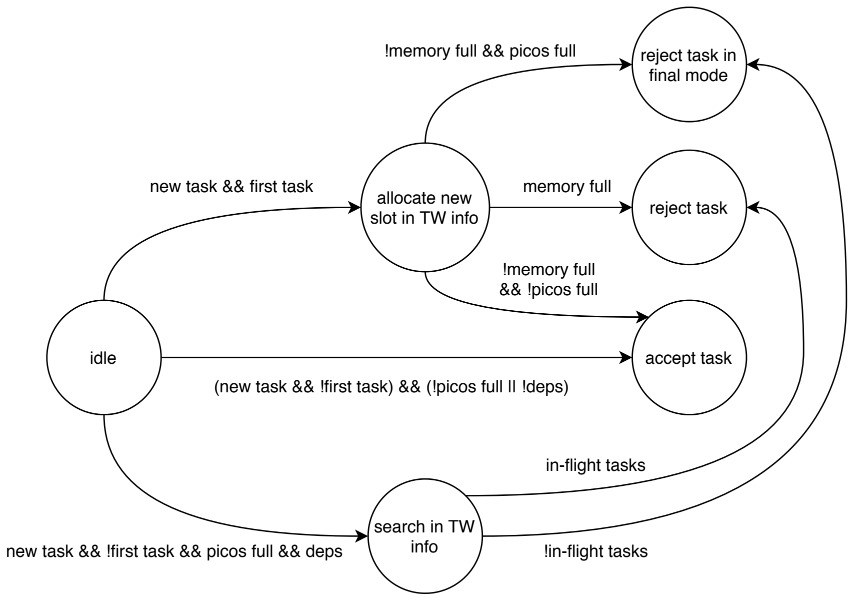

!memory full && picos full
reject task in final mode
new task && first task
allocate new slot in TW info
memory full
reject task
!memory full && !picos full
idle
accept task
(new task && !first task) && (!picos full || !deps)
in-flight tasks
search in TW info
new task && !first task && picos full && deps
!in-flight tasks

Figure 4.2: Cutoff manager finite state machine

4.2 Differences with Picos++

Although Picos Daviu and POM are an evolution of Picos++, they differ in both internal and external designs. Figure 4.3 compares these systems. The intermediary version which included hardware task creation and taskwait support in Picos++ is not contemplated in this section.

4.2.1 External interface

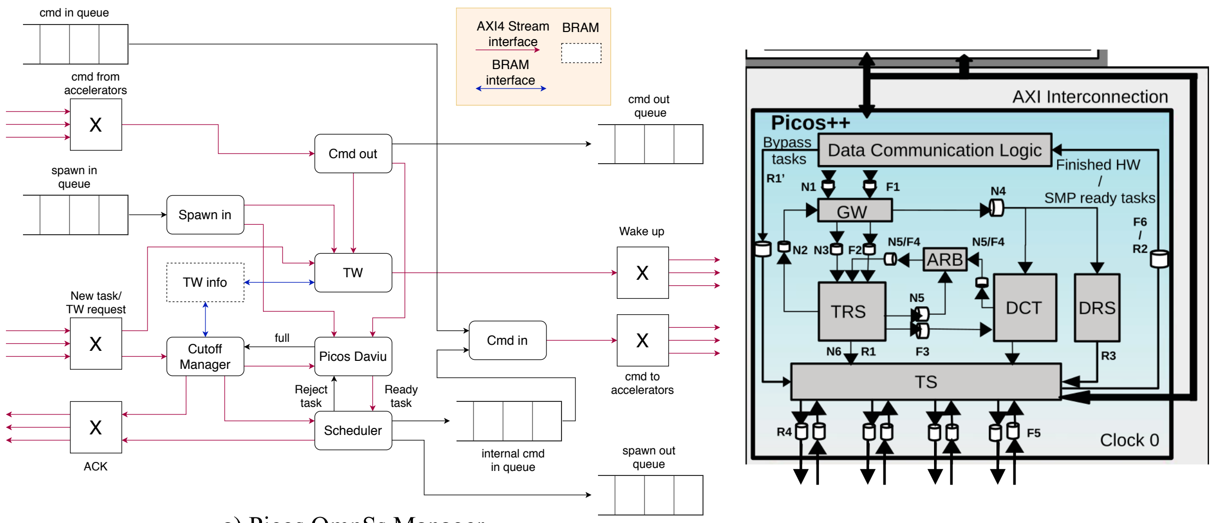
The first main difference is related to the external communication queues. They are not shown in the figure because Picos' buffers are stored in main memory (DRAM), whereas POM uses local SRAMs (BRAM in Xilinx FPGAs). There are also more queues since OmpSs@FPGA offers more features than Picos++.

- Command in queue: This is the main entry point for the hardware runtime. The CPU places commands which can be used to execute a task in a specific accelerator.
- Command out queue: The response queue for each command placed by the CPU in the command in. When a task initiated by the CPU finishes, the notification is placed here.
- Spawn out queue: This queue is used for tasks that have to be executed on a thread in the SMP.
- Spawn in queue: When a SMP task finishes, the notification to Picos is placed in this queue.

Accelerator communication is achieved in Picos++ with independent queues, stored in local RAM. When a task is ready to be executed, the scheduler module puts the arguments in the correspondent buffer. The same mechanism is used upon task finalization. POM does not include these queues and uses a shared interconnection instead. All accelerators are connected through a AXI-stream protocol to a crossbar, which shares the data bus. To avoid blocking when the accelerators are occupied, there is a multiqueue which works like the command in queue, but is internal to SOM. This buffer only allows one writer and reader at the same time. For the writer side it does not imply a performance degradation since all ready tasks are generated by Picos Daviu sequentially. For the reader side however, it does not allow to send arguments in parallel to the accelerators. Whereas in Picos++ it is indeed possible, this modification reduces significantly the area used by the design (that was one of the objectives of the new implementation).

4.2.2 Task creation

Both runtimes use different methods to reach the same goal. Moreover, they are not able to create tasks with the same source code. Picos++ relies on the new queue to receive tasks created by the CPU, and therefore it does not allow accelerators to create tasks. POM does not include this buffer, thus it is not capable to receive directly tasks created in the SMP. It relies instead on the task creation interconnection. Nevertheless, this interconnection was designed to be agnostic. Although it has not been tested at the time of writing, POM is prepared to receive tasks from multiple CPUs or other devices if there is a mechanism to communicate with them.



a) Picos OmpSs Manager (POM)

b) Picos++

Figure 4.3: Internal designs of POM and Picos++ hardware runtimes

4.2.3 Dependence management module

Picos Daviu is the dependence management module of POM. In Picos++, this includes the gateway, TRS, arbiter (ARB in the figure) and DCT. These modules communicate exclusively with queues, where commands are placed to take different actions. The main design difference, besides the dependence address matching algorithm, is the role of the TRS. In Picos++ it stores the equivalent to *in-in* chains in Picos Daviu. Thus, when a task finishes, the DCT sends only one message to notify that the next dependence is ready. The TRS starts to send one message to itself for each *in* dependence in the chain. Because we moved this labor to the DCT in Picos Daviu, the arbiter is no longer needed. Furthermore, since the TRS is much simpler, it does not need to communicate with queues which reduces area. An extra feature not present in Picos++ is the reject queue. When the internal command queue of an accelerator is full, the scheduler of POM rejects the ready task sent by the RTD, which stores it back on another queue. This is not needed in Picos++ because each accelerator has an independent buffer with enough length to store all tasks, thus increasing memory usage. Another difference is that Picos Daviu does not admit tasks without dependencies. This case is handled by the cutoff manager, which sends the task directly to the scheduler. Picos++ allocates a TRS ID instead (going through the gateway) and forwards the task to the scheduler immediately.

4.3 Performance study

In order to get more insights about how POM and Picos++ affect performance of real applications, we performed a study with a variety of benchmarks. Since the original version of Picos++ uses a different external interface, we decided to use the integrated version of the runtime with OmpSs@FPGA, which includes new features and improvements.

4.3.1 Methodology

To do this study, we designed a hardware system with 31 core simulators connected to the hardware runtime. Each task assigned to these cores contains only one argument, specifying the number of cycles the core has to wait before finishing that task. This method reduces significantly the resources of each core, and can be reused for any application. Moreover, there is one extra core that creates tasks for the other cores. This module reads a trace from main memory, which contains each task to be executed, its dependencies and the execution time. These traces are generated with a real execution of the benchmark in a SMP system with only one thread. The system is implemented on a FPGA inside the SoC of an Axiom board. It features a SoC with four ARM Cortex-A53 cores and an engineering sample version of the Xilinx Zynq Ultrascale+ FPGA (xczu9eg-ffvc900-1-e-es2), running at 100MHz.

We configured Picos Daviu to have the same internal memory capacity as Picos++, which has 256 TM entries and 512 DM and VM entries. However, the VM does not store the same information in both runtimes, Picos Daviu requires more memory to store the *out-in-in* chains. The factor of extra memory it needs depends heavily on the application, so we decided to not change it.

Benchmarks

There are two sets of benchmarks. The first group is synthetic and have specific properties we want to test. The other set is made of selected applications with different characteristics. These benchmarks are programmed in C using OmpSs. The synthetic benchmarks are:

- Free: Set of independent tasks with zero execution time.
- Chain: Each task depends on the previous (except the first one) and has zero execution time.

- Random graph: Each task can depend on previous tasks with a random factor, and has a random number of dependencies. The execution time of each task is configured with a parameter.

The free and chain benchmarks are used only for a preliminary study of the round trip time of a task. The random graph application aims to stress the dependence tracking system of the hardware runtime. The other set of benchmarks is:

- Matrix multiplication: A simple multiplication algorithm with square matrices. Each task applies the operation $C = C + A * B$, where A, B, and C are blocks of the input matrix.
- Nbody: A simulation algorithm to calculate the gravitational interaction between a set of stellar bodies over a period of time (steps).
- Cholesky: Matrix decomposition to the product of a lower-triangular matrix and its conjugate transpose.
- Spectra: Calculates the particle-particle distance of a set of molecules and updates a histogram. This is generated from a theoretical 3D model of a system structure, and compared against the X-ray spectrum of a real sample.
- Heat diffusion: Calculates heat propagation over a 2D surface.

