Technical University of Crete

Diploma Thesis

A Full System Simulator for Disaggregated Computing Platforms and Cloud Data Centers

Konstantinos N. Kyriakidis

July 26, 2017

A thesis submitted in fulfillment of the undergraduate requirements in the:

School of Electrical and Computer Engineering

Tripartite Committee:

Professor Dionisios Pnevmatikatos

Professor Apostolos Dollas

Faculty Researcher Dimitrios Theodoropoulos

Abstract

Modern computing systems servers, whether low-power or high-end ones are created around a common design principle: the mainboard and its hardware components form a baseline, a monolithic building block that the rest of the hardware/software builds upon. The proportionality of the resources remains static throughout the machine lifetime, with known ramifications (1) in terms of low system resource utilization, costly upgrade cycles, and degraded energy proportionality.

As a result, a novel approach takes on the challenge of revolutionizing the low-power computing market by breaking server boundaries through materializing the concept of disaggregation. Such an innovative approach needs good planning and a lot of preliminary work in order to reach the desired outcome.

Thus, a full system simulator platform is required, capable of modeling this kind of a data center environment in order to catch a glimpse of what it can deliver in terms of performance. This thesis is all about this idea and its implementation. Furthermore, it provides an introduction to the virtual platforms and their significance in hardware system evaluation before prototyping.

Acknowledgements

I want to give sincere thanks to my parents that supported me all these years. Without their aid, I wouldn't be able to study at all and achieve my personal dreams and ambitions. So I want to dedicate this diploma thesis to both of them.

I also want to thank my supervisor, Professor Dionisios Pnevmatikatos, who gave me the opportunity to be a member of his research team as an undergraduate student. He was the one that instructed to select my thesis subject and helped me ever since.

Furthermore, I have to specifically mention the support that I had from my research coordinator and personal friend Dimitris Theodoropoulos. He was my guide and I thank him for that.

This Thesis is associated with the dRedBox project and in my case, it has been supported by Imperas as regards to the simulation platform.

Finally, I would like to name all those that actively helped me throughout my work. Duncan Graham and Jim Straus, two of the OVP forum administrators. Oscar Gonzales del Dios, Hugo Daniel Mayer and Alejandro Molina-Sanchez form B.S.C. Dimitrios Syrivelis from the University of Thessaly and also Iosif Koidis from the T.U.C. Lastly, a huge shoutout to my friends and studying partners Vasilis Amourianos and Kostas Malavazos. Thank you, guys!

Contents

1 Introduction

There is currently an immense need for high-performing, parallel, distributed and heterogeneous computing across industries of any kind. We see that all sorts of high-end embedded systems are being used for any kind of industrial applications and their variety of uses is seemingly endless. From FPGA-based video analysis, aerial transportation guidance systems and all kinds of everyday safety/security accommodations that we enjoy, to national security applications, parallel supercomputers for advanced industrial manufacturing and cutting-edge medical equipment. **The majority of these industries have been traditionally employing embedded computing, primarily for achieving low power consumption, cost reduction - thanks to fit-for-purpose designs - and for abiding by stringent real-time constraints**.

Embedded systems have always been one of the simplest and easiest solutions. However, as the complexity of the problems that emerged increased, it led to a significant ramp up of the computational power and integration plurality of such designs. One of the main advantages of this evolution is that it has enabled the reduction of component development and maintenance costs through re-purposing of software/hardware used on conventional systems. In terms of modern data centers, the later in conjunction with the explosion of the penetration of mobile into the consumer and enterprise market is constantly pushing for higher performance low power systems-on-a-chip(SoC). High-end embedded computing systems, from small form factor boards to multi-chassis systems, exhibit superior features compared to conventional high-end server designs (i.e. dual or quad socket SMPs), across key performance indicators (KPIs) such as compute density, granularity of resource allocation, power consumption and virtual machine migration times.

These KPIs challenge the way we need to design the next generation computing systems to guarantee environmental, societal and business sustainability. Both technological trends are based on a common design axiom: the mainboard and its hardware components from the baseline, a monolithic building block that the rest of the system software, middleware and application stack build upon. **The dRedBox [KSP+16] design proposes a customizable low-power data center architecture, moving from the paradigm of mainboard-as-a-unit to a flexible, software-defined block-as-a-unit**. This approach allows an optimization of both performance and energy consumption. The baseline disaggregated building blocks to enable the on-demand hardware are, a) microprocessor SoC module, b) high-performance RAM module and c) accelerator (FPGA/SoC) module. Disaggregating components at that level significantly improves efficiency, increases resource utilization and has the potential to revolutionize the way the data centers are being built.

All of this sound very new, very promising and even thrilling. The question is this: is it worth it? The only way to answer that kind of questions is by testing it. There is no better way for us engineers to experiment with a new platform design to find out if it really is as good as it seems. This is why this Thesis is focused on my attempt to building a simulator platform in order to model a part of the original idea of the dRedBox server design. This job was not an easy task and it took a lot of hard work and countless trials and errors to get to a stable and easy to work simulator platform design. But finally, I think that I can say that I developed a sufficient and well-working simulator platform that can help us test the capabilities of the new idea of the dRedBox disaggregated server platform. The entire task of building this simulator platform from scratch can be found on chapter 4 but in the meantime let's say a few more things about the advantages of disaggregation, embedded systems and what lead us to experiment with those new ideas.

1.1 Motivation

Let us talk a little about how we decided to take up the task of trying to emulate the dRedBox platform. It all starts with all those weaknesses and disadvantages about data centers that I mentioned before. There are three inevitable limitations that are side-effects of the way data centers are being built today.

First of all, as we mentioned, the resource proportionality of the entire system follows the proportionality of the basic building block (mainboard), both at production and during upgrades. For instance, if at some point we decide to double the memory capacity in an operational system, we have extra costs because this action carries with it the obligation of adding all the additional but not necessary components throughout the server in order to follow the mainboard design.

In addition, the allocation of resources to processes or virtual machines is upper bounded by the resources available within the boundary of the mainboard, leading to spare resource fragmentation and inefficiencies. A great example to view what I am talking about is to take a look at an application that uses 100% of the CPU and only 40% of the server memory. This automatically means that we can't run any other workload not even memory-bound and thus this 60% of free memory is unusable until the CPU-bound procedure is finished.

Furthermore, If we decide to upgrade any of the server boards this upgrade needs to be carried out to all other boards as well for reasons that we talked earlier. An upgrade can be as simple as changing to a later model of processor.

All of the above create the urgency of a radical solution. Many tries are already in action and some of them I am going to mention in chapter 2 (Related Work). Our approach for the disaggregated dRedBox system is described in the following subsection.

1.2 Disaggregation and the dRedBox approach

All those limitations that I described earlier have been adequately addressed in modern data centers at the peripheral level e.g. by Network Attached Storage (NAS) for persistent storage and PCIe off-board switches for network media. The introduction of the dRedBox aims to deliver a platform with memory disaggregation at the hardware integration level by interfacing the CPU chip memory controller with remote memory modules that can be located on the same or a remote mainboard tray. For example, this kind of memory disaggregation means that the memory resources can be allocated in a significantly more efficient way than traditionally. The connection is going to happen via a novel optical network interconnection technology capable of delivering performance up to 1800million transfers per second, and latencies close to 10ns when using state of the art DDR-to-DIMM interconnection. As for the approach, the hardware-level disaggregation and the software-defined wiring of the resources undertaken by dRedBox will be matched with a novel, required innovation on the system-software side.
In particular:

- Delivering novel hypervisor distributed support to share resources that will allow the disaggregated architecture to bootstrap (15) a full-fledged Type-1 hypervisor and execute commodity virtual machines. That way, by using the Type-1 hypervisor device driver techniques, the disaggregated memory support shall be further used for peripheral disaggregation with proper forwarding of Direct Memory Access (DMA) interrupts.

- Employing deep software-defined control of all resources at the hardware programmability level, including allocation of memory resources to micro-servers and software-defined network control. This hardware-orchestration software, running out off the data-path resources, is to be interfaced via appropriate Application Programming Interfaces (APIs) with higher-level resource provisioning, management and scheduling systems, notably cloud management (e.g. OpenStack) and cluster management systems (e.g. Apache Mesos).

- Finally, reducing the power consumption, which is a key parameter for dReDBox and will be attacked at all layers. When designing the platform, the power consumption of hardware components will play a major role in their selection. The optical network will utilize key technologies that significantly reduce power consumption and improve latency. The hardware platform will provide a per component IPMIv2 interface that will allow the hypervisor and the orchestration tools to extensively control component mode of operation and also completely switch them off when not used. The later is due to the dRedBox central reservation system which will fully control the component interconnect, so in any given point in time it knows which components are in use. In this context, novel online myopic policies will be designed that can take instant decisions to migrate VMs and switch off resources. **The target is to improve the power consumption by 10x compared to state-of-the-art platforms**.

All of the above can give you a good idea of the course that the dRedBox platform idea is headed. It is an innovative approach to the major database resource allocation and power consumption problems. But it needs testing, and here is where my work comes into play. In this Thesis, I was given the task of creating a platform that can simulate this new environment. Based on the original dRedBox platform idea, that will be described in chapter 3, I started working on creating a working MPSoC emulator. By introducing the MPSoC emulation I am basically making the introduction to my thesis main work and I think that this is the best moment to do so. It is worth noting that all this work started by having only the initial dRedBox paper at hand and by having no idea what simulating program to use. The first step was to find a simulator platform/program, capable of simulating the architecture that we had in mind. But first, let me explain why we chose this approach in the first place.

1.2.1 The embedded software evolution

The embedded software contained in modern electronic products used by the majority of the population on a daily basis has evolved dramatically in recent years. This evolution came primarily in three dimensions:

- **Scale**: As the cost of the IC hardware production and products increases, the use of embedded software operating on standardized hardware platforms has become increasingly common. This lead to a tremendous growth of code required for each project and with it the effort to produce it.

- **Complexity**: Multi-core processor architectures are continuously improving, providing the necessary performance and capability to meet modern product requirements. However, as the hardware advances, coding these new processors has become exponentially more complex than previous generations.

- **Quality**: Modern electronic product functionality and quality requirements are more strict and higher than ever, suggesting a zero tolerance for post-production bugs. Another common thing these days is that embedded software has become harder to change as a product moves into production.

In the past, embedded software development and verification were typically performed by executing the codes on a prototype hardware platform for the design. The tests were performed until the product research team satisfied with a result. This approach was time-consuming, highly impractical for all next generation high-end designs and also unreliable in terms of quality and use. Similar issues can be found not only in embedded software development but also in hardware verification procedures. A new solution has to be found. The answer to this is the use of virtual platforms.

1.2.2 Simulating this new design using virtual platforms

Virtual platforms are an alternative to hardware prototypes. Software models of the key components in a processor platform are combined to form an executable system. This kind of models have enough functionality to execute the code correctly and do all the verification procedures and testing. Nevertheless, they retain a certain level of abstraction that provides the performance necessary for rigorous testing.

Typically, all virtual platforms make use of Instruction Accurate (IA) processor models. They also contain a large variety of abstract memory blocks and key peripherals making it possible to create all kinds of different virtual platform designs. The virtual platform needs to be accurate enough so that the software can't tell the difference between running on an actual platform or not, and also production binaries of the embedded software should be able to run unmodified.

The main advantages of using virtual platforms are:

- **Early development potential**: A virtual platform model can be available far quicker than any prototype hardware equivalent, thus creating the potential for much earlier software development and testing than before, saving off time for a product's time-to-market.

- **Visibility and Controllability**: Many hardware prototypes make it difficult to view internal registers and signals and also offer no opportunity to change or control hardware and software execution. Within a well-constructed virtual platform, the tester can view all nodes and apply a range of controls. This is essential for effective verification.

- **Performance and Accessibility**: Prototype hardware platforms often have limited availability in their early production stages. This means a restriction on the amount of testing that can be performed. One of the key aspects of using virtual platforms is that they can be replicated on all available compute platforms, allowing concurrent use by individual members across large teams, or many test platforms operating in parallel. Also with good planning, correct and careful construction and some abstract but correct limitations, they can execute faster than the actual final hardware, allowing for extended testing cycles.

But the question remains: What development environment to use? The construction of a virtual platform can vary greatly according to each new designs needs and can have a significant impact on their performance. In our case, after testing other development platforms as well, Imperas was the choice to go, because it has the right technology and expertise to provide fast and effective development environments based on virtual platforms.

1.2.3 Why Imperas?

After testing other platforms as well e.g. the GEM5, Imperas gave us the package that we were looking for. Imperas develops and markets state-of-the-art virtual platforms and tools to enable the most comprehensive embedded software development, debug and test solutions available today. It combines advanced simulation algorithms, modeling excellence, and a broad range of tools to produce a system that offers:

- **Very fast execution performance**: Imperas leading JIT code-morphing simulation technology allows models of processors, such as the ARM® Cortex^{TM} A-72, to execute at a peak speed of almost 5,000 Million Instructions per Second (MIPS), and booting multi-core Linux in under 4 seconds, on an average desktop PC, as they proclaim.

- **Extensive Library of Accurate Models**: The Imperas OVP model library includes a full range of processors from ARM, Imagination MIPS, PowerPC, OpenCores, Renesas, Synopsys ARC, Altera Nios II, and Xilinx Microblaze. The models are fully featured, e.g. ARM's TrustZone® and Virtualization technology are supported. Example platforms and peripherals are also available that run for example Linux, Android, Nucleus, FreeRTOS, uClinux, eCos. Models operate with SystemC TLM2 and other standards, making it easier for a novice user like me to get familiar with this tool.

- **Advanced Development Tools**: Powerful verification, analysis, and profiling tools plus a multi-core debugger use ToolMorphing^{TM} to merge them into the simulator, to operate with minimal performance degradation or execution alteration. Tools operate from bare metal instructions to CPU and OS-Aware abstract models and can be customized for platform and scenario specific operations.

This environment has been used by Imperas customers to find bugs in previously fully tested production code. Furthermore, in 2008, Imperas founded Open Virtual Platforms (OVP), an industry consortium dedicated to providing open-source models and infrastructure to create virtual platform models as easily as possible making it a very friendly but also a very strong entry and advanced level embedded development tool. OVP provides an easy entry point into the world of virtual platforms with a today's community of over 10,000 members from many leading electronics companies and academic institutions. OVP models operate with other standards, including SystemC and TLM2, while building on the Imperas simulation technology to enable high-performance, high-capability models for any environment. From all of the above, you can see why this was the way we chose this program to work with.

2 Related Work

On the road to developing a better and more efficient data center capable of meeting all the new technological, economical and performance requirements, we are not alone. Other research team and institutes have produced their own alternative ideas/works as a solution to this matter. Furthermore, I was not the only one working on the dRedBox project. There is already a lot of work done in our lab and also others like B.S.C. and there is still a lot of work in progress. In this chapter, I am going to mention some of the more recent and related work that has been done or is currently being developed, regarding the data center evolution and also the disaggregated dRedBox approach on this matter.

2.1 General data center improvement approaches

Starting off I would like to mention some distinct approaches regarding the evolution of data centers today. These are not all the works that are done or currently under development but are a good indication of where projects other than the dRedBox are headed and why do we think that these solutions are not perfect.

- The RAMCloud: As I mentioned before, trending data-intensive workloads most of the time require resources disproportional to the fixed proportionality of the mainboard. In this spirit, the RAMCloud project outlines that strategic use of (permanent) DRAM instead of storage significantly improves the execution time of specific, widely used, cloud applications. To achieve this it aggregates memory modules that are located on different traditional mainboards, using a fast network and appropriate software, wasting this way a significant amount of processor resources and power. Also, integrated CPU/memory/storage mainboard architectures can inherently create resource inefficiencies, both in terms of upgrades and workload-proportional system utilization. This creates the requirement of whole mainboard upgrades and also server bound resource allocation that guarantees a suboptimal investment of capitalization and operational expenses.

- Dynamic CPU, memory, and accelerator scaling: Many research teams have tried to integrate the idea of dynamic CPU, memory, and accelerator scaling at runtime in response to dynamically changing service and application needs. This is suboptimal, due to the fact that it is bounded by what is available in the mainboard tray. If more memory or CPU resources are required, VM migration to another tray is undertaken and that means an incurring overhead and performance degradation.

2.2 Related dRedBox work

Enough with the different approaches. Let us talk about our line of work. A lot of dRedBox related projects are either completed or are currently under development. So at this point I think is the best place to overview the work of other Institutes and Universities and closing with the research that takes place at the Technical University of Crete by prof. D.Pevmatikatos's research team.

