Seventh Framework Programme – Theme 3.6: Computing Systems

Grant agreement n°248776

D1.6
"VPI Extension for Architectural Analysis & Design Space Exploration"

WP1: "From 3D Opportunities to 3D Manycore Architectures"

Editor: Martino RUGGIERO, University of Bologna

Date: Thursday 5th July, 2012

1	CEA (Coord.)	Commissariat à l'énergie atomique et aux énergies alternatives, Laboratoire d'électronique et des technologies de l'information
2	UJF	Université Joseph Fourier Grenoble 1 – VERIMAG
3	ETHZ	Eidgenössische Technische Hochschule Zürich
4	UNIBO	Università di Bologna
5	STM	STMicroelectronics
6	EPFL	École polytechnique Fédérale de Lausanne

Version: v2.1 – Status: Delivered
CEA Ref. : LETI/DACLE/12-0498

Versions of the Document

Version	Date	Author	Comment
1.0	26/06/2012	M. Ruggiero	First version
2.0	04/07/2012	M. Ruggiero	Final version
2.1	05/07/2012	M. Ruggiero	Minor fixes

Contributors

Contents

1 Introduction

The main goal of this deliverable is represented by the introduction and detailed description of the latest improvements and extensions made on the PRO3D virtual platform in order to enable efficient P2012-based full-system simulation for efficient and accurate architectural analysis and design space explorations.

Driven by flexibility, performance and cost constraints of demanding modern applications, heterogeneous System-on-Chip (SoC) is going to be adopted as the dominant design paradigm in the various computing domains. SoC architecture and heterogeneity clearly provide a wider power/performance scaling, combining high performance and power efficient general-purpose cores along with massively parallel manycore-based accelerators [1, 2, 3, 4]. The idea behind P2012 is clearly conform with this architectural template. P2012 has been indeed thought to be coupled with an ARM-based host, which behaves as general-purpose master processor, while the P2012 fabric as accelerator slave engine. Besides the complex hardware, generally this kind of platforms hosts also a complicated software eco-system, composed by an operating system, several communication protocol stacks, and various high-computational demanding user applications.

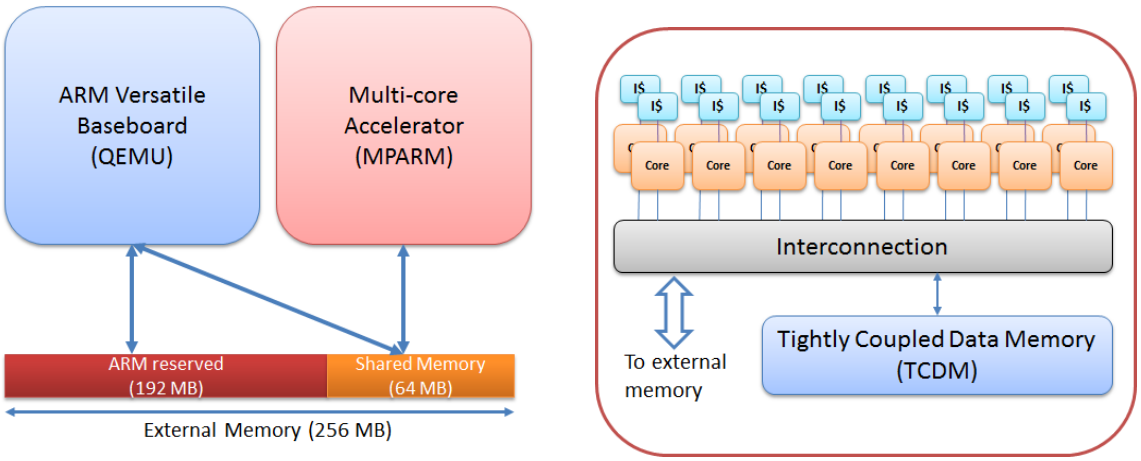
The necessity to efficiently cope with the huge HW/SW design space provided by this scenario makes clearly full-system simulator one of the most important designing tools. Full-system simulation, which emulates the entire software stack including OSes, libraries and user-level applications, is indeed extremely useful in serving the increasing need of pre-HW development of system software, characterizing performance bottlenecks, exposing and analyzing possible software bugs. QEMU [6] is one of the most popular emulator platforms, since it emulates several architectures (e.g. x86, ARM, etc.). QEMU is a fast emulation environment based on dynamic binary translation. The binary translation represents the emulation of the instruction set of a processor by the instruction set of another processor using code translation. The goal of the binary translation simulators is to be very fast and functionally correct, but not to provide information about hardware execution time. Even though QEMU can emulate a full-system, it lacks of accurate coprocessors simulation capabilities. QEMU does not provide any ways to take into account the simulation of a heterogeneous SoCs.

Moreover, when tackling the efficient virtual prototyping and design of such heterogeneous complex architectural solutions, flexibility and a good level of accuracy are indeed required since HW designers need to explore a large solution space by comparing a huge number of different design options, so as to identify an optimized system configuration. Therefore, advances in simulation technologies are essential to increase engineer productivity and decrease time to market. SystemC is a well-known HW description language frequently used in this context. However, SystemC simulation speed is very low when simulating accurate hardware models with a full-fledged operating system up and running.