All benchmarks range in a wide variety of dependence relationship, from embarrassingly parallel benchmarks to complex dependence graphs. There is also a big range of task sizes, from near zero execution time to tens of thousands of cycles. More details about these applications are specified in appendix A.

Some benchmarks were tested multiple times, changing the input parameters. Also some of them have modifiers to reduce or increase the task time. This can help to exaggerate the impact of the runtime overheads so that they are noticeable. The specific configuration of each application is on table 4.1. Chain and Free are not shown here because they were tested separately. Furthermore, the cholesky 4 configuration was not used in this study because we show the speedup over the sequential application, but the execution time of this case is zero. This configuration is used in a different study in section 5.3.

Table 4.1: Input parameters of each tested benchmark

	parameters			# tasks	modifiers
matmul 1	4096 size	256 block		4096	
matmul 2	8192 size	128 block		262144	
nbody	32768 bodies	512 block	16 iterations	66560	
cholesky 1	4096 size	128 block		5984	
cholesky 2	16384 size	128 block		357760	task time 0
cholesky 3	16384 size	128 block		357760	task time /10
cholesky 4	11520 size	64 block		988260	task time /5
rand graph 1	100 cycles/task			1024	
rand graph 2	100 cycles/task			900000	
spectra	200000 particles	2800 block		8742	distance task /500, aggregate task *2
heat	Gauss-Seidel	512 size 16 block	600 iterations	616452	

Software simulator

The main objective is to find and compare how close each runtime gets to its ideal speedup. Calculating this metric is not a trivial task since it depends on several factors, such as number of cores in the system, the scheduling policy, the communication latency, the internal memory size of the runtime, etc. We propose a semi-ideal case, where the runtime has infinite memories, the latency and any other overhead is zero. For the number of cores, we want to use as many as possible. However, since Picos++ is limited to 15 we could only use 14 of the 31 cores (and the task creator). When the number of cores is limited and the tasks are not symmetric, i.e. some tasks execute faster than others, the scheduling policy can affect the maximum achievable speedup. Since the scheduler of POM in not included in this work (it was taken from SOM), we are not interested in comparing both schedulers. We want to get the overhead caused by the dependence module itself. Therefore, to have a fair comparison, we developed two software simulators, each implementing a different scheduling policy in the semi-ideal case proposed.

The scheduler of POM uses a round-robin policy. It was designed to schedule in a simple way tasks that have multiple instances of the same accelerator. On the other hand, the Picos++ scheduler uses dynamic information to decide where to execute the task because it was designed to work on heterogeneous systems. It assigns a task to the device with less pending work.

Table 4.2: Simulator speedup with 14 cores against the sequential execution for the POM and Picos++ schedulers

benchmark	POM	Picos++
matmul 1	13.76487409	13.95233682
matmul 2	13.98150688	13.99794882
nbody	13.82848774	13.9915959
cholesky 1	13.61998353	13.82357057
cholesky 2	13.59098623	13.99749291
cholesky 3	13.96350172	13.99960651
rand graph 1	13.83783784	13.83783784
rand graph 2	13.99972001	13.99972001
spectra	8.183779991	9.042674403
heat	13.92839432	13.99030432

4.3.2 Task round-trip time

With the Free and Chain benchmarks, we can calculate an approximation of the minimum round-trip time of a task since it is created until it finishes. We tested two configurations of each benchmark, with one and eight dependencies per task for 500K tasks. The results are in figure 4.4. Almost all the time of these applications is consumed by the runtime overheads and is not affected by the scheduling policy. Therefore, we can observe from the Free benchmark, that Picos Daviu is actually faster than Picos++ processing dependencies. With only one it is slightly slower, but when increasing to eight dependencies per task, even if they are independent, Picos Daviu outperforms Picos++. This effect does not manifest in the Chain benchmark however. This is caused by the time it takes to POM to send a task since it becomes ready until it reaches the core. The time does not change from one to eight dependencies, which means that this time is much greater than the dependence processing time. POM uses a single memory to store a ready task queue per core. There is a module that is checking constantly and sequentially these queues until it finds a ready task to send. The time it takes since a task enters the queue until it is read by this module dominates the Chain benchmark. This happens because the next task is not ready until the current is finished, so it has to wait two times for the queue to be read.

Nevertheless, in the Free benchmark all tasks are independent, and therefore the ready queue is filled much faster. Thus, the module sending ready tasks does not have to start over the sequential scan to find the next ready task, and it can send bursts of ready tasks to the cores. Picos++ does not suffer from this disadvantage since it uses a different memory for each core.

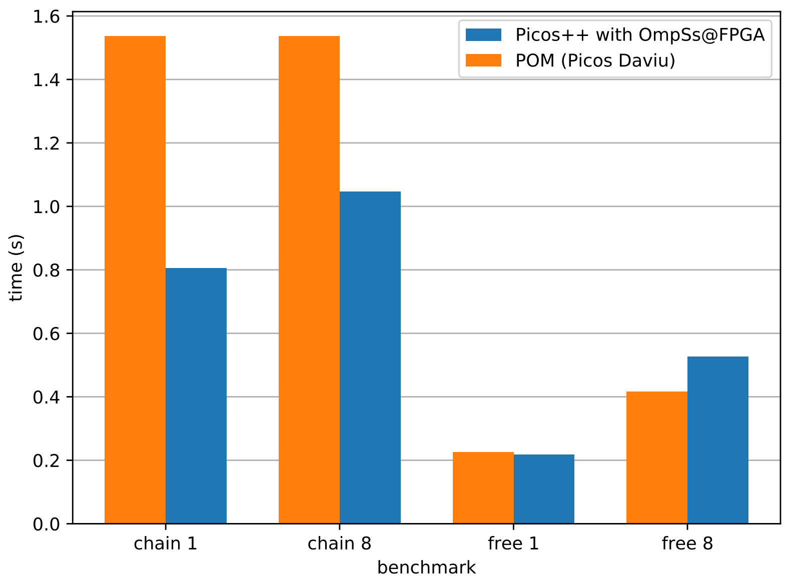


Figure 4.4: Execution time of the synthetic benchmarks Chain and Free, with one and eight dependencies per task.

4.3.3 Results

We run the other set of benchmarks on the same hardware system. Instead of comparing the execution time directly, we divide the time given by the software simulator by the results. This gives the percentage of the software simulator time over the real execution. It can also be interpreted as the percentage of speedup the hardware runtime achieves over the ideal. Figure 4.5 illustrates the results. Before analysing them, a few clarifications should be made. First, the cholesky 3 benchmark does not have a result for Picos++. It seems to be caused by an error in some part of its internal design. We couldn't get any numbers because the execution never finished. Another fact to highlight is that POM executes the nbody benchmark in less time than the software simulator. Although both systems use the same scheduling policy, the assignments of tasks to cores do not coincide. The software simulator assumes unlimited memory, so it can use the whole task graph. On the other hand, POM only analyses a part of this graph, which can alter the order in which tasks are scheduled. In addition, since the real system includes some overhead, two tasks that are ready could finish at a different order in the simulator than with POM. Therefore, since the tasks in nbody are highly asymmetric, a small change in the scheduling can have an impact in performance, even improving it.

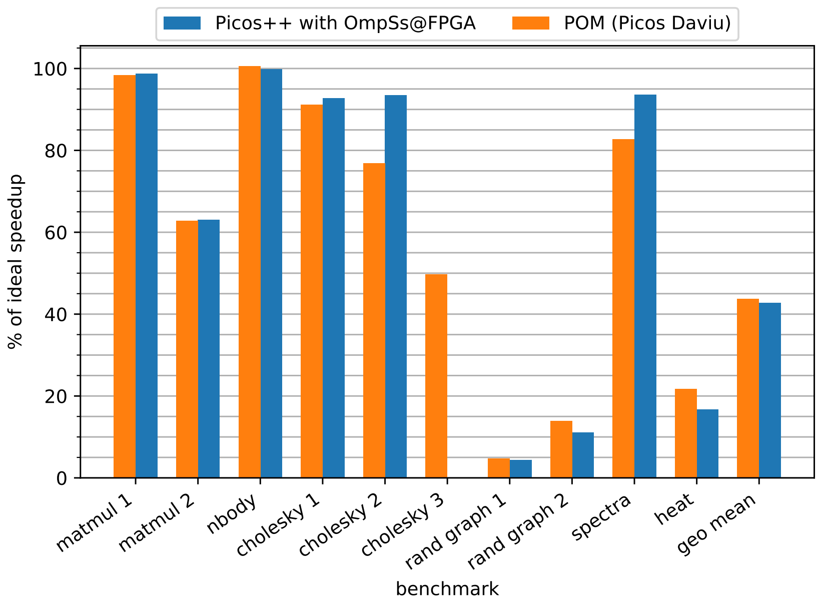


Figure 4.5: Percentage of ideal speedup achieved by POM and Picos++.

In some benchmarks, Picos++ achieves better results than POM (higher is better in the figure). This supports that it is faster delivering tasks to the cores since it has more parallelism in the ready queues.

Event if Picos++ latency is lower, it does not outperform POM in the nbody, random graph and heat benchmarks. They are the applications with most dependencies per task (except nbody). Random graph ranges from one to eight randomly, and heat uses seven. This confirms that in average, Picos Daviu is faster processing dependencies. On the one hand, it is possible that Picos++ gets full much faster because of its DM structure. On the other hand, the entire process of reading each dependence, and moving the necessary data between modules is slower because there are many more internal queues. Picos Daviu relies more on direct communication, minimizing the handshaking protocols which can delay module operations. As a consequence, the geometric mean is higher for POM. For Picos++, we used the same result in the cholesky 3 benchmark as POM to make a fair calculation.

The main reason why POM and Picos++ are very far from the ideal case in some benchmarks is because of the limited memory. For example, the random graph benchmark is highly parallel but the independent tasks are separated from each other. The hardware runtime only stores a portion of the whole dependence graph, limiting the amount of parallel tasks it can find. Furthermore the round-trip time affects negatively, since the simulator does not take into account this overhead. Thus, to improve performance one can either increase the memory capacity of the runtime or reduce the task round-trip time.