2.2.1 Ramulator: A Fast and Extensible DRAM Simulator

[KYM16] Due to that both industry and academia have proposed many different roadmaps for the future of DRAM this project targets the growing need for an extensible DRAM simulator. They focus on a project that can be easily modified to judge the merits of today's DRAM standards as well as those of tomorrow. A fast and cycle-accurate DRAM simulator that is built from the ground up for extensibility. Unlike existing simulators, Ramulator is based on a generalized template for modeling a DRAM system, which is only later infused with the specific details of a DRAM standard. Thanks to such a decoupled and modular design, Ramulator is able to provide out-of-the-box support for a wide array of DRAM standards: DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, as well as some academic proposals (SALP, AL-DRAM, TLDRAM, RowClone, and SARP). Importantly, Ramulator does not sacrifice simulation speed to gain extensibility: According to their evaluations, Ramulator is 2.5× faster than the next fastest simulator. Also, notice that the Ramulator project is released under the permissive BSD license.

2.2.2 Simulating DRAM controllers for future system architecture exploration

[HAK+14] Compute requirements are increasing rapidly in systems ranging from mobile devices to servers and these often massively parallel architectures, put increasing requirements on memory bandwidth and latency. The memory system greatly impacts both system performance and power usage so it is key to capture the complex behavior of the DRAM controller when evaluating CPU and GPU performance. By using full-system simulation, the interactions between the system components is captured. However, traditional DRAM controller models focus on modeling interactions between the controller and the DRAM rather than the interactions with the system. Moreover, the DRAM interactions are modeled on a cycle-by-cycle basis, leading to inflexibility and poor simulation performance. This work presents a high-level memory controller model, specifically designed for full-system exploration of future system architectures. This event-based model is tailored to match a contemporary controller architecture, and captures the most important DRAM timing constraints for current and emerging DRAM interfaces, e.g. DDR3, LPDDR3 and WideIO. This shows how the controller leverages the open-source gem5 simulation framework and a comparison with a state-of-the-art DRAM controller simulator. The results show that their model is 7x faster on average while maintaining the fidelity of the simulation. To highlight the capabilities of their model, they show that it can be used to evaluate a multi-processor memory system.

2.2.3 Disaggregated Memory Architectures for Blade Servers

[Lim10] Current trends in memory capacity and power in servers indicate the need for memory system redesign. Per-server memory demands are increasing due to large memory applications, virtual machine consolidation, and bigger operating system footprints. This large amount of memory required for these applications is leading to memory power being a substantial and growing portion of server power budgets. As these capacity and power trends continue, a new memory architecture is needed that provides increased capacity and maximize resource efficiency. This thesis presents the design of a disaggregated memory architecture for blade servers that provides expanded memory capacity as well as dynamic capacity sharing across multiple servers. Unlike traditional architectures that co-locate compute and memory resources, the proposed design disaggregates a portion of the servers' memory, which is then assembled in separate memory blades optimized for both capacity and power usage. The servers access memory blades through a redesigned memory hierarchy that is extended to include a remote level that augments local memory. Through the shared interconnect of blade enclosures, multiple compute blades can connect to a single memory blade and dynamically share its capacity. This sharing increases resource efficiency by taking advantage of the differing memory utilization patterns of the compute blades. In this thesis, two system architectures are evaluated that provide operating system-transparent access to the memory blade; one uses virtualization and a commodity-based interconnect, and the other uses minor hardware additions and a high speed interconnect. Both are able to offer up to ten times higher performance over memory-constrained environments. Finally, by extending the principles of disaggregation to both compute and memory resources, new server architectures are proposed that provide substantial performance-per-cost benefits for large-scale data centers over traditional servers.

2.2.4 Related work inside the Technical University of Crete

Our institute already has experience in this field of work. This work was the first one about building a **Full System Simulator**, but it was not the first one about testing disaggregated data centers. Two thesis projects have already been presented in our School and I will roughly describe them here. Note that these two were not so different from each other.

- **Andreas Andronikakis. Memory System Evaluation for Disaggregated Cloud Data Centers. Chania 2017.**
  The existing architecture of Cloud Data Centers is characterized by high energy consumption and a great waste of resources. This thesis refers to estimating the memory of cloud data centers with disaggregated (or disintegrated) servers, ie servers whose components and/or resources are in separate sub-assemblies, regarding their physical location. This Disaggregated Architecture System aims to change the traditional way of organizing a Data Center by proposing a more flexible and software-modulated integration around blocks, the Pooled Disaggregated Resources, as opposed to traditional unification around from the mainboard. The purpose of this diploma thesis is to study and develop a unified (modular) memory simulation tool of the above architecture, driven by the execution of an application, on the previously described DiMEM Simulator (Disaggregated Memory System Simulator). He studied the Dynamic Binary Instrumentation, the understanding of Cache levels and their simulation methods, the implementation of the Disaggregated Architecture Memory simulation, and experimentation with various parameters. The results approximately present the overall behavior of a Memory System of a Disaggregated Cloud Data Center.

- **Orion Papadakis. Memory System Evaluation of Disaggregated High-Performance Parallel Systems. Chania 2017.**
  Nowadays research about Disaggregated Architecture Systems aims to change the traditional mainboard-organized Data Center structure by proposing a more flexible and software-controlled one, organized around Pooled Disaggregated Resources. This thesis is part of the DiMEM Simulator, a modular execution-driven Disaggregated Memory Simulation tool study, and implementation. That tool approximately tries to depict the Disaggregated Memory System behavior using HPC workload. The DiMEM Simulator couples the Intel PIN framework with DRAMSim2 Memory Simulator, where that thesis also focuses. The main study object is the DRAMs, the Memory Simulation methods, the Disaggregated Memory Simulation implementation, as well as the parameters experimentation. This work's results show the approximated Disaggregated Memory System Behavior.

3 Platform design: Architecture and Subsystems

In this chapter, we are going to focus on the revolutionizing data center design as envisioned by the dRedBox project. I am going to present to you a detailed architecture and subsystems overview of the server platform that we want to create and also it's key modules, general specifications, and functionality.

DRedBox adopts a vertical architecture(2) to address the disaggregated challenges. At the lowest level, the optical interconnect architecture is used for remote memory communication, with latencies in the tenths of nanoseconds. The interface facilitating remote memory communication will be decoupled from the processor unit, and appropriately integrated into the system interconnect. Forwarding operations will be software controlled to provide the necessary support to other dReDBox components. The dRedBox data center will also adopt the Virtual Machine (VM) as the execution container and several challenges will be addressed at the hypervisor and orchestration tools level, like the implementation of RDMA-like support for peripheral communications using standard DMA programming as well as power consumption control support. In figures 1 and 2, I depict the high-level block diagrams of a target's server(blade) and rack-level architecture.

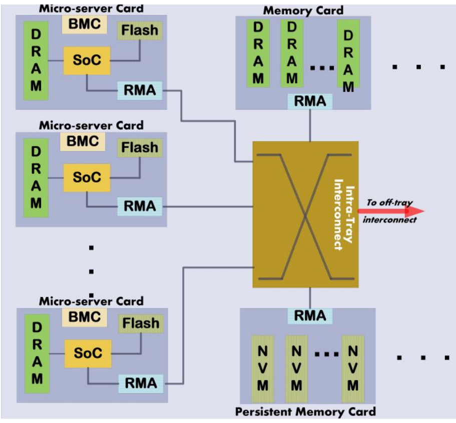


Figure 1: Block Diagram of one type of dReDBox server[tray-mainboard]

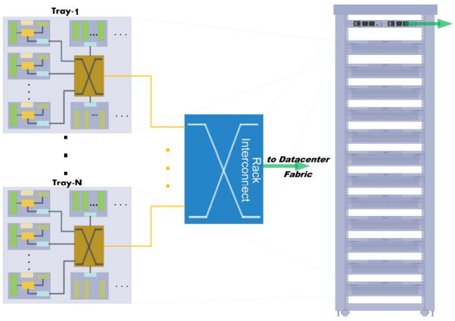


Figure 2: Block Diagram of high-level dReDBox Rack-scale architecture

3.1 Server and Rack Level Architecture

dRedBox aims to deliver at least two types of hardware component blocks and one type of mainboard tray. Both blocks will be interfaced via Remote Memory Adapters (RMA) to the dRedBox mainboard. More specifically, a compute block, as featured in Figure 1, will have a high-performance SoC integrated with a local RMA, local memory, flash memory and an Ethernet-based Board Management Controller (BMC). Figure 1 also depicts 2 memory blocks, featuring DRAM and NVM module. They are both interfaced to the rest of the mainboard chassis via their local RMA and can also serve as an accelerator module. These memory blocks will be interfaced with a much higher bandwidth interface (40G/100G) in order to support the large scale network monitoring used for the dRedBox accelerator framework.

The generic dRedBox is aimed to feature a series of memory slots, appropriately interconnected in order to provide:

1. power,
2. a serial electrical interface to the electro-optical crossbar chip,
3. also, a PCIe interface, so that selected DIMM(dual in-line memory module) slots provide connectivity to a PCIe switch, and
4. a per component IPMIv2 (3) interface that enables intelligent IPMIv2-based management from the orchestration tools.

Two or more mainboards are aimed to be featured inside a tray and get interconnected with each other via the optical network. In Figure 2, I depict a set of dRedBox mainboards interconnected to a rack-level interconnect, forming the desired disaggregated Rack-Scale architecture.

The dRedBox also aims at delivering a fully-functional PCB prototype of the described platforms in several copies to be used for the integration of the rest of the subsystems and use case demonstrations. More specifically, the mainboard will be a small factor prototype appropriate for a data-center-in-a-box type configuration.

3.2 Electro-optical switched interconnect

A disaggregated rack-scale data center with high density, poses a series of challenges to the system interconnect. Scaling of the network along with controlling end-to-end latency and power consumption requires the addition of switches, links, and hierarchy levels to support the resource pooling. Each switch that is added to the network must be connected to both a higher and lower hierarchy level and has upfront implications to the switch radix and network topology selected. One of the challenges is to decide the number of interfaces to be used for the various layers of the interconnection hierarchy(in-tray, off-tray, off-rack), as driven by workload requirements specification and technology density/cost. In any configuration, the design goal is to minimize the number of layers used since more layers imply more switches and thus higher cost as well as power consumption and latency.

Fully non-blocking networks (logical full mesh topology) typically employ a Clos (17) topology [Clo53], [ZGY10] . In our case, the Clos network is the most efficient topology and it is possible to implement when meeting these two conditions:

1. The node has enough interfaces to connect to all switches and
2. The switch has enough interfaces to connect to all nodes and switches symmetrically.

The limiting factor here, is the number of available ports on the switching device and since this number is limited, Clos network used are more than a single stage. 3- or 5-stages are common in large data centers.

Moving on, increasing the chip I/O bandwidth is not a simple task since the packaging technology is quickly reaching its limit. With more and more SerDes(4) added to the package, the trace breakout into the PCB becomes a limiting factor since - at high line rate - traces cannot be brought close to each other without special design. The complexity associated with trace breakout implies that the SerDescan be placed only on the perimeter of the package. Their number is thus constrained by the package size and the number of BGA pins that may be used on the perimeter. Currently, the state of the art is roughly 150 SerDes/package and this number is not expected to exceed 250 with very complicated and power-hungry large package design.

As a solution to this problem, the dRedBox introduces the addition of optical I/O to the switching device. More information can be found in references described in [Has14] and [HBG+14]. Integrating a large, two-dimensional optical interconnect directly on an ASIC has the benefit that low power SerDes can be used, as they need to drive only a fiber link compared to driving an approximate 30” metal trace on a board. The power saving is by a factor of 10x and is shown for the two cases in Figure 3.

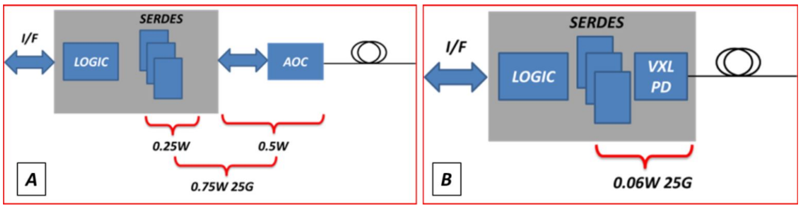


Figure 3: Power consumption of an electrical I/O (A) and an optical I/O (B); the logic block is assumed to be the same in both cases

Parallel optical interconnect directly to the SoC ASIC allows an increase in both the switch radix and the bandwidth of the device. Since data flow to and from the ASIC is routed via the optical interface, the number of SerDes used for I/O is the number of optical lanes utilized. The logic to be implemented is L2 forwarding with full crossbar functionality. A fiber matrix is assembled above the optoelectronic matrices to enable card-to-card and tray-to-tray connectivity. It is worth noting that some of the I/O can be still routed through conventional electrical SERDES, thus, the overall I/O bandwidth of the device is the combination of both optical and electrical I/O.

And now some numbers. With the optical interconnect integrated directly into the SoC ASIC, we expect a network size of 1.5Pb/s to be attainable in a single-stage fully non-blocking topology with about textbf10% of the number of devices needed with other technologies. Thus the power saving and latency gains from this technology and design are evident. Furthermore, scaling of the data center network is enabled using this integrated electro-optical switch. Lastly, since the chip radix is in the 15Tb/s range, the size of the network that may be obtained in a single-stage Clos topology is larger than any network using conventional SerDes technology.

3.3 Memory Disaggregation

Memory disaggregation will require appropriate support starting from the lowest level which is the memory interconnect architecture. In Non-Uniform Memory Access (NUMA) architectures today, the Dual In-line Memory Modules(DIMM) are typically the slowest but also the largest elements in the high-performance memory access loop. It is worth mentioning that **the cache architecture is interleaved between processor and memory**

chips to improve performance. In dRedBox, we aim to disaggregate memory by placing modules on a dedicated memory card and interface them via the system interconnect to the RMA of each micro-server. One of the biggest challenges, is to develop the appropriate memory interface and logic for the transmission over the optical network. dReDBox aims to integrate existing SoC architectures and aims to design and develop a "Virtual Memory DIMM" component that can be directly interfaced to a commodity DDR controller. This component will be configured via memory-mapped I/O using a special address range and it will be capable of moving memory data to/from the optical network. Moreover, a local DIMM module will be appropriately mapped and used for system software bootstrapping support. Virtual DIMM software-defined configuration will associate memory address ranges with optical network forwarding information so that remote DIMMs can be reached. Also, on the memory card pool, a different version of the virtual DIMM will be directly interfaced and will be able to access physical modules on the memory card.

Another important challenge that I have to mention is that the dRedBox hardware memory interface design aims to address the distribution of DMA transfer interrupts. In the dReDBox platform, the DMA chipsets that are integrated into microserver SoCs will be used. Each time a DMA transfer is programmed, a list of processor interrupt recipients will be configured at the remote Virtual DIMM. When the transfer is complete, interrupts are to be delivered to the remote processor(s) accordingly. The described inter-tray interrupt mechanism will provide valuable support to the virtualization software and aims to enable integration of peripherals which will be driven by dedicated microservers.

3.4 Operating system support for disaggregation

For applications, dRedBox aims to provide a commodity virtual machine execution, without compromising performance. But, in order to run the currently available virtual machines unmodified on the disaggregated platform, there are some important challenges that should be addressed at the OS and hypervisor layer.

The dRedBox hypervisor will be based on KVM(5), a kernel module that enables a standard Linux Operating System to execute one or more virtual machines. Evidently, an instance of KVM will need to run on each microserver platform taking advantage of the locally available memory. Unlike the current server architecture case, the host system on each dReDBox microserver may not be able to detect all available platform components using the BIOS. In fact, the BIOS on each microserver may only provide locally attached component information. **Therefore, it is crucial that during bootstrap, the host system should communicate with the orchestration tools to get information about all available memory and peripherals of a dReDBox configuration**.

Immediately prior to a virtual machine deployment, the hypervisor will interact with the dReDBox orchestration tools, asking to reserve resources and set up the appropriate network state for reachability among the pooled resources involved. We will also explore advanced OS and virtualization techniques for dynamic disaggregated memory allocation at OS level. Beyond memory modules, other peripherals are intended to be disaggregated and physically attached to dedicated dReDBox microservers. Such dedicated machines will be connected to peripherals through a given PCIe switch. According to this approach, peripherals are aimed to be made available to virtual machines through direct assignment, or remote para-virtualization (6). The former is aimed to be available for VMs running on the same server where peripherals are interconnected and will leverage the current support from the KVM Hypervisor (16).

3.5 Resource allocation

The dRedBox disaggregated platform needs orchestration support that is currently unavailable by state-of-the-art data center resource management tools. The new challenges that this architecture introduces and are not yet addressed by any other platform are:

- a switching network that requires forwarding information that is capable of interconnecting any combinations of the platform's components,
- the need for distribution of the globally accessible physical memory address space located throughout the data center, and
- a novel per component IPMIv2 (3) control.