Several approaches trying to link QEMU and SystemC [7] environments have been proposed as a software/hardware SoC emulation framework to leverage the strengths of QEMU and SystemC [8, 9]. QEMU CPU emulation can be used to reproduce the behavior of target processors while SystemC and its simulation environment can be used for the simulation HW devices under design. QEMU already supports the use of host-mapped devices, while SystemC supports the HW description at many abstraction levels. Although co-design can strongly benefit from such a potential winning combination, several constraints make this breaking through approach targeting complex HW/SW design space explorations extremely inefficient and limited. Previous works indeed suffer of several drawbacks, implicitly generated by the selected interfacing paradigm. Drawbacks belong both to performance and flexibility. The optimal approach should clearly provide an asynchronous interface with a non-blocking semantic, thus

inherently enabling parallel execution between QEMU and SystemC environments. Moreover, a lightweight communication and synchronization infrastructure among the two main simulation frameworks is necessary. It also should enable the sharing of the same memory hierarchy (like L2 or RAM).

Part of Work Package 1 of the PRO3D project focuses on the definition and development of simulation tools for functional modeling of 3D IC, such the upcoming P2012 from STM. In this deliverable, we present the enhancements developed on the PRO3D virtual platform in order to enable efficient and accurate full-system simulations. We present in this report a new emulation framework based on QEMU and 3D-MPARM which overcomes the mentioned issues of previous works. The main outcome of our approach has been the possibility to have QEMU and 3D-MPARM running effectively in parallel, allowing the programmer to catch the real behavior of the architecture and to design its software accordingly. In order to achieve our purposes, we have enhanced the PRO3D virtual platform (i.e. 3D-MPARM) with a series of extensions aimed at enabling efficient P2012-based full-system simulation for architectural analysis and design space explorations.



(a) (b)

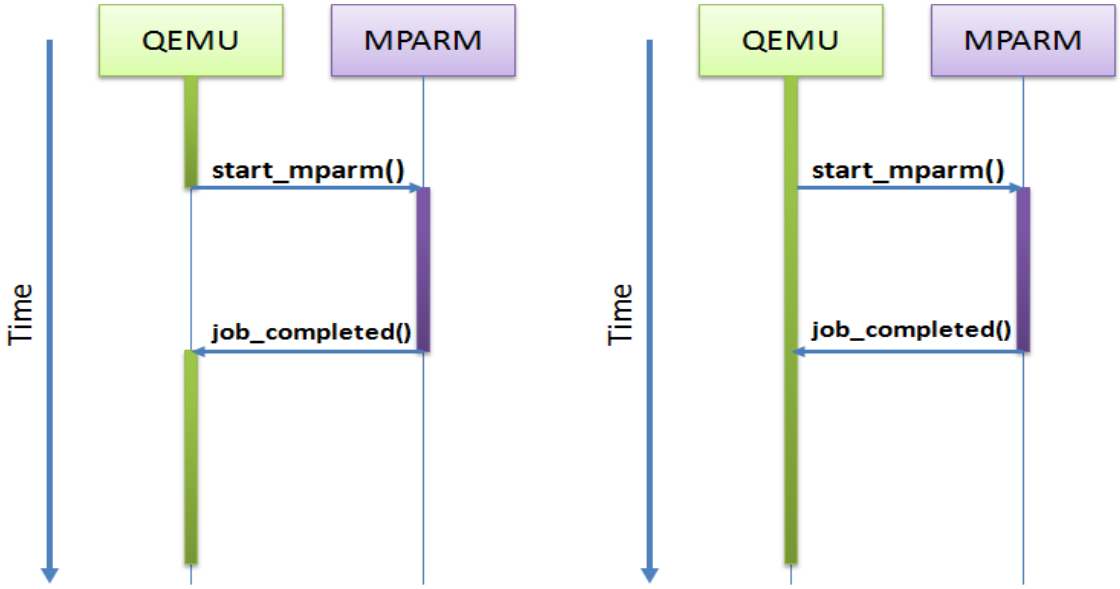
Figure 1: a) Target simulated architecture. b) Coprocessor generic architecture

2 Conceptual Architecture

During the last years, ever a large number of silicon manufacturers are spending high design efforts to create architectures able to provide an ever increasing computing power at a very reduced power budget. This trend is driven by modern applications, requiring increasing real-time computing capabilities and facing at the same time with shrinking constraints on power budget, and of course on device's battery lifetime. Examples and results of this evolution are AMD Fusion [1], NVidia Tegra [2] and P2012 [4]. The mentioned examples feature a general purpose multi-core processor accompanied by a more energy efficient and powerful many-core accelerator. In those kind of systems the general purpose processor is intended as a coordinator and is in charge of running an operating system, while the many-core accelerator is used to speed up the execution of computing intesive applications, or parts of them. Although their great computing power, accelerators are not able to run operating systems due to the lack of all needed surrounding devices and due to the simplicity of their micro-architectural design. The general purpose processor and the accelerator often share the main memory.
The architecture targeted by this work is inspired by the mentioned examples and composed in detail as follows:

- ARM Versatile Platform Baseboard;
- P2012-based accelerator.

The ARM baseboard is emulated by QEMU and is composed by an ARM926 processor, featuring an ARM v5 ISA, and a group of peripherals needed to run a full-fledged operating system (e.g. uarts, network controllers, real time clock). The P2012 accelerator is modeled using the PRO3D virtual platform based on MPARM [5], a SystemC based cycle-accurate MPSoC simulator. As shown in Fig. 1a the ARM processor and the co-processor share a portion of the external memory (64MB over a total of 256MB), which is also used as communication medium between them. The co-processor target architecture is shown in Fig 1b, it features clusters (only one depicted in the figure) of several homogeneous cores with segregate I caches and all sharing a Tightly Coupled Data Memory (TCDM) accessible via an interconnection medium.



(a) (b)

Figure 2: a) Sequentialized execution. b) Parallel execution.

3 Implementation

In this section we provide a complete description of the extensions introduced in the PRO3D virtual platform, as well as the required QEMU modifications. As first the idea behind the implementation will be introduced and then each element is described in detail.