Table 4.3 shows the resource usage of Picos++ and POM and the maximum frequency each runtime can achieve in Xilinx FPGAs. Therefore, we could compensate the performance lost due to the latency to send tasks by increasing the area. Moreover, all tests have been performed with the same frequency (100MHz) but POM can achieve higher frequencies, thus compensating the performance gap of both runtimes.

Table 4.3: Post-synthesis resource utilization of POM and Picos++ for Xilinx FPGAs & maximum frequency

Hardware runtime	LUT	FF	BRAM	LUTRAM	Freq max (MHz)
POM	9492	10184	48.5	177	300
Picos++	17692	17365	142.5	518	200

Chapter 5

Use cases and results

This chapter shows the different platforms and uses cases where POM with Picos Daviu has been or it is going to be implemented and tested, with performance and area results for each case.

5.1 Classic FPGA-CPU systems

The current use cases of SOM and OmpSs@FPGA consists in a single FPGA connected to one or more CPUs. The objective is to use the FPGA to accelerate applications, moving most of the computation to specialized hardware modules in the FPGA. In these systems, POM can be used to reduce communication by also moving the task creation code and dependence management to the hardware. Also, it is able to improve performance over the same application with the SOM runtime, which doesn't support dependencies and relies in taskwaits instead.

5.1.1 Heterogeneous SoC: Xilinx Zynq ultrascale+

This platform features four ARM Cortex-A53 cores and an FPGA also called Programmable Logic (PL) embedded in the same chip, among other components. A block diagram showing all components is in figure 5.1. The FPGA has shared access to main memory with the ARM cores, thus being able to communicate fast. However, the downside of this type of platforms is the size of the FPGA, which can't grow in area as much as other systems like discrete devices.

We have tested three benchmarks in this platform: matmul, nbody and spectra [19]. However, we don't show matrix multiply results since there is no noticeable difference with SOM due to its embarrassingly parallel nature. The nbody application has an intrinsic high level of

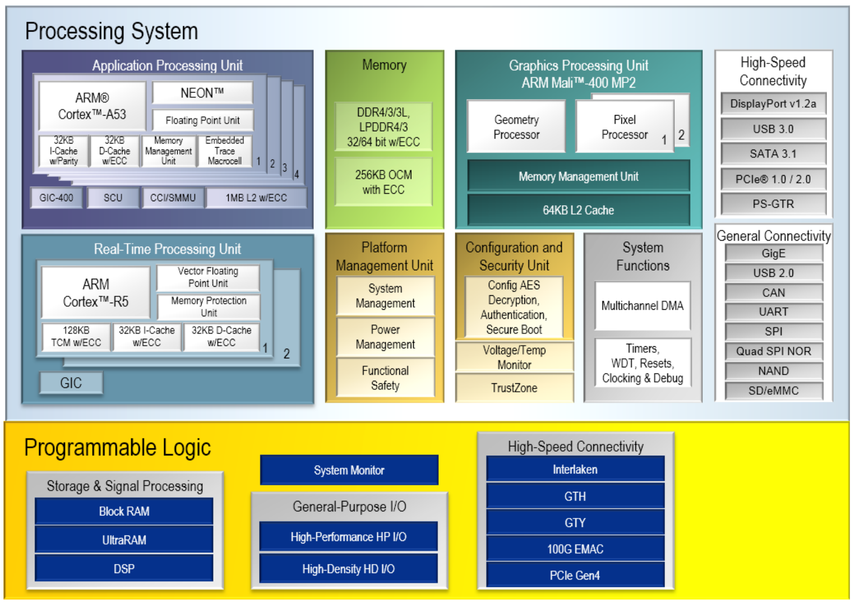


Figure 5.1: Block diagram of the Xilinx Zynq UltraScale+ EG platform

parallelism too. In this case nevertheless, POM can exploit parallelism between iterations. It is able to overlap the calculation of the next step before the current has finished, whereas SOM uses a taskwait to synchronize between iterations. This can be observed in table 5.1, where the speedup of the nbody benchmark is around 1%. The spectra application shows much better results, achieving almost double performance, since the SOM version doesn't use hardware task creation. Using this feature would require to change the original code and add additional taskwaits which would also degrade performance. In addition, though Picos++ is not included in the comparison, it could not be used in the same conditions for the spectra. The place and routing tools could not find a solution good enough to fit the same number of accelerators as with POM, which leads to less performance.

Table 5.1: SOM and POM performance comparison in the Zynq Ultrascale+ board

benchmark	SOM	POM	POM speedup
nbody	7.8 GPairs/s	7.9 GPairs/s	1.3%
spectra	2.71 s	1.41 s	92%

5.1.2 Discrete FPGAs: Alphadata ADM-PCIE7v3 and Xilinx Alveo U200

The Alphadata and Alveo devices incorporate an FPGA in a stand-alone platform, connected to an external server through PCI express. Figure 5.2 shows a schematic of the environment we use to test both platforms. Although communication with the FPGA is slower, they have more resources than SoC FPGAs, thus reaching higher performance. However, this is not the case when comparing the Virtex-7 with the Zynq since the first is an older model, using a technology with larger transistors. This also affects to the working frequency of the designs, the Alphadata has an approximate limit at 250MHz while the Zynq Ultrascale+ and Alveo can reach above 300MHz. These platforms also feature a dedicated main memory accessible only through the FPGA. All data needed by a task has to be transferred through PCIe to the memory before starting the execution.

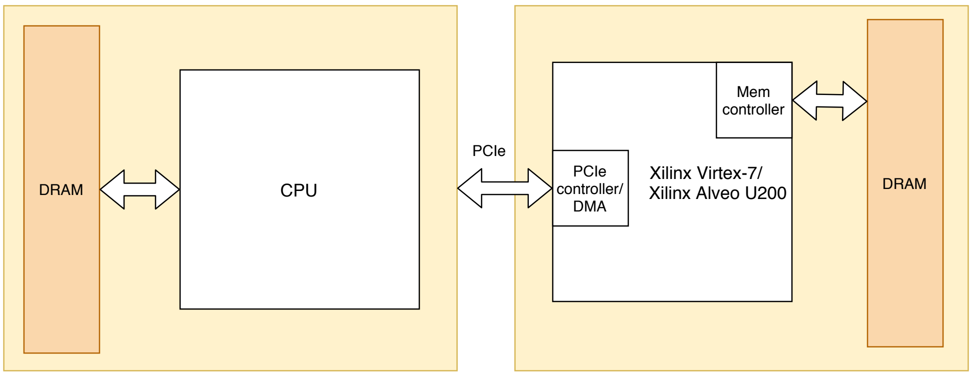


Figure 5.2: Schematic of an discrete FPGA system connected to an x86 server

We tested several benchmarks and applications on the Alphadata board, but since it cannot give any relevant result we don't show them in this work. On the other hand, the Xilinx Alveo board uses the same technology as the Zynq and it contains more resources. Therefore, it can be used to get even better results with spectra just by adding more computation accelerators. With three accelerators at 300MHz, the Zynq board can reach up to 19 billions of atoms pairs per second (bapps). With the Alveo FPGA, it is possible to place 10 accelerators at 300MHz, reaching a performance of 51.9 bapps (2.73 times faster). However, it is expected to get better results with this FPGA since it is an ongoing work at the time of writing, and better implementations are being studied. The performance results of this application on the ZU102 and the Alveo have led to a publication [19] where they are compared against an Intel implementation of the same computation in a Stratix 10 using Intel OpenCL runtime. Although the Stratix 10 is roughly 2 times bigger than the Alveo our implementation using Picos Daviu runs more than 2 times faster.

5.2 IBM cloudFPGA

The cloudFPGA project [20] proposes a different paradigm than the usual host-accelerator architecture. In both previous use cases, we use a CPU to send work to the FPGA, and to take back the results when the work is done. It's similar to the GPU case, where it is used to accelerate a CPU application. IBM proposes a cluster architecture, where several discrete FPGAs are connected to a CPU and between them through an internet network [21]. Each FPGA has the ability to communicate the same way a CPU does, thus giving it more control over the system and the ability to run stand-alone programs. They use Xilinx Kintex UltraScale FPGAs, a smaller and cheaper version of the Virtex series. A comparison of available resources with all mentioned FPGAs is in table 5.2.

Table 5.2: Available resources for the FPGA models tested with POM

Model	LUT	FF	BRAM	URAM	DSP	LUTRAM
Virtex 7 XC7VX690T	432368	864736	1470	0	3600	173992
Zynq UltraScale+ ZU9EG	274080	548160	912	0	2520	144000
Alveo U200	1182240	2364480	4320	960	6840	591840
Kintex UltraScale XCKU060	331680	663360	1080	0	2760	146880

In figure 5.3 there are two FPGAs and one CPU connected to the cloud through ethernet. The FPGAs include a IP developed by IBM to handle UDP and TCP protocols (and all layers beneath) called the Shell. It helps the end user to program the device by providing a single AXI-stream interface for input and output data packets.