In other words, the dRedBox project approach aims to develop a new orchestration tool layer that aims to:

- implement dynamic platform synthesis by analyzing the physical hardware resources and software performance requirement in order to allocate components and set appropriate forwarding information to interconnect them,
- maintain a consistent distribution of physical memory address space and accordingly provide support to running hypervisors for memory segmentation and ballooning (7), and
- taking advantage of the component level usage information and IPMIv2 control to significantly decrease the required power budget.

This resource allocation and orchestration layer is to be integrated via a standardized API with resource management tools like OpenStack. That way, virtual machine deployment steps will be extended to include a resource scheduling and a platform synthesis step, which aims to reserve the required hardware and configure the platform interconnect in a power-budget conscious manner. The following Figure 5 depicts the target dReDBox platform, featuring orchestration tools and relevant component interactions.

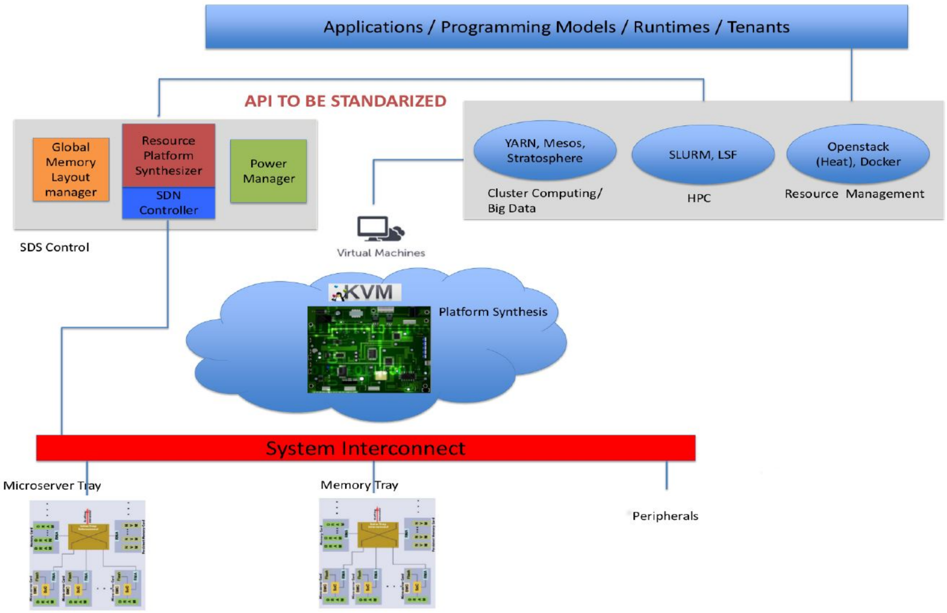


Figure 4: Resource Allocation and Orchestration in dReDBox

3.6 And now what?

We can see now that the dReDBox project and architecture vision is to disaggregate computing resources at the lowest possible level, which would result in a datacenter box that is fully configurable and can adapt to any targeted applications profile. For example, if applications are highly parallel and CPU intensive, more cores can be accommodated rather than memory, whereas when applications are I/O intensive, cores can be traded off for memory, disks and network media. As we said before dReDBox will also deliver a vertically integrated working prototype, featuring integration of hardware, system software derivatives and pilot applications porting on the embodiment of the dReDBox architecture.

So as the working prototypes are under construction it is my job to do the preliminary work and that is to find a platform capable of emulating and hosting at least a microserver-card of the envisioned dRedBox database. In the next chapter, I will provide all the steps that I followed to accomplish this task as well as all the drawbacks, stepbacks and obstacles that I have encountered along the way. But, as I mentioned in the introduction, the fastest way to test hardware is through virtual platform testing. So this is where my role in the development of the dRedBox project comes into play and I hope that it will come in handy after its completion.

4 Emulation platform implementation

As I said in the introduction, the best way to test a new embedded hardware architecture before having an initial prototype platform is by using virtual platforms and for that, we decided that the Imperas platform was the way to go. The reasons for this decision can be found in the appropriate sections of the Introduction 1.2.2 and 1.2.3. However, the work that followed this decision will be described thoroughly in the following sections. But first, let us talk about the idea and how we envisioned this virtual platform emulation design.

My work began as the quest of finding the best emulation platform to better suit our needs. This came up due to a number of reasons apart from the benefits provided by Imperas. I mentioned earlier that the use of virtual platforms is the way to go as regards of system design and evaluation. We needed to come up with an idea of how to implement our vision into reality and for that we needed a good platform to work on. After quite some research and thinking ahead, we decided what we needed to 'create' in order to make the first ever design of our platform.

For starters, we needed a baseline. So, we thought that the best way to start was to create, with the minimum compromises, a platform that consists of a Microserver Card (SoC,local Memory Module, Ethernet device, Virtio Block device e.t.c.), an Interconnect(possibly not optical but with configurable timings) and a version of the Universal Remote Memory DRAM module. Those modules are roughly described in Chapter 3 and more definitions can be found in the dRedBox paper and the references. This initial platform meant to be as close as it could to the original architecture design of the dRedBox Server when it is using only one microserver card. The platform had to be capable of running basic bare metal tests provided by the Imperas platforms to see if it actually runs. It also had to be extremely configurable especially in terms of testing all sorts of interconnection delays, memory sizes(local and remote) and also different versions of processors-SoCs.

The real challenge for us is that at some point we should be able to run advanced data center benchmarks and for that, we settled that we had to make this platform capable of loading an OS. This idea also came as a result of the vision of the dRedBox where apparently every microserver-card will eventually have an instance of a VM running on it. Thus, the idea of the "Simulatorception", as we called it, came to life. It is basically the idea of simulating a working OS on the already simulated platform and run benchmarks on it. It might sound crazy but from our point of view, this was the best and also the most interesting way to address this matter.

In the following sections, I am going to describe everything that went through in the process of creating this platform and provide all the information regarding the decisions I made as well as some of the drawbacks, breakthroughs, successes, and compromises along the way. It is worth noting, that I am going to also add some code samples and many figures in order to help you better dive into the world of the virtual platform emulation and especially this Imperas simulator platform. And for that matter, I think that you will be able to see clearly what my work was all about and that I made a good start. And this is what my thesis was meant for in the first place!

4.1 The Imperas platform

Now that we have selected the simulator that we are going to use, it is time to decide a version. Imperas offers a variety of simulator platforms for everyone's needs. We were given the opportunity by Imperas to choose which version we saw fit and provided us with the appropriate academic license as part of their politics about research University teams. In our case, we went all out and acquired the Imperas Multicore Software Development Kit(M*SDK) (8) which is the top version of the Imperas simulator. The licensing procedure was as simple as sending a request email and from there, my supervisor filled in the necessary verification documents and voila. The procedure of setting up the license server in order to use the simulator will be addressed later, but for now, let us talk about the M*SDK.

Although we might not need all the capabilities of the M*SDK we thought that it was better to have the full version than to find later on that we lack some needed features. We did that to play safe because at the time I had little knowledge of the Imperas platform and also we didn't know how far we needed to escalate the simulated dRedBox platform design and testing.

The Imperas Multicore Software Development Kit (M*SDK) is a complete embedded software development environment that operates using virtual platforms and is specifically designed to handle complex multicore related issues. M*SDK contains all the capabilities of the M*DEV (9) product, together with a comprehensive verification, analysis, and profiling (VAP) (8) toolset, plus an advanced 3-dimensional (temporal, spatial and abstraction) debug solution, named 3Debug^{TM} (8), for heterogeneous multicore processor, peripheral, and embedded software debug. You can view the general idea of the M*SDK in Figure 5 and all the information about the VAP tools and the 3Debug can be found in the appropriate references, although, I am not going to say anything more about them here. That is because, besides the fact that we had the M*SDK platform, I only made use of the M*DEV features as part of my work. I made only a little use of them because I personally wanted to see what they have to offer. However, I haven't dug deep enough to fully understand their capabilities so I will not address them at all in this thesis review.

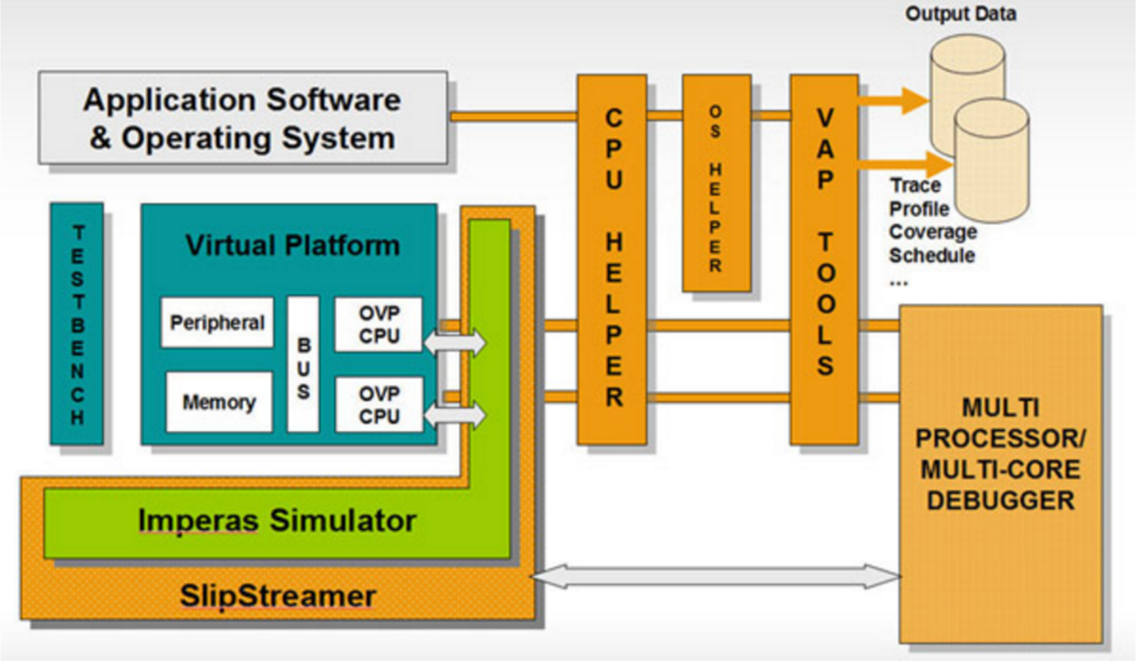


Figure 5: The M*SDK platform concept. More information can be found in reference (8)

4.1.1 Imperas Developer Platforms

M*SDK is a superset of the M*DEV platform, which is one of the three developer products of Imperas(C*DEV, S*DEV, M*DEV)(9) and has all its features. Specific details for it can be found in (9) but I am also going to say a few things on that matter, here.

The Imperas Developer products consist of tools, models and infrastructure components critical for the high quality, rapid development, and verification of embedded software, by utilizing virtual platforms. **They provide the necessary capabilities to develop platforms, including software simulation capability to execute embedded code on the platforms**. In Figure 6 you can view the concept of the M*DEV platform. Terms like iGen and OVP will be explained later in detail.

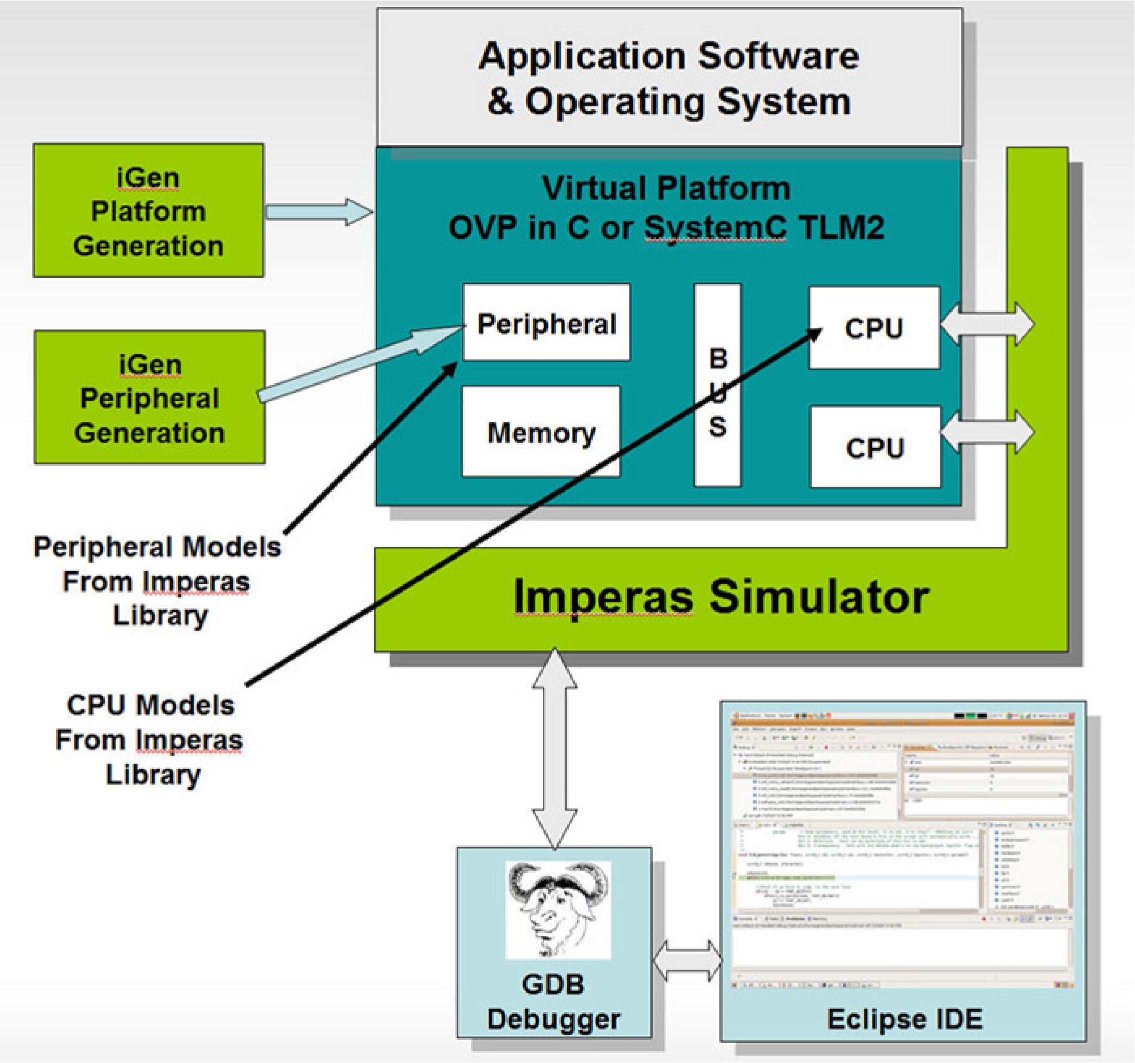

Application Software & Operating System
iGen Platform Generation
Virtual Platform OVP in C or SystemC TLM2
iGen Peripheral Generation
Peripheral
Memory
BUS
CPU
CPU
Peripheral Models From Imperas Library
Imperas Simulator
CPU Models From Imperas Library
GDB Debugger
Eclipse IDE

Figure 6: The M*DEV platform concept. More information can be found in reference (9)

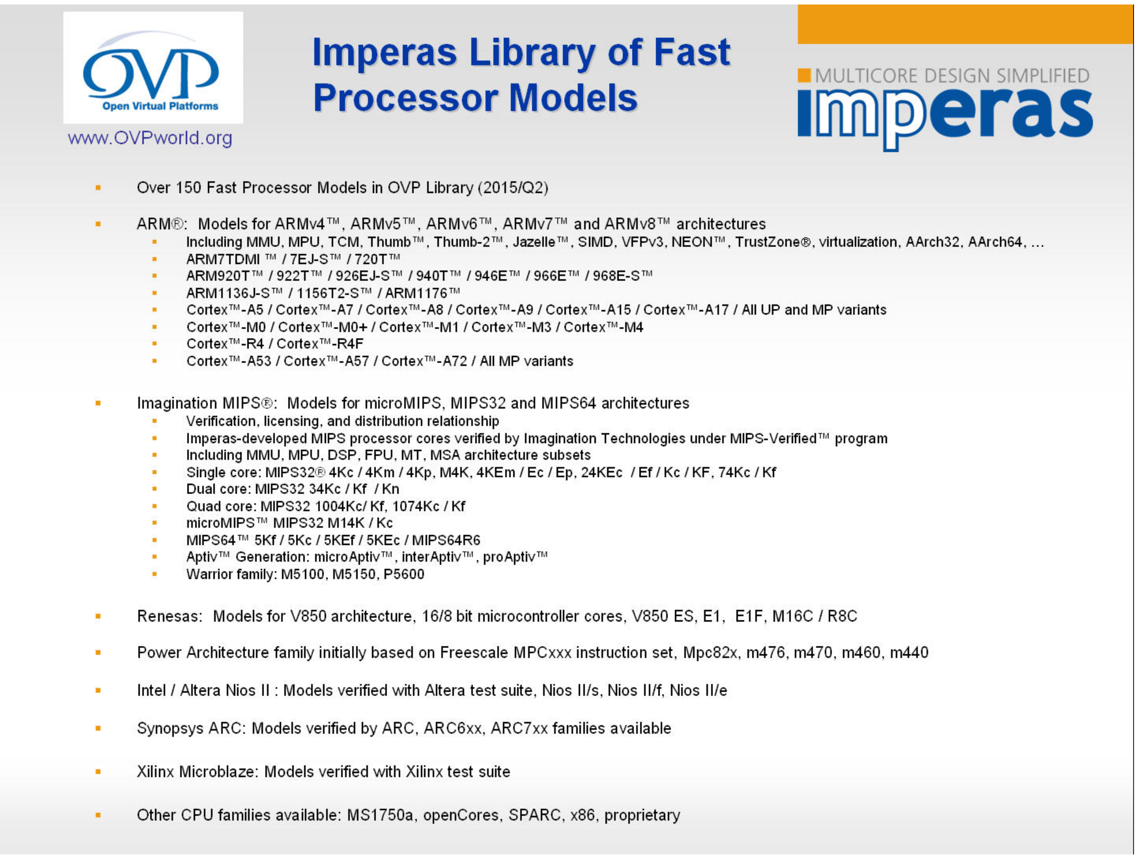
To enable the rapid creation of accurate and efficient virtual platforms, all the Developer products contain these 3 key features:

1. The Imperas model library which is composed of a wide range of processors, platforms, and peripheral models.

2. The iGen Model Generator, which automates the creation of a code framework for new models, and simplifies the laborious and error-prone initial phase of model generation. These models are built on the platform development infrastructure of Open Virtual (OVP).