3.1 Basic Idea

As introduced in Section 2 the main interest of software developers is focused on the interation between the host processor and the co-processor. In a real platform those two actors can execute in an asynchronous and parallel fashion and this behavior has to be captured when designing a virtual platform. The basic idea is then to design a virtual platform in which the host processor system and the accelerator can run in parallel, this implies having QEMU and 3D-MPARM running on different threads or processes. Other approaches interfacing QEMU with SystemC models apply normally two possible schemes: wrapping QEMU as a SystemC module or adding to QEMU a device in charge of starting the SystemC simulation whenever needed. The first approach is detrimental for QEMU's performance, leading to a complete lost of binary translation benefits. While the second approach leads to the sequentialization of the execution of QEMU and the SystemC model (Fig. 2a). Such type of implementation is obviously unacceptable when the focus is the interaction between host processor and accelerator.

Our proposal is to have QEMU and 3D-MPARM running effectively in parallel, allowing the programmer to catch the real behavior of the architecture and to design its software accordingly. Another achievement of this work is the definition of a time synchronization mechanism, able to provide a global simulation time for the entire full-system virtual platform.

3.2 Having QEMU and 3D-MPARM running in parallel

Before having 3D-MPARM and QEMU running in parallel it is necessary to understand wether to use two different processes or threads. Taking a look at the target architecture we can see that

the host processor and the co-processor need to share a portion of the external memory. Both QEMU and 3D-MPARM as standalone applications instantiate an external memory device, where the effective memory data is designed as a byte array. Due to major flexibility and also a major integration inside the platform, we decided to keep the global external memory of the platform inside QEMU's infrastructure, modifying 3D-MPARM accordingly. What we did is to pass 3D-MPARM, at platform startup, the pointer to QEMU's memory array shifted by 192MB (offset to shared memory segment). Given this assumption it is obvious why our choice felt on using two different threads instead of processes, we of course need to share the memory and heavy processes cannot do that.

The main simulation environment is essentially represented by QEMU, which during the platform initialization phase creates another thread in charge of executing 3D-MPARM and to start the SystemC simulation. Here we had to modify the Versatile PB initialization code in order to add the creation of 3D-MPARM's thread, passing at creation time a set of parameters comprising the pointer to QEMU's external memory, correctly shifted to the start of shared memory segment. Whe chose to split the main memory in two segment to protect the Linux operating system memory. The portion of memory which is not shared between QEMU and ARM is also accessible only from the host processor, avoiding any possible interference with possible co-processor's accesses.

The main modifications to obtain the parallel execution have been applied on 3D-MPARM. It is a SystemC model and is not meant to run on multithreaded applications. A standard SystemC model is a normal C++ application which has the *main()* method substituted by the *sc_main()* method, this because the main of the final application is hidden inside the SystemC library. So, when you run a SystemC simulation the execution starts from the main inside the library which in turn will call the *sc_main* of your model. This behavior is not good fo our goal, we want to run the SystemC simulation from another thread created by QEMU and the only *main* function allowed is the one of QEMU. To overcome this problem we slightly modified the SystemC library sources, removing the main function and introducing the *init_systemc_main()* method. After this modification, when 3D-MPARM thread starts it also calls the *init_systemc_main()* launching in turn the SystemC simulation. It is important to highlight that the 3D-MPARM SystemC simulation starts immediately at QEMU startup, and the accelerator starts executing the binary of a firmware (until the shutdown), in which all cores are waiting for a job to execute. This firmware will be described in detail in Section 3.4.2.

3.3 Time synchronization

QEMU and 3D-MPARM run independently in parallel with a different notion of time, leading to only functional simulations due to the lack of a global time management. Application delevolpers often need to understand how much time, over the total application time, is spent on the host processor or on the accelerator. Also, without a global simulation time it is not possible to appreciate execution time speedups due to the exploitation of the many-core accelerator.

To manage the time synchronization between the two environments it is necessary that both QEMU and 3D-MPARM have a time measurement system. As QEMU does not provide this kind of mechanism we instrumented it to implement a clock cycle count, based on instructions executed and memory accesses performed. On the contrary for 3D-MPARM there is no need for heavy modifications because it is possible to exploit the SystemC simulation time.

The implemented synchronization mechanism is based on a threshold protocol acting on simulated time: from an initial time t_0 both QEMU and 3D-MPARM start counting the number of clock cycles elapsed, with each actor maintaining its own count. At fixed synchronization points, cycle counts from QEMU (c_Q) and 3D-MPARM (c_M) are multiplied by the respective

clock period (p_Q and p_M) and compared. Given a time threshold h if $|c_Q * p_Q - c_M * p_M| > h$, one of the two systems is forward in the future in respect to the other and will be blocked until $|c_Q * p_Q - c_M * p_M| > 0$.

One can comment that with the proposed mechanism the virtual platform is globally slowed down, due to synchronization points. This is true and is the price to pay for increasing accuracy. Anyhow programmers not always need time measurements during applications design, and we provide an interface to activate and de-activate time synchronization. Another possible comment is related to accuracy. Stopping the platform with a higher simulation time leads to inaccuracy due to possible delays on events exchanged by QEMU and 3D-MPARM. Consider as an example that QEMU is faster and is waiting an event from 3D-MPARM. If while waiting for the event the simulated time gap becomes higher than the threshold, QEMU is stopped and 3D-MPARM goes ahead running to fill the gap and produce the event. When QEMU is resumed it sees the event arrived at simulation time t_r, while 3D-MPARM produced it at $t_p \leq t_r$. If $t_p < t_r$ the global simulation time has an error equal to $t_r - t_p$. The error depends on the value of the threshold and is anyway acceptable because with this virtual platform we are not targeting hardware developers, so we don't need cycle level accuracy. It is important to highlight that this error does not introduce functional inconsistency, but only a skew on the simulation time. Results of slowdown due to time synchronization will be presented in Section 4.