Although the FPGAs have more potential, we still use them as accelerators handled by the CPU. Similar to the discrete case, the server initially sends the data needed by the program to all FPGAs in the cluster and then it controls the accelerators. Nevertheless, there is a new problem introduced in this environment, which is the latency of the network. Compared to the SoC or even the PCIe cases, the ethernet network has a higher latency. This affects very negatively the performance of applications, and it gets worse as the number of tasks increases. Therefore, it is critical to limit the communication between CPU and the FPGAs. POM is ideal for this situation, since it allows the FPGA to create tasks and handle all their dependencies without having to notify the CPU. We can reduce most applications to one task which creates all tasks of the program. This task can be implemented in the FPGA so the CPU only has to send the data and one message to initiate the accelerator. After the application finishes, the

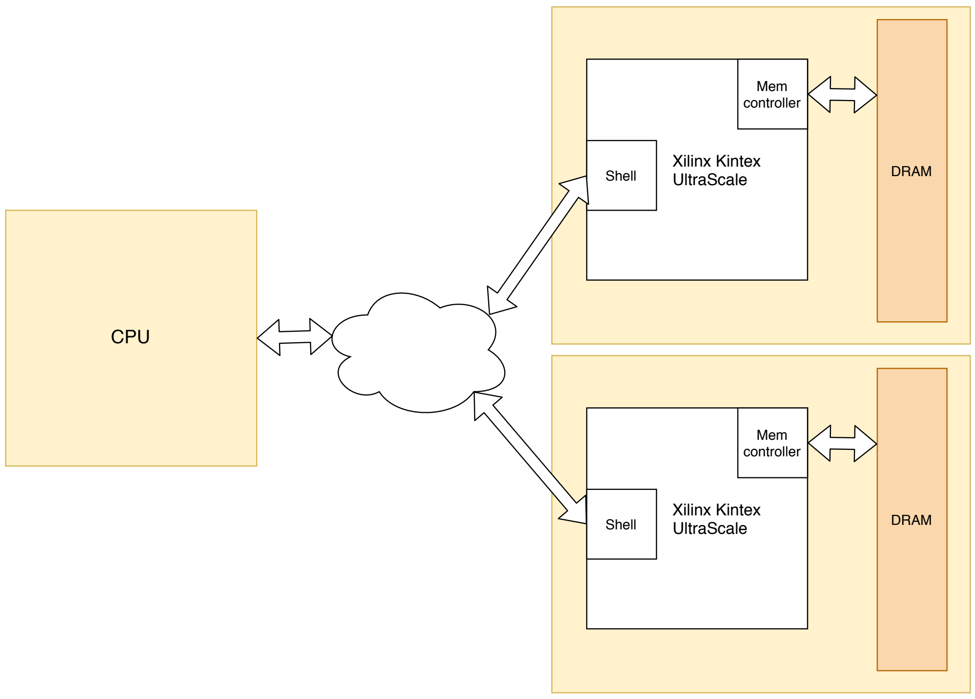


Figure 5.3: cloudFPGA cluster

FPGA sends a message to the CPU and it only has to take the data back. In the case where there are many FPGAs, it only has to send a single message per device.

Nbody results

We performed three different tests with nbody. The first one runs the application over a SMP only, with an 8-core x86 Skylake processor. Then, we moved the computation to only one FPGA. The CPU generates and sends the initial positions of the bodies, triggers the accelerators in the device, waits for completion and gets the final positions back. Then, we tried doing the classic approach of the OmpSs@FPGA flow, by creating the tasks in the CPU and sending them to the FPGA for execution. In this version, Picos is not used since all dependencies are handled by the SMP. Figure 5.4 compares the result of the tests with different sizes. The FPGA versions do not include the time to transfer data between devices. This time affects the final result more than it should because the communication protocol was not optimized. As expected, handling dependencies in the CPU is not a viable option, as the number of tasks get higher, more messages have to be exchanged between the SMP and the FPGA. We didn't test it with 32768 bodies since it would take too much time to complete. It takes about 15 times more than using the hardware runtime. On the other hand, the SMP-only version is about 3 times slower.

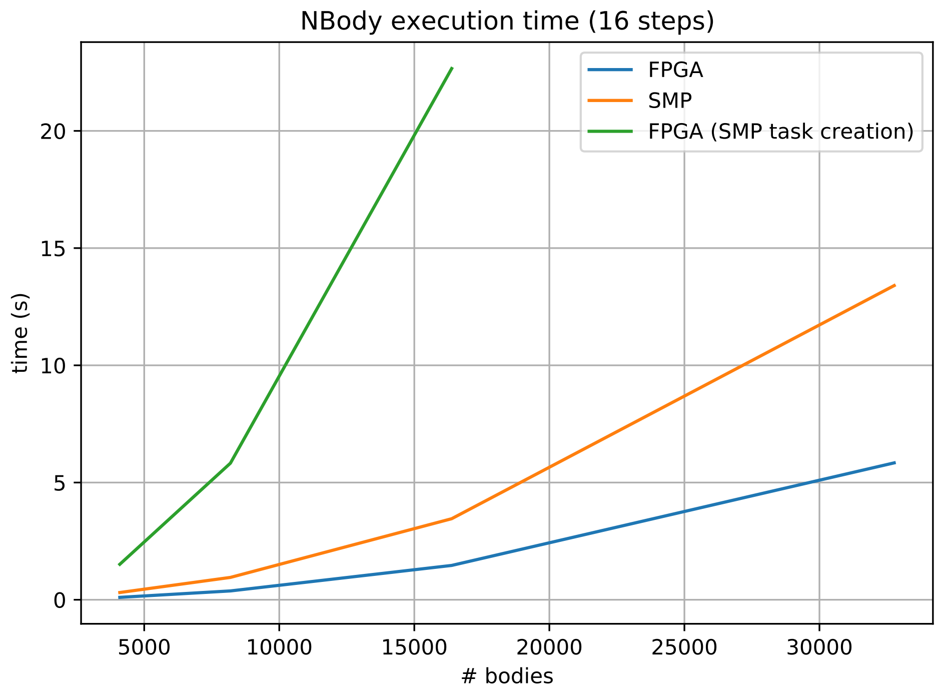


Figure 5.4: NBody test results on the cloudFPGA platform

We did another separate test with the nbody benchmark and two FPGAs. The application runs with an MPI like distribution. Each FPGA calculates half of the forces, but updates all particles' positions. Therefore, each device needs the forces computed by the other to do that part. Furthermore, dependencies are handled by two separate instances of POM in each device. Sending and receiving forces acts as a synchronization point, which does not allow to overlap calculations of different simulation steps. Figure 5.5 shows the results. Using two FPGAs can improve performance with enough number bodies. Since the communication method used was not optimal, it takes a significant amount of time to send data between FPGAs, limiting the speedup. Optimizing the protocol was not possible at the time of the test, due to some limitations in the hardware infrastructure handling the UDP protocol. This problem will be fixed in future versions of the Shell.

Matrix multiply

We also prepared some tests with the matrix multiplication benchmark. As in the previous study, we compared a pure SMP execution, an FPGA implementation creating tasks inside and outside the device and with two FPGAs creating inside. Figure 5.6 shows the performance

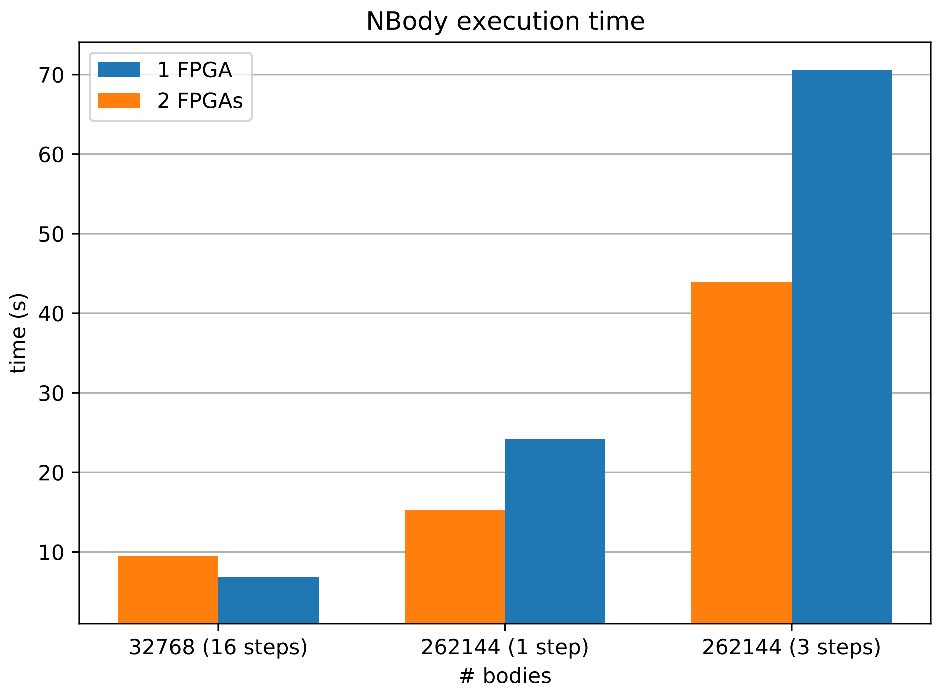


Figure 5.5: NBody test results with two FPGAs

results in GFLOPS for the different matrix sizes and every different implementation. As it can be seen the figure shows a similar effect to the nbody benchmark. The main difference is that for the SMP implementation we used an OmpSs approach in which each block is calculated with OpenBLAS, an optimized open source library mostly targeting scientific applications [22]. The figure shows again that the best option is moving the computation and task creation to the FPGA. In addition, using two FPGAs almost doubles performance because the benchmark does not need to communicate data between FPGAs. Each device counts with the three matrices at the beggining, and use their own copy to compute half of the multiplication, which is recovered by the CPU upon finalization. However, the time to send and receive the matrices is also not counted here due to the high overhead it adds.

5.3 European Processor Initiative (EPI): eFPGA

The EPI project [23] aims to design an European General Purpose Processor (GPP) targeting High Performance Computing (HPC) and automotive applications. The first version will include multiple symmetric ARM Zeus cores, an embedded FPGA (eFPGA) and other accelerators like a RISC-V vector unit. As figure 5.7 illustrates, all hardware blocks are connected to a Network

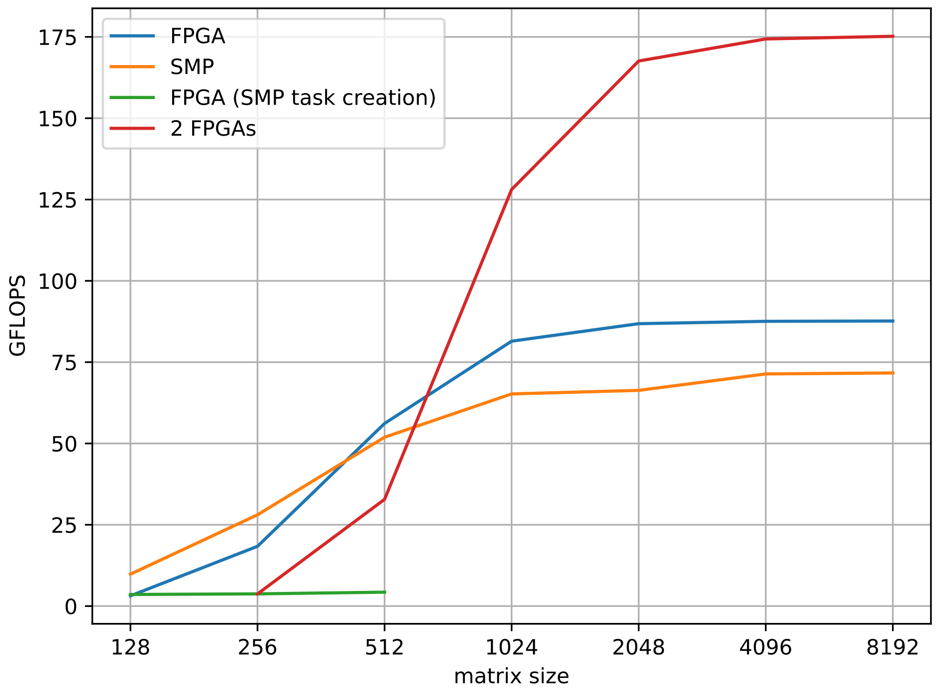


Figure 5.6: Matmul benchmark results with different configurations on the cloudFPGA platform

on Chip (NoC) which leads to main memory. This is shared among all blocks, and core caches are coherent throughout the system.