3. The targeted simulation solution for the execution of embedded code, dependent on the processor capability requirements. The simulator can operate with GDB and the Eclipse IDE, as well as the Imperas Multicore Software Design Kit but we are not going to deal with them here.

Imperas has developed an extensive model library of processors, peripherals and reference platforms for use in Virtual Platforms. The model library contains more than 160 models of CPU devices and most of them can be viewed in the following Figure 7. I used only processors from the Cortex-A ARMv8 family and more specifically A53, A57 and A72. Almost one hundred common peripheral models may also be downloaded from the library. In addition to the processor and peripheral models, many reference platforms are also included inside this library and they represent well-known industry examples, such as the MIPS Malta. All the models are written in C and all use the OVP APIs thus, enabling execution on OVP compliant simulators such as the Imperas simulator. All models may be downloaded directly from the OVPworld.org website. More information about the Imperas models can be found in (9) and (10).


OVP
Open Virtual Platforms
www.OVPworld.org
MULTICORE DESIGN SIMPLIFIED
imperas

Imperas Library of Fast Processor Models

- Over 150 Fast Processor Models in OVP Library (2015/Q2)
- ARM®: Models for ARMv4™, ARMv5™, ARMv6™, ARMv7™ and ARMv8™ architectures
  - Including MMU, MPU, TCM, Thumb™, Thumb-2™, Jazelle™, SIMD, VFPv3, NEON™, TrustZone®, virtualization, AArch32, AArch64, …
  - ARM7TDMI ™ / 7EJ-S™ / 720T™
  - ARM920T™ / 922T™ / 926EJ-S™ / 940T™ / 946E™ / 966E™ / 968E-S™
  - ARM1136J-S™ / 1156T2-S™ / ARM1176™
  - Cortex™-A5 / Cortex™-A7 / Cortex™-A8 / Cortex™-A9 / Cortex™-A15 / Cortex™-A17 / All UP and MP variants
  - Cortex™-M0 / Cortex™-M0+ / Cortex™-M1 / Cortex™-M3 / Cortex™-M4
  - Cortex™-R4 / Cortex™-R4F
  - Cortex™-A53 / Cortex™-A57 / Cortex™-A72 / All MP variants
- Imagination MIPS®: Models for microMIPS, MIPS32 and MIPS64 architectures
  - Verification, licensing, and distribution relationship
  - Imperas-developed MIPS processor cores verified by Imagination Technologies under MIPS-Verified™ program
  - Including MMU, MPU, DSP, FPU, MT, MSA architecture subsets
  - Single core: MIPS32® 4Kc / 4Km / 4Kp, M4K, 4KEm / Ec / Ep, 24KEc / Ef / Kc / KF, 74Kc / Kf
  - Dual core: MIPS32 34Kc / Kf / Kn
  - Quad core: MIPS32 1004Kc/ Kf, 1074Kc / Kf
  - microMIPS™ MIPS32 M14K / Kc
  - MIPS64™ 5Kf / 5Kc / 5KEf / 5KEc / MIPS64R6
  - Aptiv™ Generation: microAptiv™, interAptiv™, proAptiv™
  - Warrior family: M5100, M5150, P5600
- Renesas: Models for V850 architecture, 16/8 bit microcontroller cores, V850 ES, E1, E1F, M16C / R8C
- Power Architecture family initially based on Freescale MPCxxx instruction set, Mpc82x, m476, m470, m460, m440
- Intel / Altera Nios II : Models verified with Altera test suite, Nios II/s, Nios II/f, Nios II/e
- Synopsys ARC: Models verified by ARC, ARC6xx, ARC7xx families available
- Xilinx Microblaze: Models verified with Xilinx test suite
- Other CPU families available: MS1750a, openCores, SPARC, x86, proprietary

Figure 7: Imperas model library

iGen Model Framework Generator is a tool to aid in model creation. It takes as input a simple TCL specification that includes device internals such as registers and memories, port information, component descriptors, and other elements. iGen builds three sets of model items, C code model files, user editable templates, and XML descriptions. These include model frameworks with registers, function calls, memory map, and other items, including matching XML information. It ensures that all component parts of the model are well-structured using best practices, and are consistent throughout the files, thus eliminating a common source of errors. In general, iGen presents to the developer a set of function calls and model elements that simply need to be filled in with required behavior, thereby reducing the setup overhead of a new model significantly. It also creates code in the model to provide very efficient runtime tracing and diagnostics. An idea of how it works is shown in the following Figure.

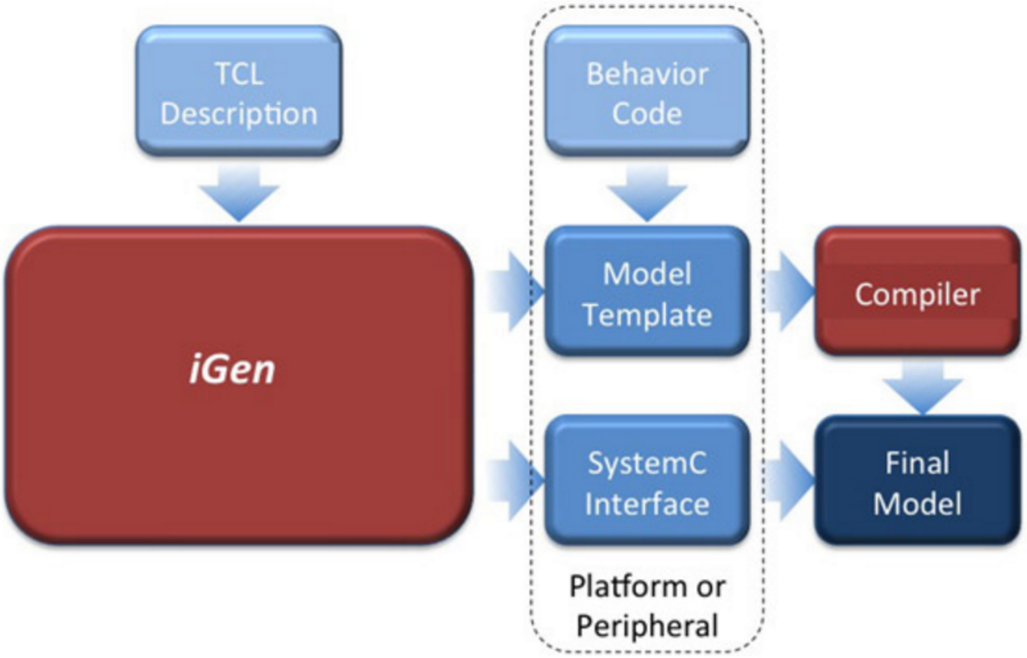


Figure 8: iGEN Model Framework Generator

4.1.2 The OVP

It is now time to talk to you about the OVP (10) that I am referring to in all those previous sections. **But what is OVP? It is a simulation to develop software on!**

As I mentioned earlier, Imperas developed top of the line virtual platforms and modeling technology to enable simulating embedded systems running real application code. These simulations run at speeds of 100s to 1000s of MIPS on typical desktop PCs, are **completely Instruction Accurate** and model the whole system. Imperas decided to open up this technology and OVP is the vehicle to make it public. With OVP you can put together a simulation model of a platform, compile it to an executable, and run bare metal applications, test programs or connect it to your debugger to provide a very efficient fast embedded software development environment.

As I said Imperas has the OVP as a vehicle to open the open virtual platforms technology to the public, and so it has to keep it up to date and also state-of-the-art. A big improvement came in March 2016 when Imperas added six more advanced capabilities to the technologies available to OVP users. I think it is crucial to add them to this thesis as they outline the work that will be described later and also it will help you understand how and why we use OVP.

- The first addition is the **Open Platforms (OP) API** that enables virtual platforms to be defined and controlled with a very powerful C API that allows the modeling of hierarchical systems. It also facilitates the creation of test benches and harnesses to precisely control these hierarchical, often many processor systems.

- The second, that we talked about in the previous section, is the introduction of **iGen**.

- Third and Fourth are the **eGui** and **iGui** which are graphical debug interfaces to allow GUI-based source code debug of virtual platforms and embedded software. iGui is a very easy to use, very efficient GUI that encapsulates GDB and provides easy access to embedded software debug. eGui is an advanced debugger based on the very popular Eclipse IDE.

- Last but not least we have the two things that give this platform its great advantage. **All OVP (and of course Imperas platform) downloads now include the Instruction Set Simulator (ISS) and Harness. ISS is a program that allows you to run compiled binary embedded software 'elf' files on specific embedded processor variants without the need to develop a virtual platform**. ISS comes with the choice of over 170 embedded processor cores. **The Harness program enables simulation of virtual platforms without the need to create a test harness or test bench**. The Harness is the one that I capitalized on the most and has been a great asset in my working procedure. I used it mostly on all my bare metal tests and builds and also on the final platform where I needed it to set up the OS booting on the simulated platform.

It is worth noting that the OVP initiative is very successful with many users of the technology from hobbyists, universities, commercial companies, and advanced research facilities. One of the best features of Imperas products and of course OVP is that the modeling standards allow for functionality to be included quickly while maintaining model performance and ease of use, reducing a typical multi-month model development project down to just a few weeks. I said before that OVP models can be used in platforms written in C or in platforms written in SystemC TLM2.0. On top of that, I will extend what I said earlier by including that the processor and peripheral models are delivered with a C bus transactional interface that is used in C based OVP platforms, and this interface is used to provide a high-speed TLM 2.0 interface layer allowing the model to be used in a SystemC TLM 2.0 environment with the fastest possible simulation performance.

As always, more information can be found in (10) and also on all Imperas and OVP websites and forums. The specific reference to learning more about the OVP technology is (11). However, for now, it is time to be a little more specific as to how the Imperas-OVP simulator actually works and in the following sections I am going to describe how I "fine-tuned" it to work for me.

4.1.3 The Imperas Simulation System

At the core of all Imperas solutions is a fully functional simulation system that operates at the highest performance of any virtual platform simulator that is available today. The simulator is designed to allow embedded software to be executed on a modeled virtual platform for verification, analysis, debug, profiling and all sorts of other testing purposes. Its purpose within the Developer products is to execute the various models, interface the models to the appropriate selected tools, run software on the processor models, and enable the tasks of OS and driver bring-up, or initial embedded software development.

Imperas achieved something completely innovative in terms of the technology and performance combination. **Its models and simulator enable high-performance simulation of systems running embedded software using the production unmodified binaries of the embedded software. The embedded software running cannot tell that it is not running on physical hardware but it is running on a simulated virtual platform model of the actual physical hardware**.

I mentioned earlier that all models of the processor and other components are created in C using the OVP APIs. A full range of single, homogeneous multicore and heterogeneous multicore processors are supported by the simulation engine, based on the respective Developer product utilized. The simulator also supports any style of bus topology and can be used in C or C++/SystemC environments. All models in the Imperas-OVP model library can be in used in C or SystemC platforms, and the models are provided with native SystemC TLM2 interfaces.

The simulator utilizes **Just-In-Time code morphing** to enable particularly high-performance execution speed on standard x86 desktop computers and it also worked extremely well on my 32bit Ubuntu16.04 machine. I will explain the 32bit OS choice in one of the following sections along with all other OS compatibilities. **By generating execution code on the fly, and combining this with other Imperas proprietary optimization schemes, the simulation performance that is achieved allows embedded code to be executed, often, faster than real time**. Very large numbers of tests may be executed on embedded code within reasonable timescales, allowing product quality to be increased. Furthermore, an extension of this mechanism, the Imperas ToolMorphing (12) technology, allows for a highly streamlined integration of the Imperas software development kit, minimizing the traditional performance overhead and enabling tool usage without impact on software execution accuracy.

In short, **Imperas' breakthrough ToolMorphing technology extends the above mechanism to generate tool and model code together**. ToolMorphing allows users to easily build the models of their hardware platforms and to integrate existing, industrially proven processor models that include tool and simulation capabilities, adding advanced, unique software development features operating at a high-performance level. The entire tool suite is in use at a number of leading customers on real systems, and the embedded software industry has just started to realize the potential benefits and capabilities of virtual platforms! As always, for more extensive information, you can visit the Imperas and OVP websites and specifically reference (12).

4.2 The Implementation

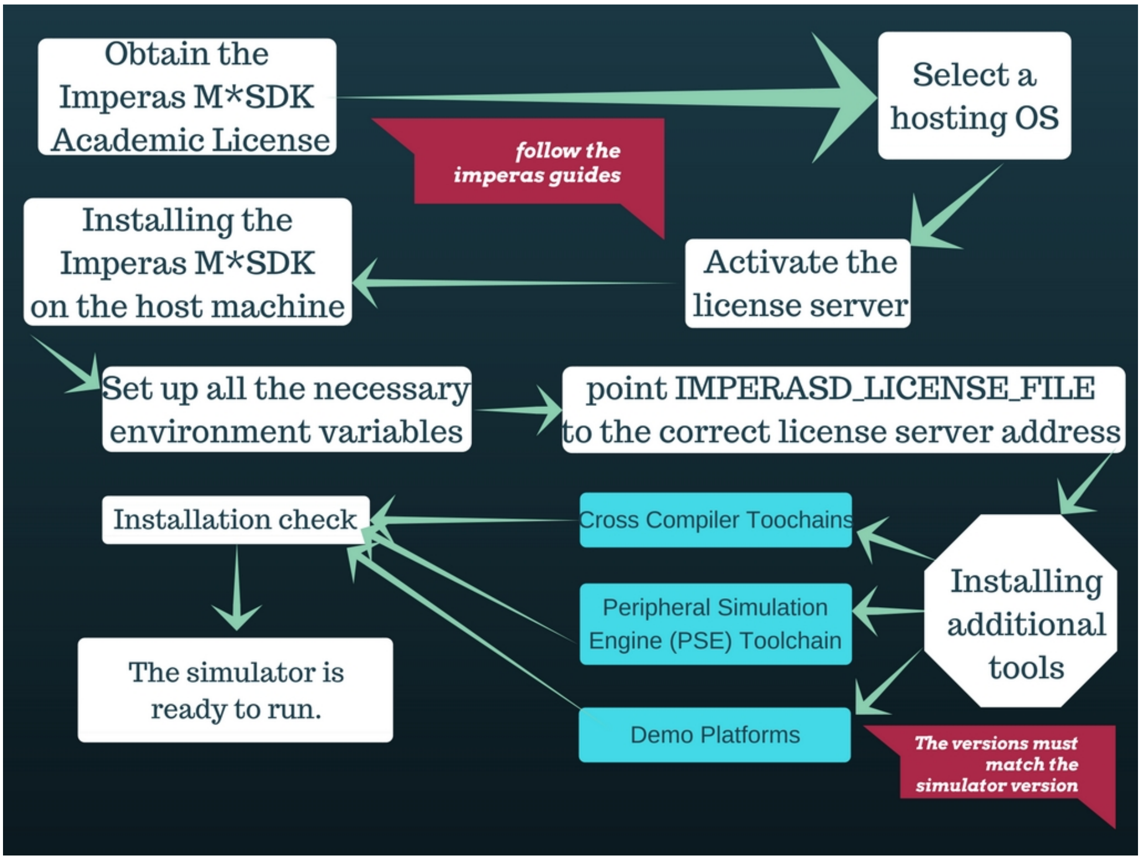
A complete introduction has been made about the envisioned dRedBox platform design, the related works and the Imperas-OVP virtual platforms. This introduction provides a better understanding of the steps followed to complete this thesis work. The idea, the motivation, the vision, the breakthroughs and all the new capabilities and possibilities that this project has to offer have all been addressed in the previous chapters and sections. This section is all about the implementation of the idea.