3.3.1 Instrumenting QEMU

QEMU is a machine emulator able to model different architectures at a very high level of abstraction and is intended to run full-fledged operating systems with a high performance. At the actual state it is not possbile to extract from QEMU a cycle count based on instruction executed and memory accesses. To extract a meaningful cycle count during the execution we need to take into account at least the pipeline model, to calculate instructions duration, and the memory model (at least caches). We modified QEMU TCG (Tiny Code Generator) to calculate at runtime the cycles spent from the ARM processor. During execution QEMU translates blocks (basic block) of target instructions in an intermediate representation which is in turn translated to host machine code, saved into TCG cache, and then executed. With each target instruction translated in one or more intermediate instructions. Our Intervention happened on the first stage of translation: during the translation of each basic block, instructions are taken one after the other and translated in the corresponding intermediate representation. In this phase we introduced a function which is able to calculate the number of cycles needed to execute each instruction. Starting from the binary representation of instructions and taking instructions timing from the ARM926 manual, this function is able to calculate the number of cycles needed to execute a certain instruction. The count is also taking into account registers interlock. Once the cycle count is obtained we save it in a tcg temporary variable, inside the translated block, which will be available during the execution of each translated block. Figure 3 shows the modifications applied to the translated block when saving cycles count, note that no extra code is added to the block. This solution fully exploits QEMU's TCG cache because in case of multiple execution of the same block, instructions duration does not need to be re-calculated.

During the execution of translated code a variable internal to QEMU, `sim_time`, maintains the cycle count of the ARM processor and is incremented at each ARM instruction executed. The increment is based on the values already annotated inside the translated block. Once the target code is translated into the host code there is no way to add extra function call inside of it, so there is no way to add the function call which updates `sim_time`. This modification have to be done before, during the first traslation of ARM code into intermediate code. QEMU provides a

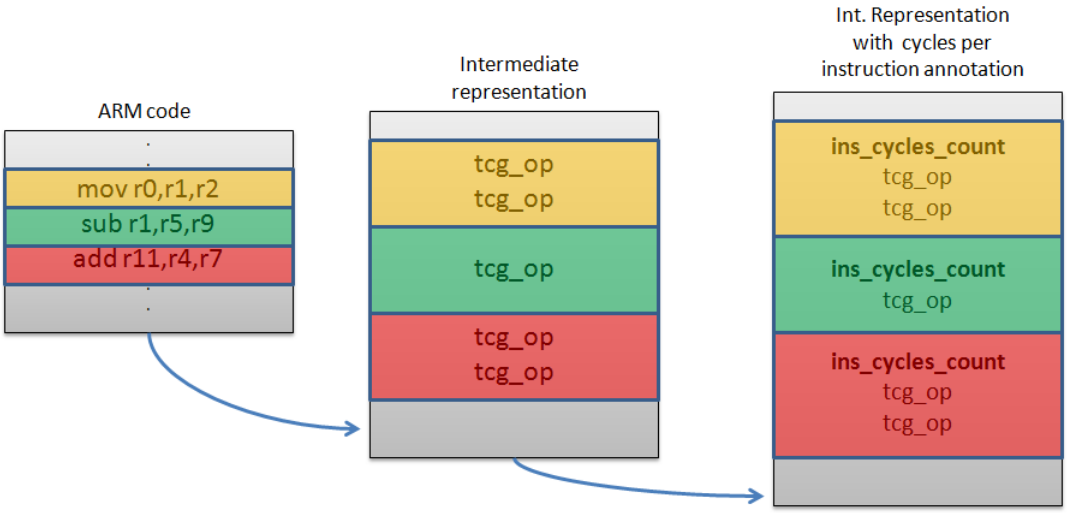


Figure 3: Modifications to qemu's translated blocks due to instructions cycle count annotation

mechanism to introduce function calls inside a translated block, called *helper functions*. Helper functions can access all internal structures of QEMU (e.g. the `sim_time` variable), and all informations embedded in a translated block (e.g. the temporary tcg variable `ins_cycle_count`). To define a helper function a macro DEF_HELPER_n(f_{name}, t_r, t_{p1}, t_{p2}, ..., t_{pn}) is used, where n is the nubmer of parameters, f_{name} is the name of the function, t_r is the type of return parameter and t_{pn} is the type of the n-th parameter. To invoke an helper function during the execution of a translated block, it is sufficient to insert *gen_helper_f()* during the translation of the block itself.

Now let's assume that our defined helper is named `helper_update_count()`, it is sufficient to put `gen_helper_update_count()` during the translation of each ARM instruction to have it called at runtime for each executed ARM instruction. The result of each call is to add the value contained in the tcg temporary variable (`ins_cycle_count`) to the global cycle count (`sim_time`).

Once QEMU is able to calculate a cycle count, this value must be used for the synchronization protocol. As already explained in last paragraph QEMU and 3D-MPARM exchange simulated time at fixed synchronization points, defined for this work at each executed ARM instruction. The synchronization happens using QEMU's helper functions, at each ARM instruction the helper is called passing as an argument the actual QEMU's cycle count. The implementation of this function is straightforward, it invokes a call-back function exposed by 3D-MPARM passing to it the simulation time measured until that moment. In case of need QEMU will be stopped waiting for 3D-MPARM. The implementation of the call-back function will be discussed in the next paragraph.