We developed a special version of POM to be implemented on the eFPGA. The first challenge we encountered is that the FPGA is developed by Menta [24]. Therefore, our methodology to develop the POM IP cannot be used in this use case since it is specific for Xilinx tools. Though the internal structure of the IP is still the same, the implementation had to change for most modules. For instance, we can no longer use Xilinx IPs like AXI-stream crossbars and BRAM instantiation templates. We developed from scratch a simpler version of those modules, implementing the minimum requirements for our project.

Another reason to change the implementation is the aim of POM in this system. First, it will control the hardware accelerators. This is more similar to the current use cases of Picos, and facilitates the programmer work to use these specialized modules. Using POM should speedup programs since the expected task size of these accelerators is very low compared to the cores. We expect that a software runtime would take more time than the actual execution time of these accelerators. The main change of this use case is that POM will also manage tasks for the cores. The objective is to act as the runtime for any task created by any core. The expected

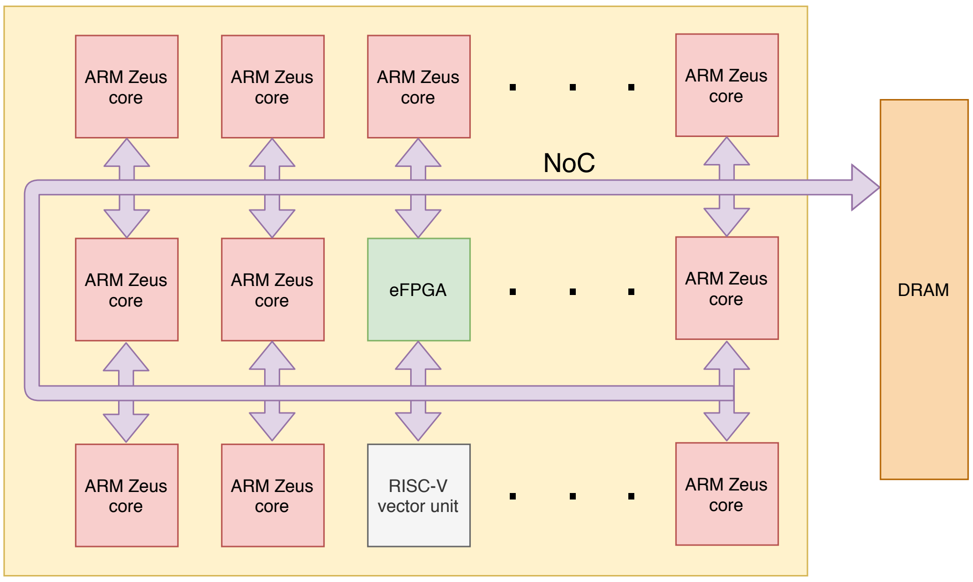


Figure 5.7: EPI GPP overview

number of cores inside the same chip is high, between 32 and 64. A software runtime would struggle to feed work to so many CPUs, and also the thread contention overhead would be more noticeable, since more threads have to access critical parts of the runtime like the ready queue or the dependence graph.

The main challenge here is making POM fit with limited resources and area in the eFPGA. The architecture can be customized, the number of logic blocks, Look UP Tables (LUTs), Flip-FLops (FF) and memories (similar to Xilinx BRAMs) can be configured and optimized for our application. However, the total area dedicated to the eFPGA tile is limited enough to not fit with the default design and configuration of POM. Therefore, we had to perform several studies to reduce area. There are two main paths to achieve this objective. The first one is to study and measure the impact of Picos parameters' in area and performance, so they can be fine-tuned and reduce area without sacrificing too much performance. The other one consists in redesigning and optimizing the other parts of POM. Some of the modules are generated by the Xilinx tool Vivado HLS. It transforms C/C++ code into RTL code, making it easier and faster to develop specialized hardware modules. Nevertheless, the resulting HDL code is functional but not optimal, neither in area nor performance.

5.3.1 Picos parameter exploration

To study the impact of the different values for each parameter, we used the same set of benchmarks as in the performance study of section 4.3.

Methodology

The methodology is similar to the study of 4.3. We execute traces of real application on a simulator environment with 31 cores managed by POM. The main difference is that in this case we can use all the cores available in the simulator and Picos++ is not included in this study.

Configurations

We tested six different configurations, changing only a subset of the parameters. The objective is to study each parameter's effect and compare it to a baseline. These configuration are in table 5.3, being configuration 1 the baseline. Configuration 2 increases the TM and VM size, decreasing the DM size. We expect that this should increase performance of some benchmarks, although the DM has more limited size, it only uses one entry per dependence, without repetitions. However, the VM can use at most one entry per dependency of each tasks, with repeated addresses. Therefore, the VM should have a bigger size. Configuration 3 replaces the binary tree implementation of the DCT by a linked list. The expected result is that it should decrease performance for some benchmarks. Configuration 4 does not use the hash table, and therefore all dependencies are stored in the same binary tree. The results are expected to be worse than the baseline. Configuration 5 uses a simpler hashing function. Instead of the Pearson method, it applies the xor operation directly to the dependence address, so it does not need any extra ROM. Finally, configuration 6 uses the linked list, xor hashing function and it increases the hash table size. This is fixed for Pearson hash, but the xor does not have this limitation. The aim of this test is to mitigate the linked list performance degradation effect by increasing the hash table size. The addresses should be more distributed and each linked list should have less elements in average.

Results

We compare the relative performance of configurations 2, 3, 4 and 5 with the baseline in figure 5.8. Each benchmark reacts very differently to each configuration, depending on their characteristics.

Table 5.3: Tested Picos parameter configurations

	Config 1	Config 2	Config3	Config 4	Config 5	Config 6
Max args/copies	15	15	15	15	15	15
Max deps	8	8	8	8	8	8
TM size	128	512	128	128	128	128
DM size	512	384	512	512	512	512
VM size	512	768	512	512	512	512
DM data structure	BTREE	BTREE	LLIST	BTREE	BTREE	LLIST
Hash function	PEARSON	PEARSON	PEARSON	NONE	XOR	XOR
Hash table size	64	64	64	1	64	256

The matmul benchmark is only affected by configuration 2, which changes the Picos internal memories. This is caused by the task creation order. Although it is an embarrasingly parallel application, if the tasks are created in such an order so each task depends on the previous, only one task per each sequence of n tasks will be free of dependencies. This n depends on the block and matrix size. In configuration 1, Picos can only hold 128 tasks, where only two can be executed in parallel in matmul 2 (it stores two chains of 64 dependent tasks). Since configuration 2 increases TM size, more parallelism can be found by the runtime, thus increasing performance. An alternative way to achieve the same result without modifying the memory size is changing the software code. By creating the task in an alternative order, where each task is independent from the next one, Picos is able to find parallelism faster and does not depend on the TM size.

In the top-right plot we can observe the effects of the DM data structure. Using a linked list instead of a binary tree does not affect most benchmarks because of the task size, dependencies per task and task dependence graph complexity. If the execution time is too big, the search algorithm time is hidden. If the number of dependencies per task is too low, the time to process each task is short enough to be hidden by the rest of the runtime overhead. Furthermore, the lower the task time, the less time a dependence exists in the DM, so the search algorithm takes less time because the number of comparisons is low. However, the random graph and heat benchmarks are the only ones with more than 4 dependencies per task, and with not very small nor very big tasks. Therefore, the search algorithm makes a difference in performance for these cases. Nevertheless, the impact does not even reach a 10% of improvement for the binary tree in the worst case.