This work was about modeling at least a blade(one microserver card with local and remote memory) of the dRedBox server, combined with all the the necessary components so that it would be capable of both bare metal application testing and also loading a fully functional Operating System. Then, the idea was that we would be able to run all the desired benchmarks on the simulated platform using the simulated OS interface. But, before that, a series of preliminary steps had to be completed in order for this idea to become a reality. These steps will be briefly discussed here. More information can be found in Appendix B appB and in the Imperas installation guides.

This work begun by getting access to an Imperas M*SDK academic license, provided by the Imperas administrators. After acquiring this license,a license server has to be set up in order to host it. Note that running the Imperas license server within a virtual machine is not permitted. For this work, a floating license was used, described in the Imperas installation guide. On the host side it is crucial that the **IMPERASD_LICENSE_FILE** environment variable should be set to point to the license server.

With the license server up and running, the installation process can be initiated by executing the Imperas M*SDK self-extracting executable file provided by the Imperas model library. Imperas products are developed to work on x86 hardware, 32-bit or 64-bit. Also, GCC compilers versions on all available OS versions are used by Imperas for generation of host code. After successfully installing Imperas M*SDK, the Linux Environment variables have to be set accordingly, as instructed by the installation guides appB.

Before we are able to run anything on the simulator, some basic features need to be installed.**It is crucial that the installed tools are for the exact version of the installed simulator that we are using**. First, the platforms require the Cross Compiler Toolchains. These allow a user to quickly start generating application binary code that can be executed on OVP processor models. For each processor that is being used in a platform, the respective cross compiler toolchain is required.Next stop, is installing the Peripheral Simulation Engine (PSE) Toolchain. This is the simulation engine that is used to provide peripheral models in a virtual platform simulation. The installation of the appropriate Cross Compiler Toolchains and Peripheral Simulation Engine (PSE) Toolchain comes first, followed by other packages, for testing and evaluation, that can be used as a guide or as a baseline for many new designs. To verify the installation after an OVPsim or an Imperas install one of the Demos that are downloaded in both installations can be executed.



Obtain the Imperas M*SDK Academic License
Select a hosting OS
follow the imperas guides
Installing the Imperas M*SDK on the host machine
Activate the license server
Set up all the necessary environment variables
point IMPERASD_LICENSE_FILE to the correct license server address
Installation check
Cross Compiler Toochains
Peripheral Simulation Engine (PSE) Toolchain
Demo Platforms
Installing additional tools
The simulator is ready to run.
The versions must match the simulator version

Figure 9: Preliminary Steps Flowchart

4.2.1 Semihosting and the Code Morphing Simulator

In the case of Imperas-OVP simulations, the term **Semihosting** refers to specific behavior that is provided by the Host system rather than the simulation platform or an operating system running on the simulation platform. This is very important because it takes all the unnecessary workload off the simulated platform and executes it on the host machine. **This extra work is moved to the host because it is irrelevant to the simulation and does not affect the simulation results**. So, it removes the complexity of the software and the platform that is not required if we simply wanted to cross-compile and run a bare metal C application on a specific processor variant. This is just the general idea. For mode detailed information visit the Imperas or OVP websites.

A **Code Morphing Simulator** operates in two phases to provide the behavior of a target processor executing a sequence of instructions.

1. Morphing cross compiled target instructions (for example ARM Cortex-A9, MIPS 1074Kc, V850) to a sequence of native host instructions.

2. Executing the sequence of native host instructions to provide the behavior expected of the target.

In 1 we are executing in 'morph time' using the translations specified by the OVP processor model. The code morpher will continue reading instructions and morphing code until it has constructed an entire code block. Once a code block is created, the block is executed. In 2 (run time), for each target instruction behavior, a) the instruction is reread from memory - but this time as a real transaction that will have an effect on the system that the fetch would normally have and b) the native code is executed to create the behavior.

Another part of the Code Morphing Simulator is the SystemC Interface Transaction Types. In SystemC TLM2 interface code is generated for each OVP fast processor model. The component VLNV source library is found at IMPERAS_HOME/ImperasLib/source, and all the relevant details can be found in both the Imperas-OVP guides and also on their websites. Just note that it is crucial for the SystemC virtual platform to provide both debug and real interface connections.

4.2.2 The ARMv8 Foundation Model

In this section, we are going to talk about the use of the ARMv8 Foundation Model (13) as described in the ARM document DUI 0677C. We are going to discuss why I chose this platform as a baseline for my design and what modifications I made along the way. Also, I will mention how I found about this design and how its simplicity and features made me use it.

First of all, I needed a general idea in order to implement a portion of the dRedBox server design on my simulator. As I said before, my main focus was to implement at least one microserver blade, meaning a SoC with some local modules(microserver card), at least one module of local memory, external memory and a configurable interconnect. My approach was made in line with the need that this system must resemble, as close as possible, a dRedBox server blade. Furthermore, it must have all the features so that it can load a fully functional Operating System like for example a Linux OS and also have the ability to interconnect with the host in order to get internet access.

As I searched through the OVP library, I had already tried a whole bunch of demos and tests and at that point, I had a very good idea of how the simulator actually works. But, I needed a baseline and this came as a boon to me that Imperas-OVP had already made a demo platform that had the basic implementation of an ARMv8 Foundation platform. I needed to do a lot of modifications and also understand all the extra modules and their functionality, but at least I had a starting point! So now I am going to demonstrate the ARMv8 Foundation platform architecture and how its modules are written and implemented inside the Imperas-OVP simulator.

This Demo has the name **ARMv8-A-FMv1** and it can be installed directly from the OVP library. This specific demo has many useful guide files located inside its own folder. These are the **ArmV8LinuxNotes.txt** and **README.txt** and they are extremely helpful for understanding the platform functionality. The key modules of this platform are:

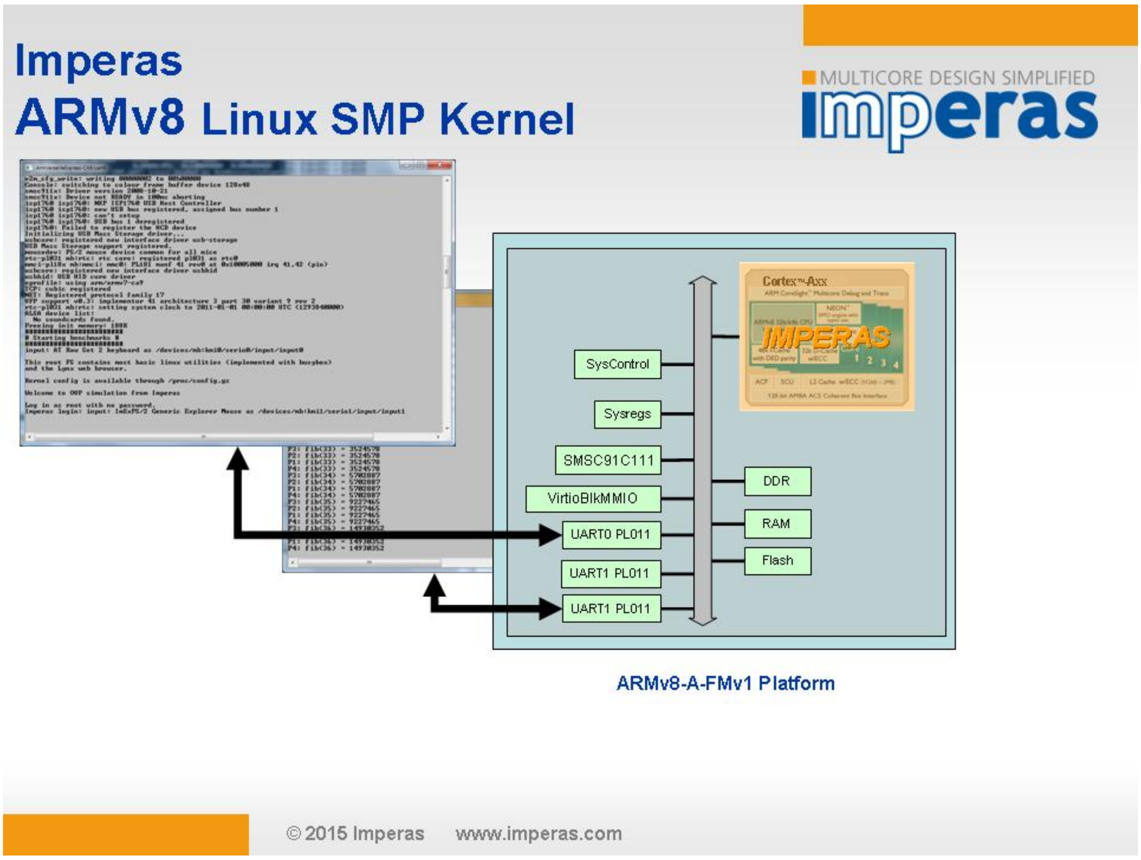
- **ARM Cortex-A57MPx4** processor. Imperas also provides the chance that we can use other Cortex-A armv8 processors **if we have already installed the aarch64 toolchain that I mentioned in a previous section**. For our tests, we used ARM

Cortex-A57MPx4, Cortex-A53MPx4, and Cortex-A72MPx4 with all possible options x1, x2, x3, and x4.

- GIC v2 or v3 (implemented in the processor model). ARM's GIC (General Interrupt Controller) architecture provides an efficient and standardized approach for handling interrupts in multi-core ARM-based systems (21).
- 8GB DRAM memory. At this point, I need to address that I have decided to allocate only 2 GB of DRAM locally. Inside the design and also by modifying the device tree, I have been able to allocate 6 GB (in other tests 8-16 GB) of **remote memory**, in order to simulate the memory disaggregation. I did that by applying a certain delay on the remote memory interconnect. The delay values of the remote access represent the estimated delay for the actual hardware and have been given to us by our associates!
- We also have some System Registers.
- SMSC LAN91C111. This is the dedicated Ethernet module of my platform. I capitalized on it in order to gain internet access on my final platform but I did all that via the OS not by adding anything to the existing model structure or configuration.
- Four UARTs. I am not going to use all four. To be exact, I only used the UART0 as my operating system's console. Through that, I was able to monitor the boot process of the OS and also use it as my gateway to interact with it.
- Of course, we also have a Virtio block device, which roughly is an I/O virtualization framework for Linux (22). I used this module to load the Ubuntu Linux OS rootfs image on my platform.
- The platform also includes a 'SmartLoaderArm64Linux' pseudo-peripheral that loads the device tree and initrd and inserts a small bit of boot code to initialize GIC and System registers and then jumps to the Linux entry with the proper values in GPR registers r0-r3. This can be used in lieu of having a boot loader such as 'uboot' in your simulation platform.
- All of the above, are interconnected by a 40bit address bus module, that is also implemented inside the common module.op.tcl that the whole platform is created with. **The modules are all 'called' from the ImperasLib source model 'pool' but the module.op.tcl file is where they are called and configured**!!!

There is a document that describes the details of the platform and its configuration options, memory map, and components in the file: $IMPERAS_HOME/ImperasLib/source /arm.ovpworld.org/module/ARMv8-A-FMv1/1.0/doc/ Imperas_Platform_User_Guide_ARMv8-A-FMv1.pdf.
More detailed definitions for all the modules are provided in specific guides inside the imperas installation allocated 'deep inside' the ImperasLib folder. A rough idea of how the platform architecture really looks like can be viewed in Figure11. Do not be bothered by all the unknown words you see in this picture, I will explain these later. As you can see, this design is very basic and can be seen as a simplification, or if you like, a more novice approach to a dRedBox microserver card. I made this compromise in order to be able to implement part of the dRedBox vision in my simulator. So only by adding a remote memory and configuring the network interconnect to better simulate the envisioned optical interconnect, I had an ace in my hands. But again, it was more workload than I initially though. Also at that point, I knew that this platform was capable of loading an Operating system and it had also a precompiled Kernel and device tree already up and running. **All**

I had to do was to modify them and start testing, but that was far easier said than done!



Imperas
ARMv8 Linux SMP Kernel

MULTICORE DESIGN SIMPLIFIED
imperas

```
v2m_cfg_write: writing 00000002 to 00500000
Console: switching to colour frame buffer device 128x48
smsc911x: Driver version 2008-10-21
smsc911x: Device not READY in 100ms aborting
isp1760 isp1760: NXP ISP1760 USB Host Controller
isp1760 isp1760: new USB bus registered, assigned bus number 1
isp1760 isp1760: can't setup
isp1760 isp1760: USB bus 1 deregistered
isp1760: Failed to register the HCD device
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
mousedev: PS/2 mouse device common for all mice
rtc-pl031 mb:rtc: rtc core: registered pl031 as rtc0
mmci-pl18x mb:mmci: mmc0: PL181 manf 41 rev0 at 0x1c050000 irq 41,42 (pio)
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
oprofile: using arm/armv7-ca9
TCP: cubic registered
NET: Registered protocol family 17
VFP support v0.3: implementor 41 architecture 3 part 30 variant 9 rev 2
rtc-pl031 mb:rtc: setting system clock to 2011-01-01 00:00:00 UTC (1293840000)
ALSA device list:
  No soundcards found.
Freeing init memory: 188K
##########################
# Starting benchmarks #
##########################
input: AT Raw Set 2 keyboard as /devices/mb:kmi0/serio0/input/input0

This root FS contains most basic linux utilities (implemented with busybox)
and the Lynx web browser.

Kernel config is available through /proc/config.gz

Welcome to OVP simulation from Imperas

Log in as root with no password.
Imperas login: input: ImExPS/2 Generic Explorer Mouse as /devices/mb:kmi1/serio1/input/input1
```

```
P3: fib(33) = 3524578
P2: fib(33) = 3524578
P1: fib(33) = 3524578
P4: fib(33) = 3524578
P3: fib(34) = 5702887
P2: fib(34) = 5702887
P1: fib(34) = 5702887
P4: fib(34) = 5702887
P3: fib(35) = 9227465
P2: fib(35) = 9227465
P1: fib(35) = 9227465
P4: fib(35) = 9227465
P3: fib(36) = 14930352
P1: fib(36) = 14930352
P4: fib(36) = 14930352
```

© 2015 Imperas www.imperas.com

Figure 10: ARMv8-A-FM-Linux-SMP baseline system design

Now that I have given you a rough idea of what my platform baseline contains let's start modifying and building our design. Again I have to state that it is not so different from a basic Microservercard with remote memory and network design as visioned in dRedBox! Also, all the ARMv8 Foundation platform modules that have nothing to do with the dRedBox microserver card, do not compromise the simulation result so we can leave them as they are. In the next sections, I will provide the Tcl code for creating the modules, all the steps to make the platform OS ready(it actually already is) and also the functionality of all the complimentary files and scripts that I have used in my implementation.

4.2.3 Building the platform

The ARMv8 Foundation Demo approach is close enough to our desired design and with some modifications it can come even closer. The ARMv8-A-FMv1 demo provided us with the baseline for a complete Linux OS booting, Internet connection and more importantly a good start, as it featured 2 examples of OS images ready to run on it (initrd and rootfs). In this section demonstrates all the components that we incorporated and how we configured them to create the final platform. This process starts by modifying the existing module.op.tcl given in the demo followed by modifying or adding what is needed along the way.

First, as in every programming language, we have to set some platform parameters appC. It is like setting up some 'defines' in a C code. Next, we initiate the new platform creation by issuing the **ihwnew** TCL command followed by our desired parameters along with the parameters we set earlier. Before we proceed, it is time to create the 40bit bus which is the one that interconnects the system together. The 40bit address value was taken directly from the ARMv8 Foundation platform model. From now on it will be referenced as **’pBus’** in all of the following code samples.

```
ihwaddbus -instancename pBus -addresswidth 40
```

With the baseline declarations and the common bus completed, it is time to integrate a processor design! An idea of the processor implementation is visible in the following code sample.

```
ihwaddprocessor -instancename cpu -type arm -vendor arm.ovpworld.org
-version 1.0 -endian little -simulateexceptions -mips 500
ihwconnect -instancename cpu -bus pBus -busmasterport INSTRUCTION
ihwconnect -instancename cpu -bus pBus -busmasterport DATA
ihwsetparameter -handle cpu -name variant -value Cortex-A57MPx4 -type
enum
```

After setting the ’version’, the ’endian’, activating the ’simulateexceptions’ and placing the nominal MIPS at 500, it is priority to connect the ’DATA’ and ’INSTRUCTION’ master ports to the pBus. In our platform, the Cortex-A57MPx4 processor was the default CPU option. The full declaration of this CPU is referenced in Appendix C appC, along with the use of the ’override_timerScaleFactor’ parameter. **When trying make a platform ’OS compatible’, it is crucial to know that the Kernel is not checking CNTFRQ register value, so at all times, the nominal MIPS value must match the CPU ’speed’ defined in the device tree.**

After initializing the processor we have to connect the supported interrupt request nets to the processor’s **Shared Peripheral Interrupt ports**. In this allocation of the Interrupt ports, we select which to use and which to remain idle appC.