A final note regards the performance. Our solution implies the execution of two helper function calls per each ARM instruction executed by QEMU, it of course slows down the simulation speed of QEMU and so of the entire platform. Note that the syhchronization mechanism is not always needed, it is sufficient to consider the case when a programmer only wants to test the functional correctness of his application. In that case simulation speed is very important while the accuracy is not. We provide a set of APIs allowing the programmer to enable and disable this mechanism, and to access the simulation time from a user-space application running on top of QEMU. The synchronization is disabled as default parameter, in which case no helper

function is inserted into translated blocks.

3.3.2 Instrumenting 3D-MPARM

For 3D-MPARM, modifications needed are really lightweight. It is already able to measure time and we only needed to implement two call-back functions used from QEMU side, and a simple module used to stop 3D-MPARM in case of need and to count clock cycles elapsed when synchronization is enabled. The call-back function, executed by QEMU's thread, is in charge on implementing the synchronization mechanism. If QEMU is in advance, it will be stopped until the difference between the two clocks is not zero. Otherwise, if 3D-MPARM has to be stopped, a flag is set inside 3D-MPARM and the SystemC simulation is stopped at the next clock cycle. A second call-back has been defined to access from QEMU 3D-MPARM's clock cycles count.

The new SystemC module, *synch_module*, is in charge of counting clock cycles elapsed from time t_0 (See Section 3.3), holding this count in a variable accessible via the call-back and used by QEMU to check wheter someone should be stopped. Beside the cycle count this module is also in charge to check whether 3D-MPARM clock has to be stopped due to synchronization, and if this is the case the *synch_module* does not finish its execution until the simulated times are aligned again. The *synch_module* is sensible to the clock of 3D-MPARM architecture and then it is scheduled by SystemC at each clock cycle. With this method we are easily able to stop the SystemC simulation for the time needed.

Given that both 3D-MPARM and QEMU threads access concurrently a set of variable, we surrounded those accesses with a crititical section to avoid any possibile race condition.

3.4 Software stack

In this section we provide a complete description of the software stack used to allow the programmer to fully explot the accelerator from within the operating system running on top of QEMU.

3.4.1 Memory map

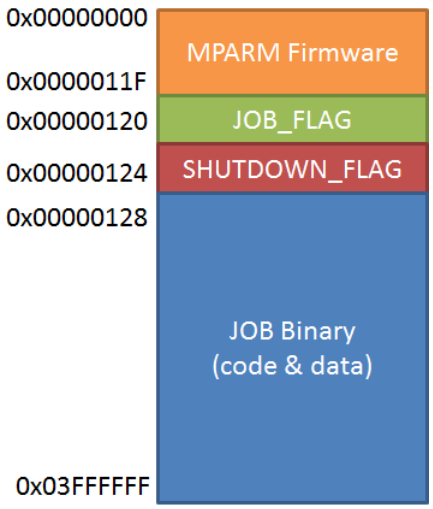


Figure 4: Shared memory segment organization

Before starting the description of the software stack it is important to describe the memory map of the virtual platform. All the software layers of this system are based on top of it.

Figure 4 shows the organization of the shared memory segment. At the very top of this region we put the binary of the firmware, this is where 3D-MPARM starts fetching code at platform startup. Then two flags are defined: *JOB_FLAG* and *SHUTDOWN_FLAG*. The former is used by the host processor to signal the accelerator that there is a new job to run, the latter is used to shutdown the platform. Finally the rest of this segment is reserved to contain code and data of the actual job.

3.4.2 3D-MPARM Firmware

As already introduced, at simulation startup 3D-MPARM starts executing the binary of a firmware. This firmware is necessary to correctly manage the accelerator from the host processor. The firmware is executed by all 3D-MPARM cores, and during its executions cores are waiting for commands from QEMU such as a new job to execute. Beside the management of jobs this firmware is also in charge of managing accelerator's instruction caches, especially regarding the invalidation of caches between the execution of two different jobs. For the design of this virtual platform we had to modify 3D-MPARM's execution model, originally it takes a single binary which is executed until the end of the simulation. Running two binaries meant two run two different simulations. Now we run 3D-MPARM with a first binary which is the firmware, and then it will run until the shutdown of the entire platform. Jobs are pushed at runtime putting their binary at a fixed location (in the Job Binary region) and then notifying the firmware, via th *JOB_FLAG*, that there is a new job to execute. Between the execution of each job the instructions caches are not invalidated and it happens that pushing a new job will find I-caches valid but with the wrong code, the one of the last job. To overcome this problem we force the invalidation of I-Caches before the execution of each job. The following snippet shows a general implementation of 3D-MPARM's firmware.

```
while(!shutdown){
  while(!execute_job);
  invalidate_Icache();
  job();
}
```

3.4.3 Linux Driver

In order to build a full system simulation environment we mapped 3D-MPARM as a device in the device file system of the guest Linux environment running on top of QEMU. A device node `/dev/mparm` has been created, and as all Linux devices is interfaced to the operating system using a Linux driver. The driver provides all basic function to interact with the device:

- open,
- close,
- read, write,
- ioctl,
- llseek

At kernel boot the memory available to the kernel is 192MB (256MB - 64MB) and the shared memory region is not still visible to the system. The first operation performed by the driver is to map this region into the kernel memory space using an `ioremap()` call. Henceforth all accesses to the shared memory region can only happen via `ioread` and `iowrite` calls from the kernel space, and from the userspace using driver's `read`, `write` and `llseek`. The read and write functions allow to read and write the shared memory segment, for example the write function is used when copying the binary of a job from the application userspace to the shared memory segment. As kernel cannot directly access userspace memory buffers it was necessary to use the `copy_from_user` and `copy_to_user` functions, which check permissions and validity of the address provided before effectively accessing the memory. It is important to notice that read and write functions can only access the Job Binary section(Fig 4), while the control locations will be accessed only through an `ioctl` call.