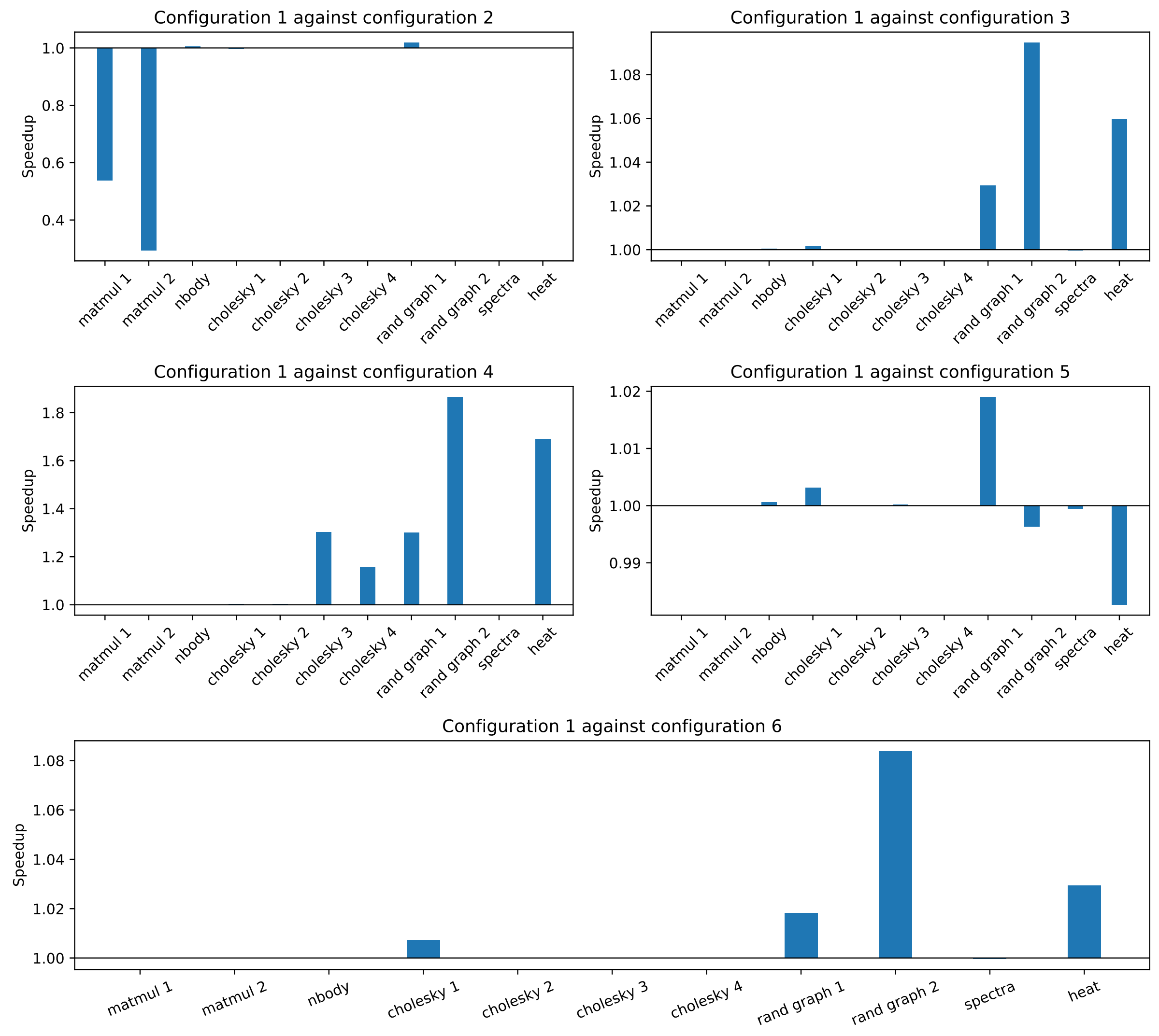


Figure 5.8: Relative performance for each Picos parameter configuration with the baseline

The middle-left plot explains what happens when there is no hash table. All benchmarks with low execution task time are affected, with the exception of cholesky 2. This case uses tasks with zero execution time, which means that dependencies do not live long enough in the DM to be affected by the search algorithm. The effect of this configuration depends also on the dependence graph complexity, because it determines how many tasks can be executed in parallel at any time.

The middle-righ plot shows the impact of using a simpler hash function. The results are not very conclusive, they do not show a significant and consistent improvement or degradation in performance. Some applications run faster with the xor function, like heat, whereas other work better with the pearson function like the random graph 1. Therefore, the hash function implementation is not a critical part of the DCT. Since the xor function takes less resources than Pearson, it is a better option to use it.

To finalize this analysis, the last plot in the bottom shows a combination of parameter changes. It uses a linked list, so it is expected to lose performance. However, the use of a bigger hash table has some positive impact. The same benchmarks as the top-right plot are affected, but with smaller performance degradation.

Table 5.4: Resource usage of the tested configurations with Xilinx FPGAs

	LUT	FF	BRAM	LUTRAM
conf 1	1516	1384	22.5	89
conf 2	1767	1455	67.5	97
conf 3	1214	1301	22.5	80
conf 4	1349	1207	22.5	81
conf 5	1497	1338	22.5	89
conf 6	1236	1257	22.5	128

In this preliminary tests we didn't take into account the area, which is one of the main reasons of the whole study. We computed the area gain of each configuration over a Xilinx FPGA (Xilinx Ultrascale+ xczu9eg-ffvc900-1-e-es2). Table 5.4 includes the number of resources used per each primitive of the FPGA.

To make a brief summary of the study, we compared the average performance over all benchmarks of each configuration with the resource usage. To do that, we compute a metric to decide which configuration is better taking into account both criteria. The final objective is to find a configuration that reduces area without sacrificing too much performance. First, we compute the relative area gain dividing each resource usage by the baseline (configuration 1). Then, we take the mean of this result, the higher it is, the less area the configuration uses. Finally, we calculate the average of the results of figure 5.8 for each configuration, and divide it by the computed area gain. The lower it is the result the better. It can decrease either by going faster than the baseline or using less resources. They are illustrated in figure 5.9. We can observe that although configuration 2 achieved the best performance, the area needed to increase the memory size of Picos is too big. Configuration 4 has the same effect but with the opposite reasons. It uses less area than the baseline, but the performance penalty is too high. Configurations 3 and 5 seem to be the most balanced. Both have a performance degradation, but since they save area the final result shows that they are better than other configurations. To finish this study, we can see that configuration 6 is not better than 3 or 5. The performance achieved by this configuration is better than configuration 3, but not enough to justify the increase in resources. This is caused by the hash table because it increases significantly the LUTRAM usage.

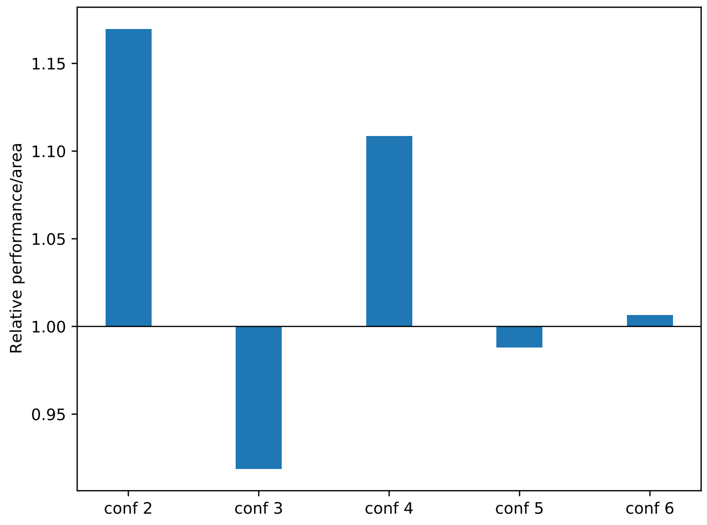


Figure 5.9: Average relative performance of each configuration against configuration 1 divided by the area gain over the same configuration (lower is better)

Chapter 6

Conclusions and future work

Some improvements and features of Picos Daviu, POM and its uses cases will be addressed in our next steps, as a continuation of this work:

- Improve the round-trip time of a task in POM. The best way to achieve this performance is implementing all modules in RTL, avoiding HLS implementations.
- Study different scheduling policies for the POM scheduler. For the moment we know that at least the Picos++ scheduler is better in most cases.
- Fully adapt POM to the EPI use case. We already have area estimation, but still part of integration with the ARM cores and the NoC is needed in order to have a functional design. For example, deciding a communication method and protocol to exchange tasks with the CPUs.
- Analyse our cloudFPGA system with more than two FPGAs when the current infrastructure, with some external limitations, allows us to do so. The idea is to have a distributed system, where each FPGA works on its own part of the problem, and is able to exchange both data and even tasks with other FPGAs. For instance, one accelerator being able to create tasks for another device. POM would be able to handle its dependencies and send it when it's ready.
- As Nexus# and first Picos design, add more than one instance of the TRS and DCT. This would accelerate the processing of new and finalized tasks since dependencies belonging to different DCTs could be handled in parallel.

In this work we have presented Picos Daviu, a hardware task dependence manager to establish dependence relationships in task-based programming models, based on the design of Picos++. We also integrate it with Smart OmpSs Manager (SOM), the hardware runtime of OmpSs@FPGA, creating Picos OmpSs Manager (POM), an improved version of Picos++.

POM specializes the critical part of a task-based programming model runtime in hardware overcoming the challenge of managing and exploiting fine-grain parallelism. Software runtimes are not able to give work at the required speed in this context of fine-grain parallelism, although larger and larger number of cores or accelerators can be used in heterogeneous systems.

We compare POM with Picos++, which presented internal design problems, making it difficult to integrate in our use cases. Picos++ uses more resources in order to achieve lower latency in the process of sending a ready task to an accelerator. Also, we encounter a situation where Picos++ does not fit on an FPGA board. Our proposal, Picos Daviu, is faster processing dependencies and is able to fit in the same FPGA board.

We have evaluated POM in several uses cases, from FPGA based SoC systems to cloud systems. Results show that the idea of handling dependencies in the FPGA can double the performance in the spectra benchmark compared to an OpenCL implementation. Moreover, POM can help to decouple the dependence management between different discrete FPGAs in a cloud system. As an example, OmpSs@FPGA versions of nbody and matrix multiplication benchmarks using POM runtime system in two FPGAs achieve a 1.6x and 2x speedup, respectively, compared to a single device. Finally, we have done a design space exploration to integrate a tuned POM design as a hardware runtime of a multicore system under EPI project. The new design has to deal with hard area limitations. First conclusion is that the linked list DM structure and xor hash function are the best candidates to tune POM and achieve a good performance vs area trade-off.