Furthermore, a series of critical modules need to be added to the platform. Luckily, the Imperas library can provide all the necessary modules. These modules have already been addressed in the previous section.

First comes the **Ethernet module**. Imperas provides the SMSC 91C111 Ethernet device model. One thing to note is to connect it to the common pbus and also to an interrupt port (i.e. ir15).

Moving on, the ARMv8 Foundation model suggests the use of **System registers** and **3 RAM modules**, as mentioned in the previous section. These 3 ram modules are to be connected to the pBus to the addresses: 0x00000000->0x03ffffff, 0x04000000->0x0403ffff and 0x06000000->0x07ffffff. The ’ihwaddmemory’ command, adds the ram module and the ’ihwconnect’, links it to the common pBus

Next, we implement one of the most important modules, the **UART**. The ARMv8 Foundation model suggests four(4) UARTs despite the fact that, for this work, only one (UART0) was used. They are the interface to the actual hardware by providing the user with a Linux terminal-console after loading an OS image. They have to be connected to an interrupt port (i.e ir5,6,..). In addition, their parameters ’variant’, ’outfile’ and ’finishOnDisconnect’ should be set to ’ARM’, ’uart.log’ and ’1’ respectively, at least for our implementation.

Coming up, is the **Virtio block device**. This module is also connected to the common pBus to an address suggested by the ARMv8 Foundation v8 module(-loaddress 0 -hiaddress 0xffffffffff). This modules functionality has already been described, but keep in mind that we have to connect its dma master-port and bport1 slave-port to the pBus and Interrupt port respectively.

Second to last, we can find the **OVPworld ARM V8 Linux BootLoader**. This module's functionality has also been described earlier. One thing to note is that the physical base address starting point must be set to 0x80000000.

Last but not least, are the **DRAM** modules. In the end, we should have both local and external ones. The local and external configuration is done outside this file and has nothing to do with configuring the modules directly. All DRAM modules are created the same way. The local and remote attribute are presented after configuring the network interconnect via an intercept library. At this point we just need to add some memory bricks inside our design. Eventually, when the OS is loaded, we will have to apply the DRAM addressing to the exact point inside the device tree, in order for them to be visible from the Operating System. A sample code of a 2-block 8GB DRAM design is described below:

```
# 0x00_80000000 0x00_ffffffff DRAM 0GB-2GB
ihwaddmemory -instancename DRAM0 -type ram
ihwconnect -instancename DRAM0 -bus pBus -busslaveport sp1 -loaddress 0x0080000000 -hiaddress 0x0ffffffff
# 0x08_80000000 0x09_ffffffff DRAM 2GB-8GB
ihwaddmemory -instancename DRAM1 -type ram
ihwconnect -instancename DRAM1 -bus pBus -busslaveport sp1 -loaddress 0x0880000000 -hiaddress 0x09ffffffff
```

After all this initial work is done and the new module.op.tcl file is completed, it is time to 'make'. Imperas provides excellent **Makefiles** for this particular part of the implementation. Inside the ARMv8-A-FMv1 Demo folder, are a series of Makefiles but at this point, we only have to focus on the one inside the demo's module folder. This makefile has the task of including a designated makefile that is capable of creating this module platform. After its execution, many files are generated but the most important and the one that we use is the model.$(IMPERAS_ARCH).$(IMPERAS_SHRSUF). For example, a model.Linux32.so file was generated on our host. That completes the creation of the platform module.

I would like to highlight that although all these steps might seem trivial, it was a really difficult and time-consuming process to complete. All the configuration and parameter values are not arbitrary, but specifically implemented to provide an as close as possible implementation of the ARMv8 Foundation module that also meets our platform requirements. The parameters were provided to us by ARM and Imperas repositories and also other project partners. Obviously, the Imperas demo platform was a great baseline and we incorporated many of its provided features, like Makefiles, tools, modules and boot scripts in our design.

A more detailed description and guidelines for building this platform can be found in Appendix C appC. Also a complete design of this module is depicted in a chart at the end of this chapter

4.2.4 External Kernel and the Device Tree

First, we are going to talk about the obvious one: **The Kernel Image file**. As far as I am concerned I was lucky on this part because inside the Linux_ARMv8-A-FMv1 demo package there was already a pre-compiled Kernel image suitable for this platform design. All I had to do was simply integrate it somewhere inside my design (well not quite inside) and designate it on the OS boot sequence. I will be mentioning that when I present to you the boot scripts and how I used this pre-compiled Kernel Image.

Inside this demo were also a lot of o guides as I mentioned earlier and there is an extensive tutorial about how to get things started and how to initiate an OS boot. This is where this particular precompiled Kernel Image comes into play as it allows the completion of this process. Without it, we would have to create a new Kernel image from scratch. Luckily, I did not utilize it. However, there is a way to rebuild the Kernel image and is described inside the **ARMv8LinuxNotes** guide that I mentioned previously.

The pre-compiled Linux kernel was built from source, downloaded from the linaro.org site. You can view the Linux/Makefile for details of the commands used to download and build the Linux kernel. To recompile the Linux Kernel, the aarch64-linux-gnu cross compiler is required for building. Also, the Linux/Makefile should be edited to point to the location of the cross-compiler on your system. The device tree also plays a key role in the creation of the Linux Kernel. **Something I would like to highlight is that although I made some changes to the device tree, I was still able to utilize the default given Kernel Image.**

Furthermore, we must have **a Device Tree**. Imperas had given me an easy introduction, as this demo also had a device tree, specifically designed for the ARMv8 Foundation Model. Although it was not perfect, it was a great baseline. **It is worth noting that as I was working on this device tree I managed to highlight one error in the memory addressing and informed OVP about it. This feedback will become a fix in their later simulator patches!**

In order to 'tweak' the device tree to suit my needs, I needed some specific tools to do it. But first, let me talk about the changes I made. Although we specified that the DRAM modules were 2GB and 6GB each, we need to provide that addressing inside the device tree as well. Also, all other platform modules should be described inside the device tree, otherwise the OS would not be able to utilize it.For my part, I had to modify the memory allocation addressing in order to utilize the entire 8GB of combined DRAM. Of course, if we make changes to the DRAM sizes, as I did for my tests, we also have to update the device tree accordingly.

Making changes to the device tree is pretty straight forward. First, we have to locate the desired code block that we need to change inside the device tree. We open the previous **foundation-v8.dts** device tree file from inside a Linux terminal using either **'vi' or 'nano'** to configure it. Also, note that the Linux_ARMv8-A-FMv1 demo provides 2 more device tree files apart from the one I used. **My point of interest was the memory specification and that is where I found that although I have configured 8GB of total DRAM inside the module.op.tcl file, I had to declare the addressing allocation here as well**. Next, I will demonstrate a portion of the device tree foundation-v8.dts file that shows the DRAM memory addressing.

```
memory@80000000 {
device_type = "memory"; reg = <0x00000000 0x80000000 0 0x80000000>,
<0x00000008 0x80000000 0x00000001 0x80000000>; }
```

As you can see this depicts the allocation of 2 separate 2GB and 6GB DRAM blocks. In my tests in order to implement more combinations of memory sizes (i.e. 1-7, 2-6, 4-4), the only thing I had to do was to change the addressing above and also the addressing inside the module.op.tcl file.

Now that we have reconfigured the device tree .dts file, it is time to recompile it. To do this. we have to install the **device-tree-compiler**. After that initial step, if it exists, we delete the previous .dtb file. As an example, in my case, I deleted the foundation-v8.dtb file that came with the demo installation. Next, the only thing to do is:

```
dtc -O dtb -o file_name.dtb file_name.dts
```

At last, the Kernel Image and the newly configured Device Tree are ready to use. **It is time to load the OS!**

4.2.5 The Custom rootfs image, 'RUN' Scripts and the Internet Access.

We decided to approach the OS implementation with a root filesystem(rootfs) image. That is why the VIRTIO block device peripheral was included on this platform. There are several prebuilt rootfs images for AArch64 Linux available at the 'linaro.org' repository, that are mainly suggested for testing if a platform works. The real task was to load a fully functional Ubuntu16.04 Linux Operating System rootfs image.

Luckily, we had a good partner on this particular task. At that time, **D. Syrivelis** from the University of Thessaly happened to be working on a Linux image for his own project. After contacting him and explaining what we needed, he sent us his basic Ubuntu16.04 Linux Operating System rootfs image. This took a lot of work out of our hands and I honestly thank him for that.

With the Ubuntu rootfs image ready, we turned our attention to loading the OS image on our simulated platform. In order to load the OS, I created a starting script, by configuring a baseline script that I found inside the Demo platform. The 'RUN.sh' script that I used to load the OS is featured below. Thus, all the parameters have the values that we used and will vary from a different design. However, I think that this demonstration will be a perfect way to explain the booting procedure.

```
#!/bin/sh
MODULE=ARMv8-A-FMv1
harness.exe \
–modulefile ../module/model.${IMPERAS_ARCH}.${IMPERAS_SHRSUF}
\ –output imperas.log –verbose \
–startaddress 0x80000000 \
\
–override ${MODULE}/cpu/variant=Cortex-A57MPx4 \
–override ${MODULE}/smartLoader/kernel=../Linux/Image \
–override ${MODULE}/smartLoader/dtb=../Linux/foundation-v8.dtb \
\
–override ${MODULE}/uart0/console=1 \
\
–override ${MODULE}/smartLoader/append="root=/dev/vda1 rw console=
ttyAMA0 –" \
–override ${MODULE}/vbd0/VB_DRIVE=ubuntu16.04arm64.img \
–override ${MODULE}/vbd0/VB_DRIVE_DELTA=0
"$"
```

So what does this script do exactly? It will be explained step-by-step.

First of all, I have to clarify that this is done using the **Harness** program. As I mentioned earlier, the Harness program enables simulation of virtual platforms without the need to create a test harness or test bench. All we need to do is call the harness.exe and make all the necessary configurations.

Obviously, we begin by defining the name of the platform {**MODULE=ARMv8-A-FMv1**} that we created. Next we call the **harness.exe** and start adding a series of configuration values. First we specify where the **modulefile** is located. In our case, we point to the **model.Linux32.so** file. Next, we specify that we want to print an **output** but also enable the **–verbose** output results. Moving on, we set the **startaddress** as I mentioned in a previous section at **0x80000000**. Then, we **override** some parameters to the module that we created, starting with the **cpu variant**. In this case I demonstrate the Cortex-A57MPx4, but I also used the 53 and the 72 with all possible x1,x2,x3 and x4 combinations. Next, we override the **smartLoader** parameters in order to point them to the correct **Kernel Image** and **device tree** location. In addition, the console parameter of the **UART0** model will be set to 1, in order to provide the OS console.

Before pointing to the rootfs image, we have to set the **append** parameter of the smartLoader to the desired **"root=/dev/vda1 rw console=ttyAMA0 –"**. Afterwards, we point the module's **vb0/VB_DRIVE** to the location of the **Linux Ubuntu 16.04 rootfs image file**.

Last but not least, we have to set the module **VB_DRIVE_DELTA** parameter. This parameter can be set to 1 in order to prevent the simulator from modifying the rootfs image file during the simulation, or 0 to let the simulator modify it. In short, to keep any changes (i.e. newly installed packages) inside the image, this parameter should be set to 0.

With all these steps completed, the script is ready. We can simply execute it and load the OS onto our simulated platform. Figure.11 depicts a sample of the UART0 output. In the next chapter, I will also demonstrate the **–verbose** output and other simulator results of similar OS simulations.

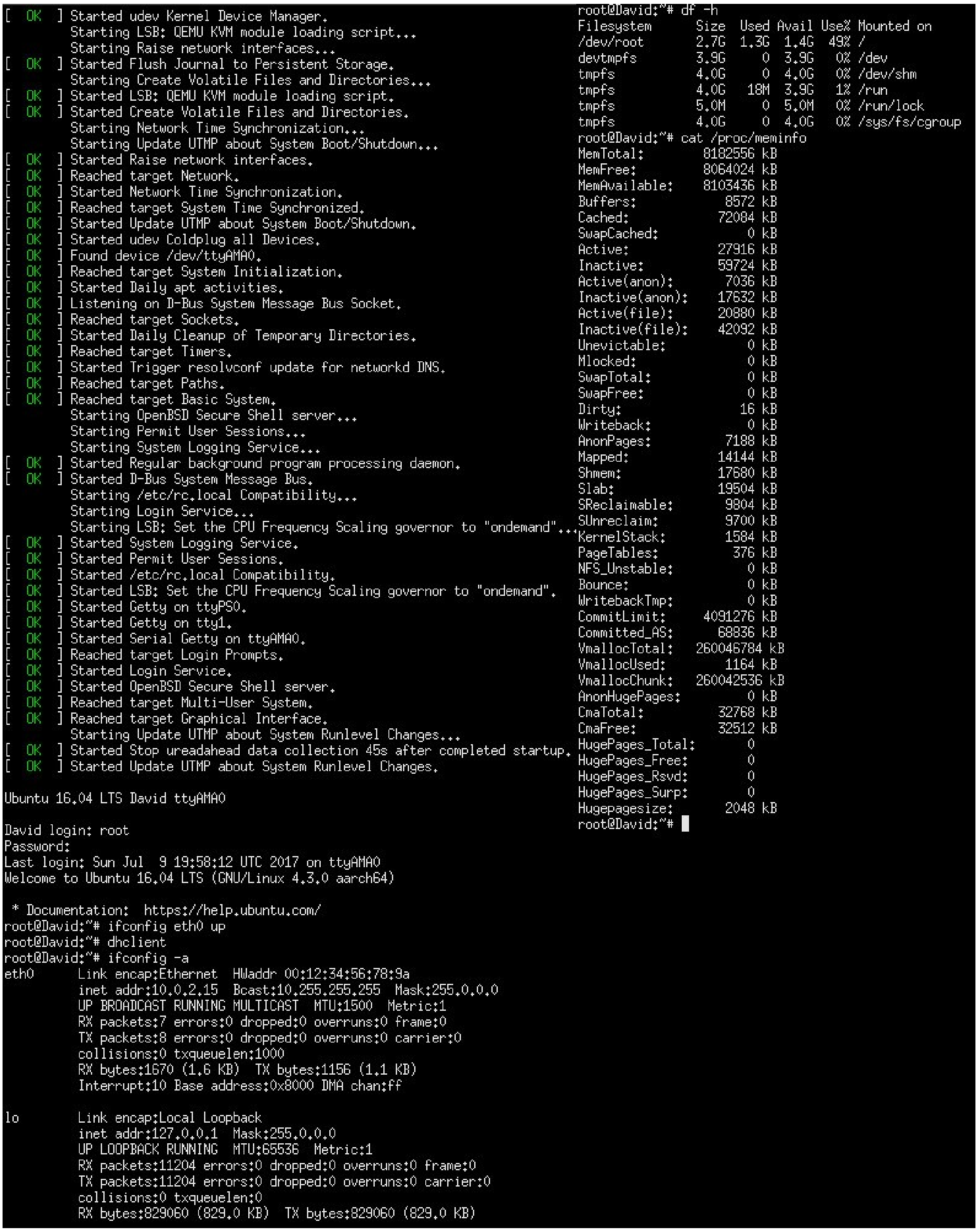
To complete this 'Linux installation' the platform requires INTERNET Access!. In Figure11, the **ifconfig** output shows the **eth0** interface activated and that networking is also activated. This feature does not come as a default. First we need to install a virtual bridge on the host machine. Using this bridge, we can connect the simulated platform to the host's network-manager, thus gaining INTERNET access. Every time the host is restarted, we need to allocate the network interface to the virtual bridge so that the networking can be re-activated. After this procedure, we boot into our simulated rootfs Image and simply activate the network interface (i.e. eth0). The platform can now access the webin.

4.2.6 Simulating the Network Delay using an Intercept Library

After discussing with one of the OVP forum administrators Duncan Graham, we decided that the best way to simulate the dRedBox interconnect delay modeling was by using the **vmirtAddSkipCount** function. This is a Run Time Function provided by the Imperas VMI API. This function, adds to the count of skipped nominal instructions (or cycles) of a designated a processor. **This function is typically used inside an intercept library to approximate timing effects, so it is perfect for our job**. This function takes as parameters a targeted processor that in the Imperas simulator are instantiated with a nominal MIPS rate. It also takes as a parameter the number of steps that you need to advance (skip). **So, the processor nominal instruction count is advanced by the specified count, reducing the 'budget' available for execution of instructions**. For example, if a processor with skip count 4 is required to execute 10 nominal instructions, then only 6 true instructions will be executed. **The skipCount number that I used is not arbitrary but it is calculated along with the native platform's interconnect delay values, given to us by our partners**(18).