The `ioctl` function works as control interface for the accelerator, it allows to:

- Reset the shared memory segment: writes 0 in all locations.
- Start the execution of a job: writes 1 at JOB_FLAG address
- Shutdown the system: writes 1 to SHUTDOWN_FLAG address
- Check the state of the simulation: reading at address JOB_FLAG, this location will contain 1 until the end of the job
- Control the synchronization mechanism: start, stop and get actual simulation time.

The last point is the only needing a deep explanation. To start or stop the synchronization and to read the actual cycle count, it is necessary to cross the line between the target simulated operating system and the simulation environment to trigger all strucures and mechanisms necessary to perform the synchronization. The crossing phase is implemented using *semihosting*.

Semihosting is a technique developed for ARM targets allowing the communication between an application running on the target and a host computer running a debugger. In other words, it enables the redirection of all the application's IO system calls (e.g. printf()) to a debugger interface. The essence of semihosting is a particular software interrupt (svc 0x123456) which when caught from a debugger interface is redirected to the host machine (CPU platform) instead of otherwise simulating on target platform. This interrupt takes two parameters in registers r0 and r1. The r0 carries the semihosting operation identifier which are pre-defined by ARM as well as our custom-defined identifier, `SYNCH_HANDLING_ID`.

QEMU is already capable to handle semihosting calls, so we only had to extend the existing structure in order to manage call for the `SYNCH_HANDLING_ID` semihosting identifier. The second parameter in r1 carries the operation identifier, QEMU performs the desired operation (start or stop synchronization, and read simulated time) according to that value.

3.4.4 Userspace library

To simplify programmer's job we have also designed a user level library, which provides a set of APIs that rely on the Linux driver functions. This API can be divided in two different layers based on abstraction level: low and high level functions.

The low level functions map directly driver's operations and can be summarized as follows:

- `int open_device(int flags);`
- `void close_device(int fd);`

API's function name	Behavior
`load_job_binary(char * filename);`	Load the binary of a job into shared memory segment
`void start_acceleration();`	Starts the execution of a job on the accelerator
`int check_job_status();`	Return non zero if the accelarator is stil running, define a user level synchronization point between host processor and accelerator
`void start_synchronization();`	Ativate the synchronization mechanism
`void stop_synchronization();`	Deactivate the synchronization mechanism
`uint64_t get_simulated_time();`	Get the actual simulation time, synchronization must be activated

Table 1: High level API's functions description

- `void mwrite(void *buff, int size, off_t off);`
- `void mread(void *buff, int size, off_t off);`

Beside the obvious `open_device` and `close_device`, `mread` and `mwrite` allow the user to write a buffer in userspace to the shared memory segment. Table 1 shows a list of high level API's functions which of course are based on the low level ones. The last three entries of the table, enable the user to manage the synchronization mechanism and are all implying semihosting calls.

4 Experimental Results

In this section we present a set of experimental results aimed at validating the proposed virtual platform. The focus of the benchmarks used it is not to test how they run on the ARM core, or to evaluate the scaling one would obtain running them on top of the accelerator; but rather we want to evaluate the interface between QEMU ad the SystemC model(3D-MPARM) and also the synchronization mechanism presented in Section 3.3.

4.1 Experimental setup

Table 2 summarizes the experimental setup of the virtual platform used for all benchmarks discussed.

We chose as ARM core clock frequency 1GHz, even if the ARM modeled by QEMU is a ARM926 works at up to 500MHz, to resemble a state of the art ARM processor performance. The frequency would only affect results in terms of global values, all considerations done in this section stay valid even changing the ARM core clock frequency.

4.2 Host side benchmark structure

To fully understand results presented in this chapter it is important to catch the structure of the benchmark, especially on the host processor side. This helps the reader to understand considerations done regarding the interaction between QEMU and 3D-MPARM.

Parameter	Value
ARM Core clock frequency	1GHz
3D-MPARM Cores clock frequency	250MHz
# 3D-MPARM Cores	up to 16
Guest OS	Debian for ARM (Linux 2.6.32)
Time synchronization threshold	100 simulated nsecs

Table 2: Experimental Setup

The following code snippet shows the structure of the ARM side of our benchmarks.

```
void main(){

  do_something();

  printf("Benchmark start time %llu",get_simulated_time());

  /*Load job binary into shared memory segment*/
  load_job_binary("/path/to/binary");

  start_synchronization();

  /*Start the execution on the accelerator*/
  start_acceleration();

  /*Wait for job to finish*/
  while (check_job_status());

  printf("Benchmark stop time %llu",get_simulated_time());

  stop_synchronization();

  do_something_else();

}
```

The binary of the job to be accelerated stays in the Guest linux file system and is a parallel matrix multiplication benchmark. We take the simulated execution time before the offload of a job to the accelerator, and after the end of the job. This measurement represents the time taken by the benchmark to run on the accelerator, plus an overhead introduced by the user level software stack.

4.3 QEMU slowdown

As described in Section 3.3, our synchronization mechanism is adding a certain overhead over QEMU due to a few helper functions calls per ARM instruction executed. To evaluate this slowdown we used a simple experiment, running the same benchmark in two different cases:

1. synchronization active

2. synchronization not active, but ARM cycle count active

In this way we are able to measure the number of simulated instructions per time unit and to evaluate the slodown due to the synchronization with 3D-MPARM. Table 3 shows the results of this test. The benchmark has the same structure seen in Section 4.2 while the job offloaded to 3D-MPARM has no importance in this test. The only 3D-MPARM's parameter we take care of is the number of processors, the accuracy increases at increasing number of cores and the simulation speed decrease as well.