Bibliography

[1] (2018) Openmp application program interface - openmp standard 5. [Online]. Available: https://www.openmp.org/press-release/openmp-5-0-preview-1-published

[2] B. S. Center. Ompss programming model. [Online]. Available: https://pm.bsc.es/ompss

[3] E. Ayguadé, R. M. Badia, D. Cabrera, A. Duran, M. González, F. D. Igual, D. Jiménez-González, J. Labarta, X. Martorell, R. Mayo, J. M. Pérez, and E. S. Quintana-Ortí, "A proposal to extend the openmp tasking model for heterogeneous architectures," in *Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP, IWOMP 2009, Dresden, Germany, June 3-5, 2009, Proceedings*, ser. Lecture Notes in Computer Science, M. S. Müller, B. R. de Supinski, and B. M. Chapman, Eds., vol. 5568. Springer, 2009, pp. 154–167. [Online]. Available: https://doi.org/10.1007/978-3-642-02303-3_13

[4] J. Bosch, M. Vidal, A. Filgueras, C. Álvarez, D. Jiménez-González, X. Martorell, and E. Ayguadé, "Breaking master-slave model between host and fpgas," in *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, R. Gupta and X. Shen, Eds. ACM, 2020, pp. 419–420. [Online]. Available: https://doi.org/10.1145/3332466.3374545

[5] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Application acceleration on fpgas with ompss@fpga," in *International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018.* IEEE, 2018, pp. 70–77. [Online]. Available: https://doi.org/10.1109/FPT.2018.00021

[6] B. S. Center. Ompss@fpga. [Online]. Available: https://pm.bsc.es/ompss-at-fpga

[7] L. Sommer, J. Korinth, and A. Koch, "Openmp device offloading to FPGA accelerators," in *28th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2017, Seattle, WA, USA, July 10-12, 2017.* IEEE Computer Society, 2017, pp. 201–205. [Online]. Available: https://doi.org/10.1109/ASAP.2017.7995280

[8] Vitis platform. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[9] Vineyard project overview. [Online]. Available: http://www.vineyard-h2020.eu/en/project/objectives-and-rationale-of-the-project

[10] S. Kumar, C. J. Hughes, and A. D. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, D. M. Tullsen and B. Calder, Eds. ACM, 2007, pp. 162–173. [Online]. Available: https://doi.org/10.1145/1250662.1250683

[11] C. Meenderinck and B. H. H. Juurlink, "A case for hardware task management support for the starss programming model," in *13th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2010, 1-3 September 2010, Lille, France*, S. López, Ed. IEEE Computer Society, 2010, pp. 347–354. [Online]. Available: https://doi.org/10.1109/DSD.2010.63

[12] Y. Etsion, F. Cabarcas, A. Rico, A. Ramírez, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA.* IEEE Computer Society, 2010, pp. 89–100. [Online]. Available: https://doi.org/10.1109/MICRO.2010.13

[13] T. Dallou, N. Engelhardt, A. Elhossini, and B. Juurlink, "Nexus#: A distributed hardware task manager for task-based programming models," in *2015 IEEE International Parallel and Distributed Processing Symposium.* IEEE, 2015, pp. 1129–1138.

[14] T. Dallou, B. Juurlink, and C. Meenderinck, "Improving the scalability and capabilities of the nexus hardware task management system," in *Proc. 1st Int. Workshop on Future Architectural Support for Parallel Programming*, 2011.

[15] F. Yazdanpanah, C. Álvarez, D. Jiménez-González, R. M. Badia, and M. Valero, “Picos: A hardware runtime architecture support for ompss,” *Future Generation Computer Systems*, vol. 53, pp. 130–139, 2015.

[16] X. Tan, J. Bosch, M. Vidal, C. Alvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, “General purpose task-dependence management hardware for task-based dataflow programming models,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 244–253.

[17] L. Morais, V. Silva, A. Goldman, C. Álvarez, J. Bosch, M. Frank, and G. Araujo, “Adding tightly-integrated task scheduling acceleration to a RISC-V multi-core processor,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 861–872. [Online]. Available: https://doi.org/10.1145/3352460.3358271

[18] X. Tan, J. Bosch, C. Álvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, “A hardware runtime for task-based programming models,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 1932–1946, 2019.

[19] C. González, J. Bosch, J. M. de Haro, A. Filgueras, M. Paolini, S. Balocco, C. Álvarez, and R. Pons, “Accelerating pp-distance algorithms on fpga using different strategies and runtime managers (under revision),” *Future Generation Computer Systems*, 2020.

[20] cloudfpga project. [Online]. Available: https://www.zurich.ibm.com/cci/cloudFPGA

[21] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, “Network-attached fpgas for data center applications,” in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 36–43.

[22] Openblas: An optimized blas library. [Online]. Available: https://www.openblas.net/

[23] Epi: European processor initiative. [Online]. Available: https://www.european-processor-initiative.eu

[24] Menta, embedded programmable logic. [Online]. Available: https://www.menta-efpga.com

Appendix A

Benchmarks

A.1 Task dependence graphs

The following figures illustrate the resulting task dependence graphs for each benchmark. They are generated with short executions, each node contains the order in which it was created and the name of the executed task (if any).