As I said, this function is mainly used inside an intercept library, so we have to create one for our platform. The intercept library will be written in 'C' and from now on we will refer to it as **intercept.c**. The intercept library that we used our baseline was located in the provided Binary Interception Imperas Examples.

Inside the new intercept.c file, the first task is to include all the VMI API header files (14) along with the packages that we need, like the 'stdio.h'. Right after, we initiate the Structure type containing local data for this plugin. At this point, we also declare any other variable that we are going to use inside our library.



```
[  OK  ] Started udev Kernel Device Manager.
         Starting LSB: QEMU KVM module loading script...
         Starting Raise network interfaces...
[  OK  ] Started Flush Journal to Persistent Storage.
         Starting Create Volatile Files and Directories...
[  OK  ] Started LSB: QEMU KVM module loading script.
[  OK  ] Started Create Volatile Files and Directories.
         Starting Network Time Synchronization...
         Starting Update UTMP about System Boot/Shutdown...
[  OK  ] Started Raise network interfaces.
[  OK  ] Reached target Network.
[  OK  ] Started Network Time Synchronization.
[  OK  ] Reached target System Time Synchronized.
[  OK  ] Started Update UTMP about System Boot/Shutdown.
[  OK  ] Started udev Coldplug all Devices.
[  OK  ] Found device /dev/ttyAMA0.
[  OK  ] Reached target System Initialization.
[  OK  ] Started Daily apt activities.
[  OK  ] Listening on D-Bus System Message Bus Socket.
[  OK  ] Reached target Sockets.
[  OK  ] Started Daily Cleanup of Temporary Directories.
[  OK  ] Reached target Timers.
[  OK  ] Started Trigger resolvconf update for networkd DNS.
[  OK  ] Reached target Paths.
[  OK  ] Reached target Basic System.
         Starting OpenBSD Secure Shell server...
         Starting Permit User Sessions...
         Starting System Logging Service...
[  OK  ] Started Regular background program processing daemon.
[  OK  ] Started D-Bus System Message Bus.
         Starting /etc/rc.local Compatibility...
         Starting Login Service...
         Starting LSB: Set the CPU Frequency Scaling governor to "ondemand"...
[  OK  ] Started System Logging Service.
[  OK  ] Started Permit User Sessions.
[  OK  ] Started /etc/rc.local Compatibility.
[  OK  ] Started LSB: Set the CPU Frequency Scaling governor to "ondemand".
[  OK  ] Started Getty on ttyPS0.
[  OK  ] Started Getty on tty1.
[  OK  ] Started Serial Getty on ttyAMA0.
[  OK  ] Reached target Login Prompts.
[  OK  ] Started Login Service.
[  OK  ] Started OpenBSD Secure Shell server.
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.
         Starting Update UTMP about System Runlevel Changes...
[  OK  ] Started Stop ureadahead data collection 45s after completed startup.
[  OK  ] Started Update UTMP about System Runlevel Changes.

Ubuntu 16.04 LTS David ttyAMA0

David login: root
Password:
Last login: Sun Jul  9 19:58:12 UTC 2017 on ttyAMA0
Welcome to Ubuntu 16.04 LTS (GNU/Linux 4.3.0 aarch64)

 * Documentation:  https://help.ubuntu.com/
root@David:~# ifconfig eth0 up
root@David:~# dhclient
root@David:~# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:12:34:56:78:9a
          inet addr:10.0.2.15  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1670 (1.6 KB)  TX bytes:1156 (1.1 KB)
          Interrupt:10 Base address:0x8000 DMA chan:ff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:11204 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11204 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:829060 (829.0 KB)  TX bytes:829060 (829.0 KB)
```

```
root@David:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       2.7G  1.3G  1.4G  49% /
devtmpfs        3.9G     0  3.9G   0% /dev
tmpfs           4.0G     0  4.0G   0% /dev/shm
tmpfs           4.0G   18M  3.9G   1% /run
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           4.0G     0  4.0G   0% /sys/fs/cgroup
root@David:~# cat /proc/meminfo
MemTotal:        8182556 kB
MemFree:         8064024 kB
MemAvailable:    8103436 kB
Buffers:            8572 kB
Cached:            72084 kB
SwapCached:            0 kB
Active:            27916 kB
Inactive:          59724 kB
Active(anon):       7036 kB
Inactive(anon):    17632 kB
Active(file):      20880 kB
Inactive(file):    42092 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                16 kB
Writeback:             0 kB
AnonPages:          7188 kB
Mapped:            14144 kB
Shmem:             17680 kB
Slab:              19504 kB
SReclaimable:       9804 kB
SUnreclaim:         9700 kB
KernelStack:        1584 kB
PageTables:          376 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     4091276 kB
Committed_AS:      68836 kB
VmallocTotal:   260046784 kB
VmallocUsed:        1164 kB
VmallocChunk:   260042536 kB
AnonHugePages:         0 kB
CmaTotal:          32768 kB
CmaFree:           32512 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
root@David:~# █
```

Figure 11: UART0 console sample

Now that the initial steps are done, it is time for the main configuration. First of all, we create a **Constructor** function. In this function we set the **read and write** callbacks on the processor data domain. More specifically, we set callbacks on a DRAM memory address range to monitor incoming and outgoing 'traffic'. That way, we know exactly when these addresses are accessed, thus we can simulate the interconnect delay and create the remote DRAM effect.

After that, we can write the **Destructor** function, from which we can print out all the output results that we want from the intercept library. For my design, I used the

Destructor function to print the total number of read and write callbacks, the program counter, the issued instructions, the skipped instructions, the calculated executed instructions (issued-skipped) and the instructions that were actually executed (vmirtGetExecutedICount value) for every processor inside the platform. We made use of the VMI Run Time functions vmirtProcessorName, vmirtGetICount, vmirtGetSkipCount and vmirtGetExecutedICount. These functions will be thoroughly addressed in chapter5.

The only thing left to do, is to implement the functions that will be triggered every time a read or write callback is activated. Inside those functions I will be using the **vmirtAddSkipCount** function that I mentioned earlier, with specific parameters for my design. The **processor** parameter will point to the current platform processor and is a variable that gets its value once we link the intercept library with our design. The **SkipCount** delay value is the one that we set according to the specifications for the network interconnect that we want to simulate.

A more detailed description and guidelines for building this intercept library can be found in Appendix D appD. Do not forget that we still need to 'make' this intercept.c file in order to use it. There is a specific **Makefile.intercept** to do this task.

4.3 Completing the Simulated Platform Emulation

With everything set, it is time to merge everything together! You might argue that this is done in the RUN.sh script that is used to boot the system. That is not the case here. We have to 'make' all the modules for this platform and by that I mean configuring a platforms Makefile. For better understanding, I will now demonstrate the Makefile created for our simulated system:

```
IMPERAS_HOME := $(shell getpath.exe "$(IMPERAS_HOME)")
include $(IMPERAS_HOME)/bin/Makefile.include
all: intercept loadExtLib module
module::
$(MAKE) -C module NOVLNV=1
intercept::
$(MAKE) -f Makefile.intercept NOVLNV=1
loadExtLib::
$(V) echo "-extlib ARMv8-A-FMv1/cpu=model.${IMPERAS_SHRSUF}" >
lib.ic
clean:
$(MAKE) -C module NOVLNV=1 clean
$(MAKE) -C intercept NOVLNV=1 clean
```

This Makefile includes the provided **$(IMPERAS_HOME)/bin/Makefile.include**. First, we 'make' the module and the intercept library. **The next step is to load the 'external' intercept library, link the CPUs with the intercept module.so file and also create the lib.ic file**! The 'clean', is just to remove the files from previous 'makes' in order to avoid duplicates or false makes. A complete idea of the simulated platform is shown in the following Figure. After the execution of the Makefile, we run the 'RUN.sh' scripts and load our Operating System, having the intercept library running in the background.

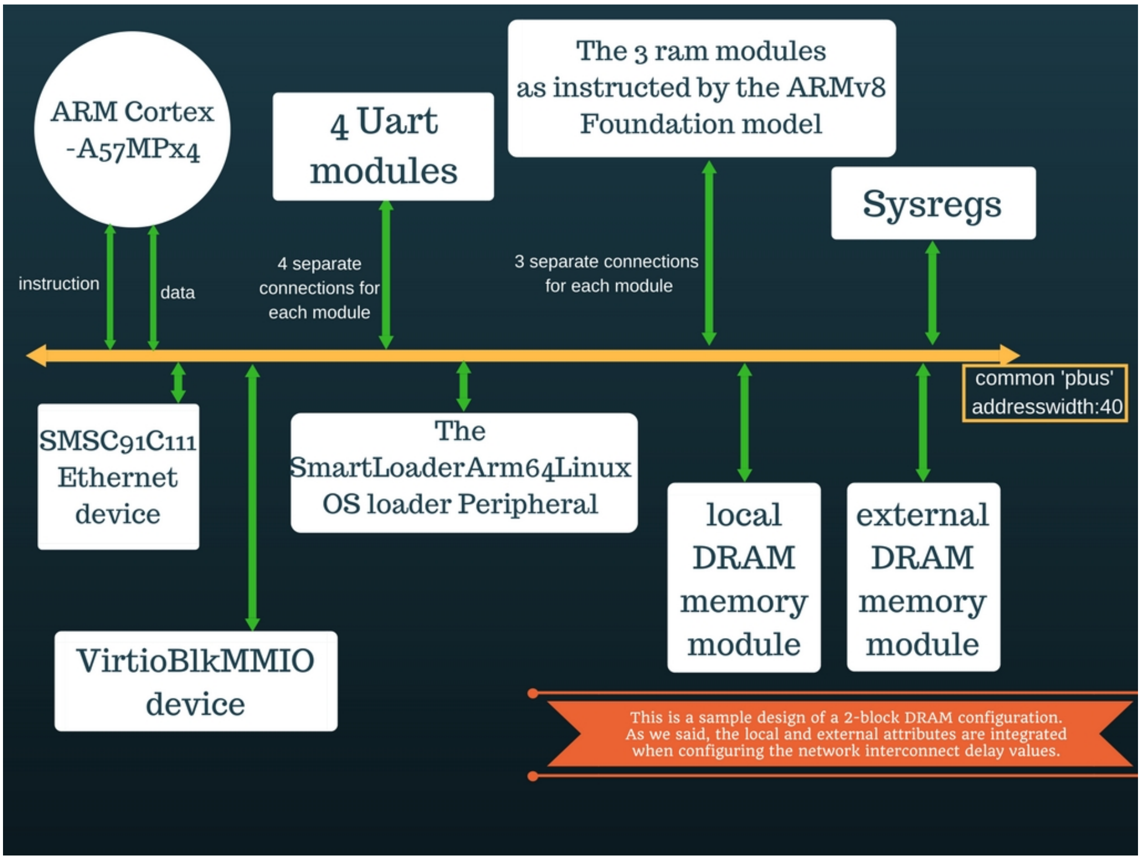

ARM Cortex -A57MPx4
4 Uart modules
The 3 ram modules as instructed by the ARMv8 Foundation model
Sysregs
instruction
data
4 separate connections for each module
3 separate connections for each module
common 'pbus' addresswidth:40
SMSC91C111 Ethernet device
The SmartLoaderArm64Linux OS loader Peripheral
local DRAM memory module
external DRAM memory module
VirtioBlkMMIO device
This is a sample design of a 2-block DRAM configuration. As we said, the local and external attributes are integrated when configuring the network interconnect delay values.

Figure 12: The Simulated Platform Design. The intercept library is not connected to the pBus so it is not shown. It is, however, 'linked' to this platform.

5 Results and Evaluation

At this point, it is time to go through all the results that came from my work and also evaluate them as to if they are going to help advance this research procedure. I will also go through the simulator's result feedback and other monitoring functions provided by Imperas. We are going to talk about the end game of this diploma thesis, what we have accomplished and all the things that we managed or tried to do along the way. Mainly, I will try to emphasize on what was the outcome of this research, what tools helped us evaluate along the way and what does this work has to offer in the development and evolution of the dRedBox project and the disaggregated data center innovation approach.

5.1 Verbose outputs and other Simulator Feedback

5.1.1 Verbose outputs

At the end of the simulation, if the verbose flag has been used, some statistics will be reported by the simulator. This output provides pieces of information that indicate how many instructions were executed on the Imperas-OVP Processor model and the equivalent simulated Millions of Instructions Per Second (MIPS) that were executed. **This can be used as an indication of how well the simulator performed on the host machine running a particular application and not how the application performs on the simulator platform**. The typical simulation statistics is shown below. Most entries are self-explanatory but I am going to describe them as we progress. **The actual**

performance reported may vary and depends on the performance of the native host.

- **OVP Fast processor model configuration information**: The output **'Type'**, shows the processor model type and, if selected, the configuration variant. **'Nominal MIPS'** by default is 100, unless you set a new one while building the processor model as we did in the module.op.tcl file. With 'wallclock' enabled, you can control the maximum execution performance i.e. only 100 million instructions will be executed in a real time 1 second. In a multi-processor system (without wallclock enabled) it provides an execution between processors, e.g. a 200 MIPS processor will execute twice the instructions as a 100 MIPS processor in a simulation unit of time.

- **Program execution information**: The **'Final program counter'** can be useful as an easy way of showing that the application program executes in a deterministic way over a number of simulation runs. The **'Simulated instructions'** will vary depending upon the application being executed. **This count indicates the number of simulated processor instructions for each processor in the platform**.

- **Simulated MIPS**: **The 'Simulated MIPS' is a measure of the number of 'Simulated instructions' over the 'host elapsed time' and is an indication of the performance of the simulator**. There is a case that this output might print 'run too short for meaningful result'. This happens because there is an overhead at the start of each simulation with Just-In-Time code morphing simulators and so the Simulated MIPS are not calculated if the simulated runtime is below a threshold. For each processor, the 'Simulated MIPS' line gives the rate at which instructions for that processor were executed in wall-clock time (do not confuse it with the Imperas' 'wallclock' feature). In other words, the simulated MIPS number for a processor is calculated by dividing the number of instructions executed by that processor by the elapsed time for the simulation process. As an example, the reported simulated MIPS for cpu1 is calculated by dividing the simulated instructions (i.e. 1,658,997,966) by the elapsed time (i.e. 2.12 seconds) to give approx. 781.2!
However in many tests this was not the case. I made my own calculations and found that the 'Simulated MIPS' value is better calculated as a result of the math: (Simulated Instructions)/(User time + System time) and that is something that should be stated!!!

- **Simulation time statistics**:
Simulated time, is the time it would have taken to run an application based on the MIPS configuration of the modeled processor and the number of instructions the application took to complete. The time fields are real-time information from the native Host machine relating to the simulator execution. It is the duration of the simulation in simulated time. This corresponds exactly to the notion of time in a simulation language such as Verilog and VHDL and is entirely unrelated to wall-clock time.
User time is the time that the simulation process spent executing instructions on the host machine.
System time, is the time the host machine spent in the system while executing instructions on behalf of the simulation process.
Elapsed time, is the overall time taken by the simulation process on the host from start to finish.
All three of these times will vary from run to run, depending on the host load average and other factors.
The 'Real-time ratio' is an indication of how much quicker the simulator was able

to execute the processor instructions than the real silicon based upon the OVP Fast Processor Model MIPS rate configuration and the time the simulator took to execute the instructions on the native host machine. **In short 'Real-time ratio' shows how much faster than the real time a simulation ran**.