# 3D-MPARM Cores	SCLK/usec	Slowdown
no synch	55813.5	1x
1	868.3	64x
4	345.6	158x
16	107.4	500x

Table 3: Slowdown due to synchronization, SCLK - Simulated Clock Cycles

The slowdown can reach 500x when 3D-MPARM runs in 16 cores configuration. This is unavoidable because the simulation speed, when the synchronization is active, is directly depedent from the slowest between QEMU and 3D-MPARM. Given that 3D-MPARM is hundreds of times slower than QEMU, the bigger the number of cores in 3D-MPARM the slower is the simulation.

Given the effects of synchronization over the simulation speed we extended the programming API with two more functions: `start_host_cycle_count()`, `stop_host_cycle_count()`. These new functions allows the user to only activate the ARM side cycle count, useful in the case when the programmer is profiling code only running on the host processor and doesn't need to be synchronized with the co-processor. We used this extension in the next section, to measure the time taken by a benchmark running only on the host processor.

4.4 Effects of synchronization threshold

To evaluate effects of the synchronization threshold consider the case where we want to measure the execution time of an application, involving both the host processor and the co-processor. Figure 5 is showing the timeline of such application: after starting the synchronization the host processor run a prolog code, then it offloads a job to the co-processor. While the job is executed by the co-processor the the host processor is waiting for its end. At the end of a job the host executes the epilogue and stops the synchronization. We Consider the total benchmark time ($T = P + W + E$) the one between start and stop of synchronization system, whith P and E the time to execute prologue and epilogue respectively and W the time spent by the co-processor to finish the job. The parallel job is a matrix multiplication, which scales the execution time increasing the number of cores available in the coprocessor. Running the whole application varying the number of accelerator's cores, we expect to see $P + E$ constant while W scaling with the number of cores.

In our experiment we run the benchmark described above, with various accelerator's configurations and with three different values of the synchronization threshold (100 ns, 100us, 100ms). Figure 6 shows the trend of the $P + E$ component, while the job time is not reported because it is not useful for this analisys. What we expect to see is that the global benchmark time is equal to the parallel job time plus a fixed overhead ($P + E$), the closer are those values the

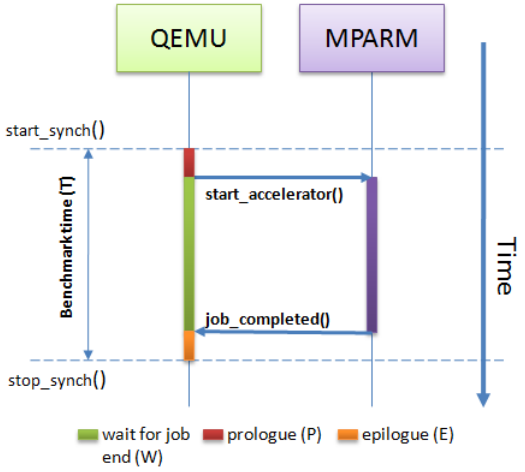


Figure 5: Application time line

better is the time measured. The $P + E$ component increase when increasing the synchronization threshold, while stays constant at varying number of accelerator's cores. This happens because increasing the threshold we increase the possible skew between the clock of QEMU and 3D-MPARM, dilating the simulation time. At the same time the parallel job execution time is not affected by the threshold, and its duration is measured from within 3D-MPARM.

It is important to notice that with a 100ms threshold the value reported in the plot is not constant at increasing number of cores. The strange trend is due to a combination of the threshold value and the duration of the parallel job, which decrease at increasing number of cores. From now on we used 100ns as synchronization threshold value, as it showed the best results.

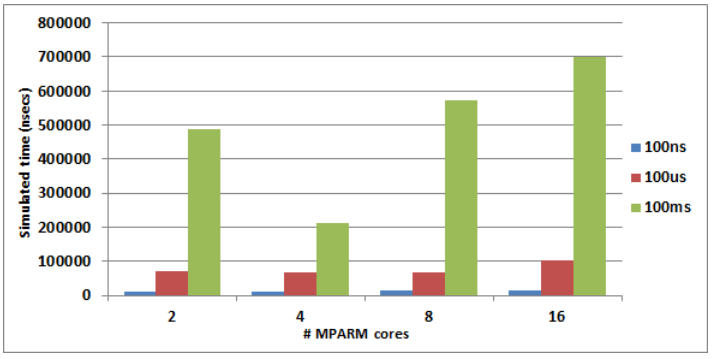


Figure 6: Synchronization threshold effect, the plot reports only the P+E time for each configuration

4.5 Validation of parallel execution

Figure 7 shows the results of running the matrix multiplication benchmark varying the number of cores available on the accelerator. The benchmark scheme is always the one described in Section 4.2, but this time we measure two different times. The plot shows results for both the time measured from QEMU, and the time measured from within 3D-MPARM. The latter is

just the time taken by the accelerator to execute the matrix multiplication algorithm.

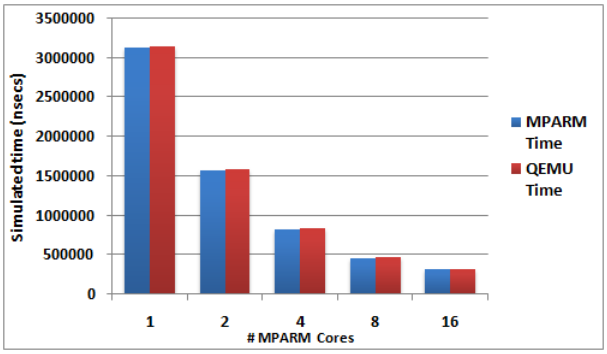


Figure 7: MMULT

The difference between the time measured by QEMU and the time measured by 3D-MPARM gives an idea of the time spent running the software API used to offload the binary, start the accelerator and wait for the completion of the job. This overhead represents less than the 1% of the total runtime and is ~ 13 simulated micro-seconds. An important observation is that the overhead is, as expected, constant even varying the number of cores in 3D-MPARM. This result means that the interface between QEMU and 3D-MPARM is not influenced by the architecture modeled by both. This feature is crucial for programming models developers, who want to know the overhead introduced by their software layer.