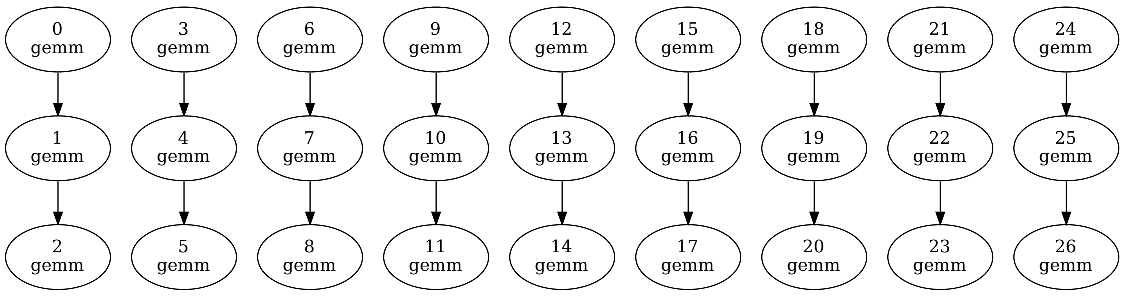


Figure A.1: Matrix multiply

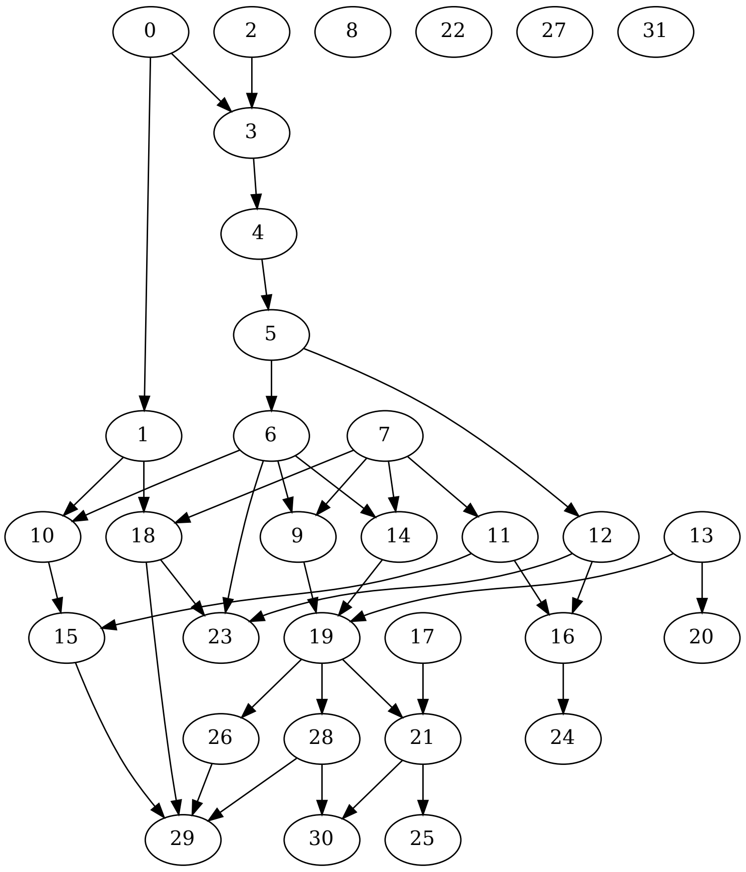


Figure A.2: Random graph

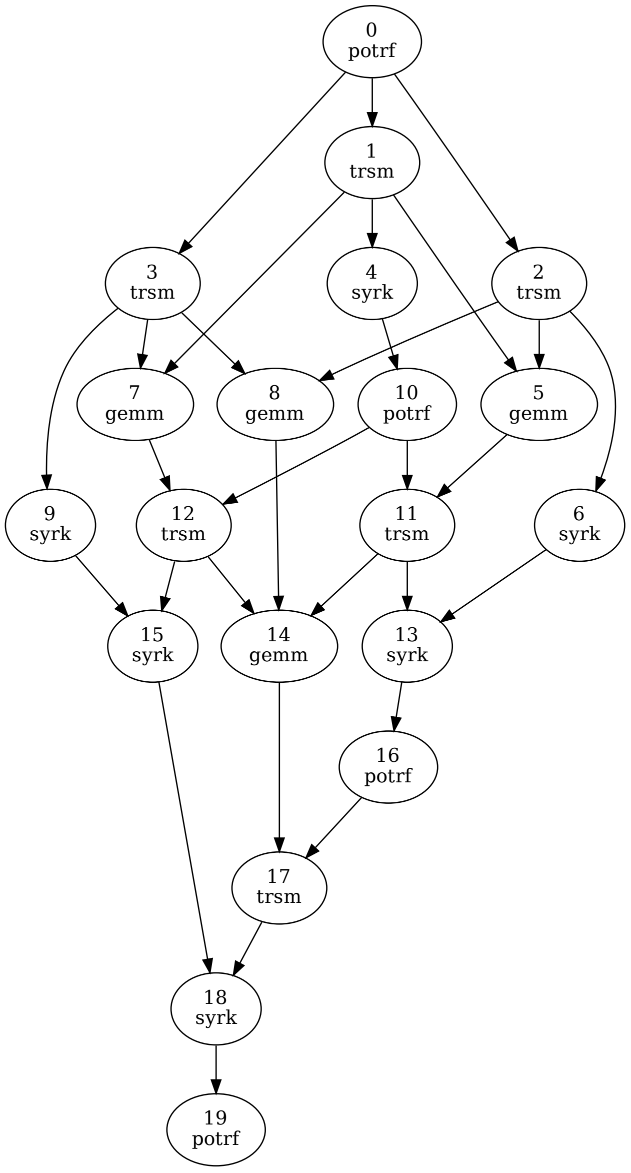


Figure A.3: Cholesky

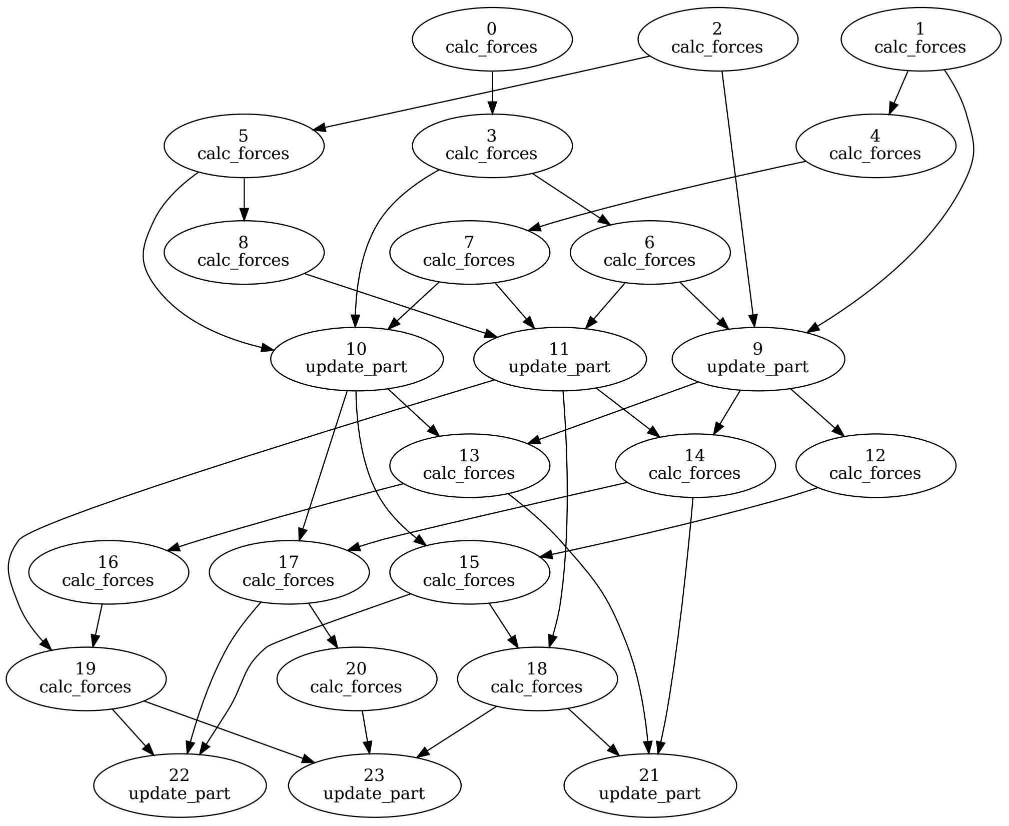


Figure A.4: NBody

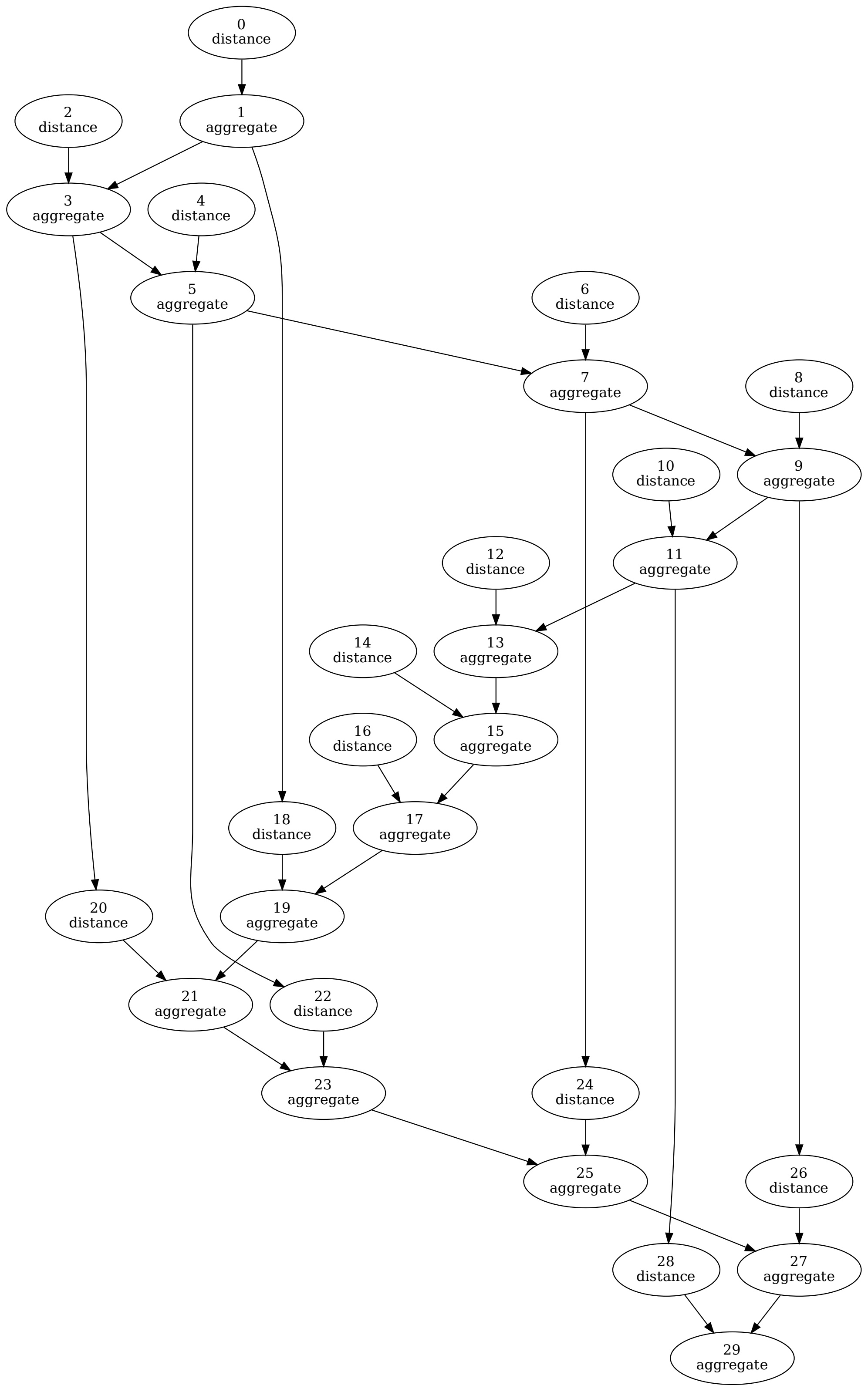


Figure A.5: Spectra

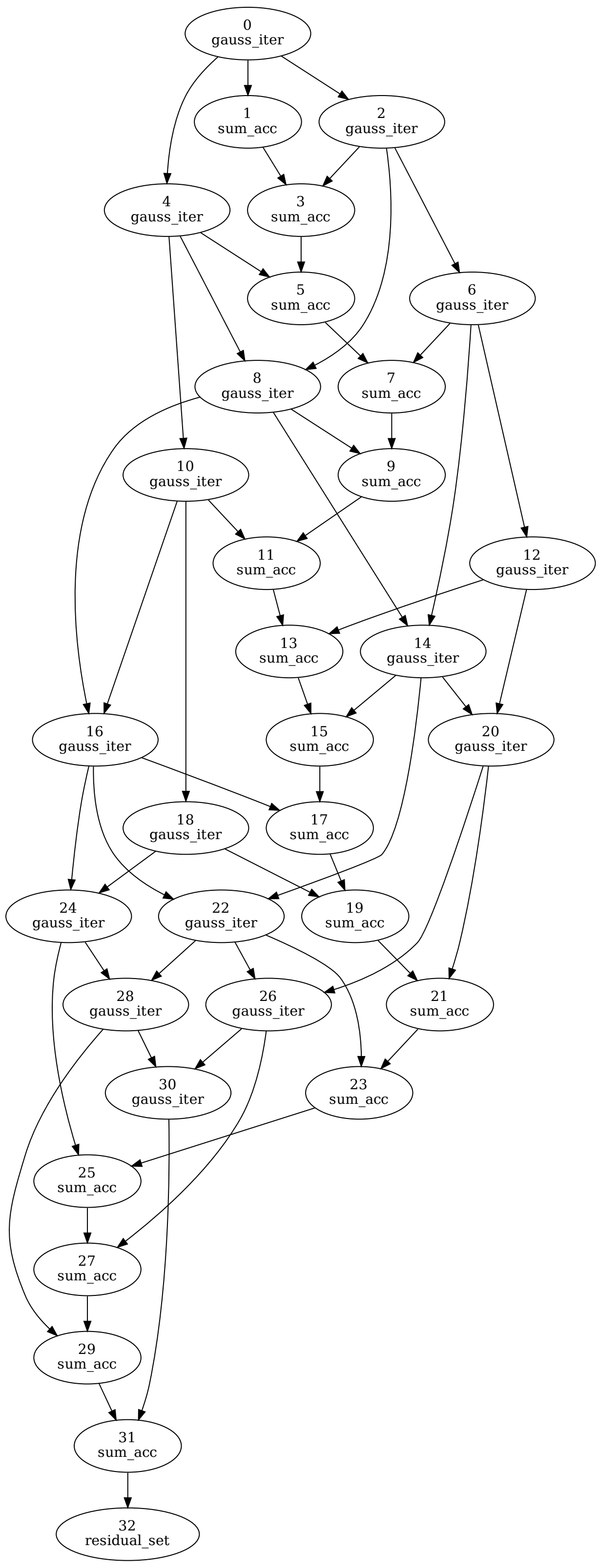


Figure A.6: Heat diffusion

Table A.1: Average task execution time

benchmark	task name	time (100MHz cycles)
matmul 1	gemm	246638
matmul 2	gemm	7645
nbody	calc_forces	128438
	update_part	210
cholesky 1	gemm	5950
	potrf	3456
	syrk	4622
	trsm	3881
cholesky 2	gemm	0
	potrf	0
	syrk	0
	trsm	0
cholesky 3	gemm	651
	potrf	406
	syrk	520
	trsm	434
cholesky 4	gemm	230
	potrf	277
	syrk	218
	trsm	169
rand graph 1	task	100
rand graph 2	task	100
spectra	aggregate	81
	distance	4457
heat	gauss_iter	720
	residual_set	23
	sum_acc	4