To conclude, the fact the the time measured by QEMU is really close to the one measured by 3D-MPARM plus a little overhead due to the software API (Fig. 8), confirms that they run in effective parallelism. If QEMU and 3D-MPARM were running in sequence, given that the global simulated time for the platform is the one provided by QEMU, we would not see the 3D-MPARM execution time on the one measured by QEMU.

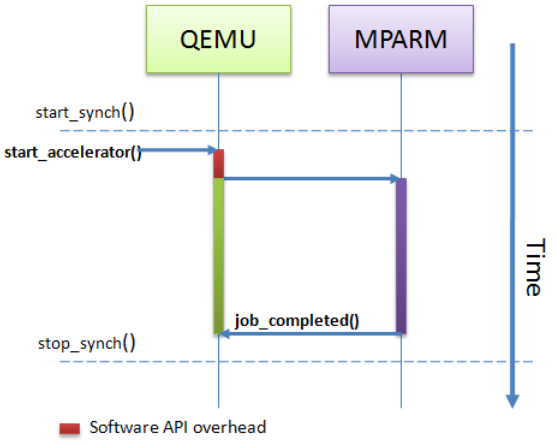


Figure 8: Overhead of the Userspace sw library

4.6 Evaluation of speedup due to exploitation of the co-processor

To validate our time measurement mechanism we wore the hat of a user of the proposed platform, a programmer. We want to know the speedup achievable when accelerating a set of algorithmd

onto the many-core accelerator present in the platform. The algorithms chosen are: Matrix Multiplication, RGBtoHPG color conversion, and image rotate algorithm.

As a first test we ran all three benchmarks on the ARM processor only, thus without any multicore acceleration. In a second test, using our programming API, we offload the execution of of each parallel benchmark to the accelerator. In both cases we measure the duration of the benchmark from the ARM side, using a scheme similar to that in Section 4.2.

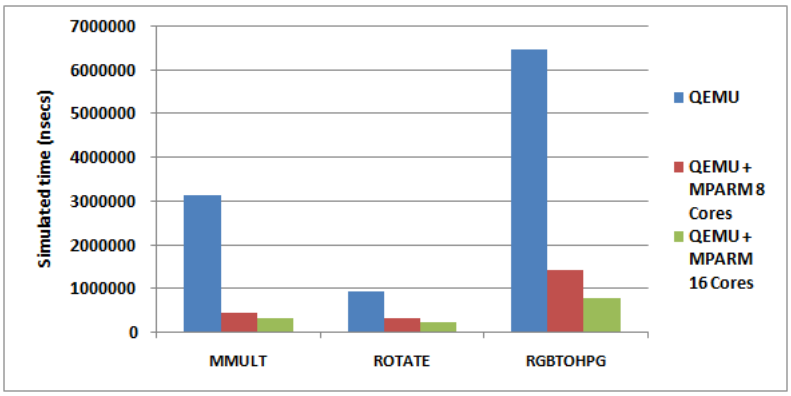


Figure 9: Simulated time speedup due to accelerator exploitation

Figure 9 shows the results of this experiment and prove that the proposed synchronization mechanism allows the programmer to appreciate the speedup achieved offloading a job to the accelerator. Using 3D-MPARM with 8 cores we can see a speedup of ~ 3 times for the matrix multiplication, ~ 3 for the rotate benchmark and ~ 5 for the RGBtoHPG benchmark. While if running 3D-MPARM with 16 cores we can appreciate an almost double execution speedup for all the proposed benchmarks.

5 Conclusion

In this deliverable, we have presented a new emulation framework based on QEMU and 3D-MPARM. The main outcome of our work has been the possibility to have QEMU and 3D-MPARM running effectively in parallel, allowing the programmer to catch the real behavior of the architecture and to design its software accordingly. In order to achieve our purposes, we have enhanced the PRO3D virtual platform (i.e. 3D-MPARM) with a series of extensions aimed at enabling efficient P2012-based full-system simulation for architectural analysis and design space explorations.

References

[1] AMD Accelerated Processing Units. `http://www.amd.com/us/products/technologies/fusion/Pages/fusion.aspx`.

[2] NVIDIA Tegra 3 quad-core and Tegra 2 dual-core processors. `http://www.nvidia.com/object/tegra.html`.

[3] Snapdragon processors by Qualcomm. `http://www.qualcomm.com/snapdragon`.

[4] Programming Heterogeneous Many-core platforms in Nanometer Technology: the P2012 experience. `http://www.artist-embedded.org/docs/Events/2010/Autrans/talks/PDF/Benini/BeniniAutrans10.ppt.pdf`.

[5] The MPARM virtual platform. `http://www-micrel.deis.unibo.it/sitonew/research/mparm.html`.

[6] QEMU: Open source processor emulator. `http://wiki.qemu.org/Main_Page`. SYSC

[7] SystemC: The Language for System-Level Modeling, Design and Verification. `http://www.accellera.org/downloads/standards/systemc/about_systemc/`.

[8] QBox : QEMU based Full System Virtual Platforms. `http://www.greensocs.com/projects/QEMUSystemC`.

[9] Ming-Chao Chiang, Tse-Chen Yeh and Guo-Fu Tseng, "A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development". Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2011 `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5737847&tag=1`.