5.1.2 Other Simulator Feedback

I mentioned in the previous chapter that I have created an intercept library in order to apply some specific delay values to my system interconnect. I did that by using a function from the VMI API and more specifically the VMI Run-Time Functions of that API. Many of these API's functions can be used to provide useful feedback about the simulator executions as well as how the platform actually performs. In my platform I used:

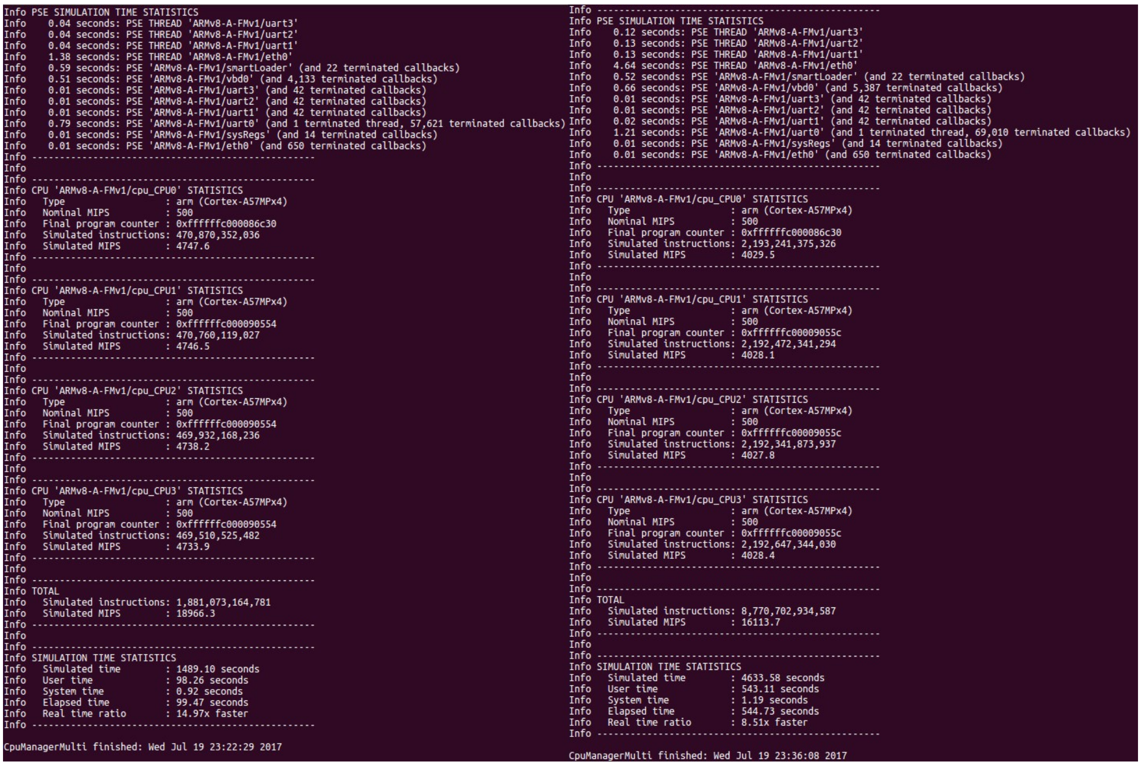
- **vmirtGetPC(processor)**: This function returns the processor's program counter. **I added this function, to see if the PC number matches the Final program counter of the Verbose output**.
- **vmirtGetICount(processor)**: The simulator maintains a count of executed instructions for every processor in a simulation. This function call returns the count for a specific processor.
- **vmirtGetSkipCount(processor)**: This function returns the total number of skipped instructions accumulated by calls to vmirtAddSkipCount function that I used to simulate my interconnect delay.
- **vmirtGetExecutedICount(processor)**: This function returns the number of instructions that were actually executed after simulating all those Skips!

By printing the return values of these functions from inside the Destructor function, we can export results regarding every processor inside the simulated platform. For example, when the ARM Cortex-A57MPx4 processor is used, we get information for all 4 simulated CPUs.

5.2 Simulation of the Linux OS

First of all, the platform actually works! But, let us not forget that running the Linux OS on the simulated platform is a simulation as well. Now that we have a working OS on the platform we can run simulations via that OS, but we can also take into account the simulation results that come out just by running the OS on this platform! The workload of creating a platform capable of both resembling a dRedBox server blade but also running OS was not an easy task. **So, it is mandatory that the OS simulation gets a separate section of some simulation results resembling only its execution on the simulated platform**. In the following figures, the verbose results of the OS execution/simulation are presented, along with the intercept library outputs. Note that on the left we have a normal simulation and on the right a delayed (18) one. **These results resemble the execution of the OS like running any bare metal application on the simulated platform**. I also have to state that these runs are executed with 'wallclock' appA disabled.

With the delay library activated, the boot sequence lasts way longer. That is obvious because this is a simulation and the delays are an additional simulated part of the whole process! There is also a decrease in the MIPS values of every processor, because the simulator is struggling to simulate the delays/SkipCounts in real time. Also, because of all this 'waiting', the UART console/terminal is less responsive and appears to be lagging, but this mainly happens because of the remote memory access delay simulation.



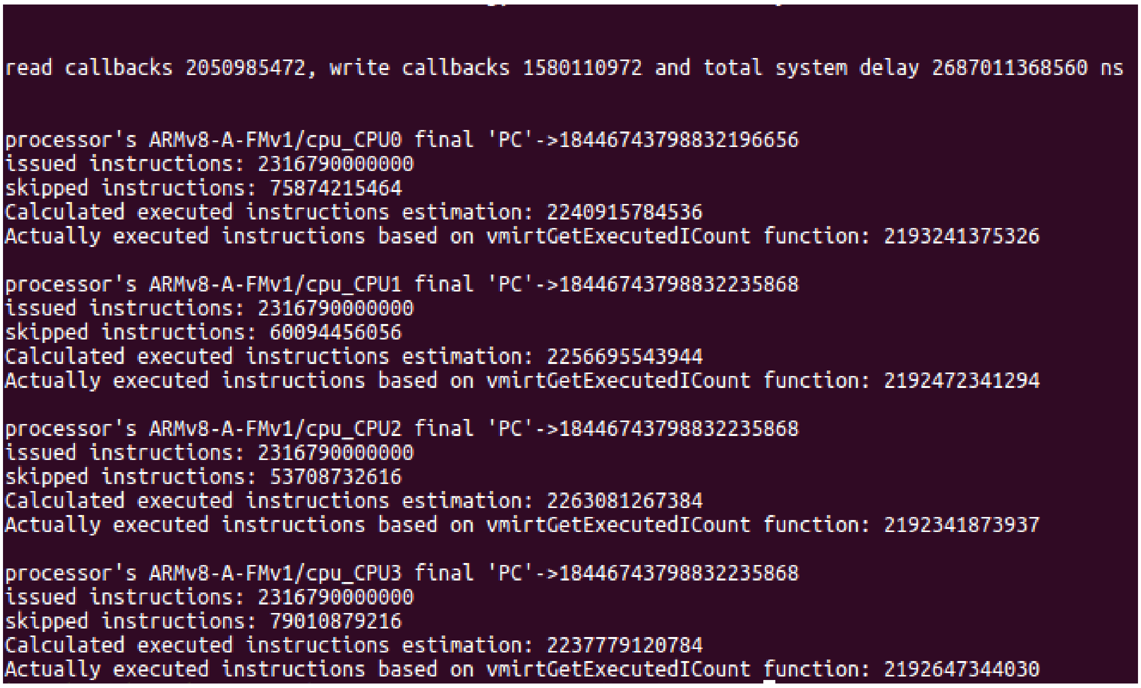
```
Info PSE SIMULATION TIME STATISTICS
Info    0.04 seconds: PSE THREAD 'ARMv8-A-FMv1/uart3'
Info    0.04 seconds: PSE THREAD 'ARMv8-A-FMv1/uart2'
Info    0.04 seconds: PSE THREAD 'ARMv8-A-FMv1/uart1'
Info    1.38 seconds: PSE THREAD 'ARMv8-A-FMv1/eth0'
Info    0.59 seconds: PSE 'ARMv8-A-FMv1/smartLoader' (and 22 terminated callbacks)
Info    0.51 seconds: PSE 'ARMv8-A-FMv1/vbd0' (and 4,133 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/uart3' (and 42 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/uart2' (and 42 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/uart1' (and 42 terminated callbacks)
Info    0.79 seconds: PSE 'ARMv8-A-FMv1/uart0' (and 1 terminated thread, 57,621 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/sysRegs' (and 14 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/eth0' (and 650 terminated callbacks)
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU0' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc000086c30
Info   Simulated instructions: 470,870,352,036
Info   Simulated MIPS        : 4747.6
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU1' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc000090554
Info   Simulated instructions: 470,760,119,027
Info   Simulated MIPS        : 4746.5
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU2' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc000090554
Info   Simulated instructions: 469,932,168,236
Info   Simulated MIPS        : 4738.2
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU3' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc000090554
Info   Simulated instructions: 469,510,525,482
Info   Simulated MIPS        : 4733.9
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info TOTAL
Info   Simulated instructions: 1,881,073,164,781
Info   Simulated MIPS        : 18966.3
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 1489.10 seconds
Info   User time             : 98.26 seconds
Info   System time           : 0.92 seconds
Info   Elapsed time          : 99.47 seconds
Info   Real time ratio       : 14.97x faster
Info --------------------------------------------------

CpuManagerMulti finished: Wed Jul 19 23:22:29 2017
```

```
Info --------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    0.12 seconds: PSE THREAD 'ARMv8-A-FMv1/uart3'
Info    0.13 seconds: PSE THREAD 'ARMv8-A-FMv1/uart2'
Info    0.13 seconds: PSE THREAD 'ARMv8-A-FMv1/uart1'
Info    4.64 seconds: PSE THREAD 'ARMv8-A-FMv1/eth0'
Info    0.52 seconds: PSE 'ARMv8-A-FMv1/smartLoader' (and 22 terminated callbacks)
Info    0.66 seconds: PSE 'ARMv8-A-FMv1/vbd0' (and 5,387 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/uart3' (and 42 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/uart2' (and 42 terminated callbacks)
Info    0.02 seconds: PSE 'ARMv8-A-FMv1/uart1' (and 42 terminated callbacks)
Info    1.21 seconds: PSE 'ARMv8-A-FMv1/uart0' (and 1 terminated thread, 69,010 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/sysRegs' (and 14 terminated callbacks)
Info    0.01 seconds: PSE 'ARMv8-A-FMv1/eth0' (and 650 terminated callbacks)
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU0' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc000086c30
Info   Simulated instructions: 2,193,241,375,326
Info   Simulated MIPS        : 4029.5
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU1' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc00009055c
Info   Simulated instructions: 2,192,472,341,294
Info   Simulated MIPS        : 4028.1
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU2' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc00009055c
Info   Simulated instructions: 2,192,341,873,937
Info   Simulated MIPS        : 4027.8
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'ARMv8-A-FMv1/cpu_CPU3' STATISTICS
Info   Type                  : arm (Cortex-A57MPx4)
Info   Nominal MIPS          : 500
Info   Final program counter : 0xffffffc00009055c
Info   Simulated instructions: 2,192,647,344,030
Info   Simulated MIPS        : 4028.4
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info TOTAL
Info   Simulated instructions: 8,770,702,934,587
Info   Simulated MIPS        : 16113.7
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 4633.58 seconds
Info   User time             : 543.11 seconds
Info   System time           : 1.19 seconds
Info   Elapsed time          : 544.73 seconds
Info   Real time ratio       : 8.51x faster
Info --------------------------------------------------

CpuManagerMulti finished: Wed Jul 19 23:36:08 2017
```

Figure 13: Verbose results with(right) and without(left) the intercept delay library.



```
read callbacks 2050985472, write callbacks 1580110972 and total system delay 2687011368560 ns


processor's ARMv8-A-FMv1/cpu_CPU0 final 'PC'->18446743798832196656
issued instructions: 2316790000000
skipped instructions: 75874215464
Calculated executed instructions estimation: 2240915784536
Actually executed instructions based on vmirtGetExecutedICount function: 2193241375326

processor's ARMv8-A-FMv1/cpu_CPU1 final 'PC'->18446743798832235868
issued instructions: 2316790000000
skipped instructions: 60094456056
Calculated executed instructions estimation: 2256695543944
Actually executed instructions based on vmirtGetExecutedICount function: 2192472341294

processor's ARMv8-A-FMv1/cpu_CPU2 final 'PC'->18446743798832235868
issued instructions: 2316790000000
skipped instructions: 53708732616
Calculated executed instructions estimation: 2263081267384
Actually executed instructions based on vmirtGetExecutedICount function: 2192341873937

processor's ARMv8-A-FMv1/cpu_CPU3 final 'PC'->18446743798832235868
issued instructions: 2316790000000
skipped instructions: 79010879216
Calculated executed instructions estimation: 2237779120784
Actually executed instructions based on vmirtGetExecutedICount function: 2192647344030
```

Figure 14: Intercept Library Outputs

Frankly, I left the delayed simulation run longer mainly because I needed a bigger sample and also due to the simulator running much slower. This is depicted in the total number of simulated instructions being way bigger that the non-delayed one and also the time metrics. The duration of these simulations was not the same because that was not the point. We were only interested in finding out about the simulation results after just booting the OS. Note that each simulation must be ended by issuing 'halt', as instructed by the Imperas platform manual, followed by closing the UART console, in order to get the results on the host's terminal without damaging the Linux Kernel.

Let us discuss the verbose results. **On the left, we have the verbose output without the intercept library and on the right with the library enabled.** The MIPS might appear 500 for each processor, but we have to know the value of the 'override_timerScaleFactor'. For our tests, we have set the 'override_timerScaleFactor=5', so the actual clock is at 500/5=100MHz to match the CPU frequency inside our device tree.

A great thing that we noticed, is that the TOTAL MIPS value is spread evenly across all platform processors. Also, with 'wallclock' disabled, I had to be quick to enter both username and password otherwise the login prompt timed out(as the guides suggested)! But after 1-2 tries it is possible, thus the need for the 'wallclock' restriction is not obligatory. However, the introduction of the 'wallclock' feature adds a useful **'Host Utilization'** output result. This line indicates how heavily the host processor was used by the simulation process.

The knowledge we obtained out of the runs, is that we now have an idea about the effects of adding the intercept library on the platform. Furthermore, we have seen 'Real Time Ratio' values from x12 to x35 meaning that the simulator runs exceptionally well, providing much faster simulations than the actual hardware. We also see that the simulator is actually running quite well under load, because the TOTAL MIPS numbers are not so far from each other. **However, from inside the UART console, the delay simulation effects are much more obvious and require a lot of waiting**.

On Figure14, we have the outputs of the intercept library that were printed for the previously delayed (Figure13-right) simulation. These outputs are useful as debug signals, but also provide good information about the instructions that were issued, skipped and also a cross-check with the verbose output results. We can view the PC, the number of issued instructions(should really be considered a cycle count), the number of skipped instructions and the number of instructions that were actually executed after all these skips, calculated by the subtraction of the two previous metrics, for every processor on the simulated platform. Furthermore, for each processor, I used vmirtGetExecutedICount VMI Run Time function to demonstrate the number of instruction that were actually executed. This number should be close to the previous estimation, but identical to the one from the verbose output for each processor respectively. All these metrics come from the VMI Run Time functions described in the previous section. **Instead, I also included an output that gives the total number of read and write callbacks, thus estimating the total system delay.** These numbers come from counter variables inside the intercept library that escalate each time read or write callback function is activated. The total system delay is calculated using these callback numbers, along with the network delay set for the simulated device.

5.3 Best effort

With the platform now completed, we knew that both the simulator and the platform were running great, because we could see a perfect implementation and execution of a Linux Operating System on our simulated design. As we had the OS up and running, we thought about trying to execute some simulating benchmarks that would run inside the OS, and thus on the simulated platform. We chose to work with benchmarks that were capable of providing a good indication about the functionality of our design on a number of different disaggregated memory and network delay configuration tests.

As a first attempt, we decided to approach the famous **Cloud suite**. Cloud Suite is a benchmark suite for cloud services. The third release consists of eight applications that have been selected based on their popularity in today's data centers. The benchmarks are based on real-world software stacks and represent real-world setups. This seemed like an ideal benchmark suite. Unfortunately, we ran into major issues installing **Docker** on the simulated OS. This was a blow, because in order to run the tests that Cloud Suite provided we needed Docker to be installed on our platform. Although I was not able at installing it with my first attempt, it seemed possible. I didn't have any prior experience with Docker so this might have been the problem or we should have better configured the rootfs image or the Kernel Image to suit the Docker needs. It is possible to do that and it is a great idea and work for the future as it might be useful for not only running the Cloud Suite benchmarks but other cloud applications as well.

After working with the Cloud Suite, we turned our attention to the **KeyServer** benchmark. Suggested to us by our colleagues, the KeyServer benchmark is part of the mami-project (23). This software is a Key Server that implements the TLS Session Key Interface (SKI)(24). The TLS Session Key Interface (SKI) defined the mentioned document makes it possible to store private keys in a highly trusted key server, physically separated from client facing servers. The TLS server is split into two distinct entities called Edge Server and Key Server that communicate over an encrypted and mutually authenticated channel using e.g. TLS. This software implements the Key Server entity.

Unfortunately, we ran into a lot of problems during this installation. We even had to tackle a lot of minor problems during the prerequisites installation but after some good thinking and knowledge, we have managed to correctly configure everything. The next step was to install the **Redis database**, which we succeeded after a lot of trial and error. The main problem was that the jemalloc did not work properly so we used the 'MALLOC=libc make' to install it. And that is where we have stopped. Although Redis was up and running inside the simulated OS, I was never capable of running the 'make test' or the 'Redis-client' example to verify if everything worked fine. Thus, all this work was halted because we could not evaluate the Redis installation. I found a number of suggested solutions and I am sure that in the near future I will be able to set everything correctly and run this benchmark suite. For now, only the preliminary work has been completed.

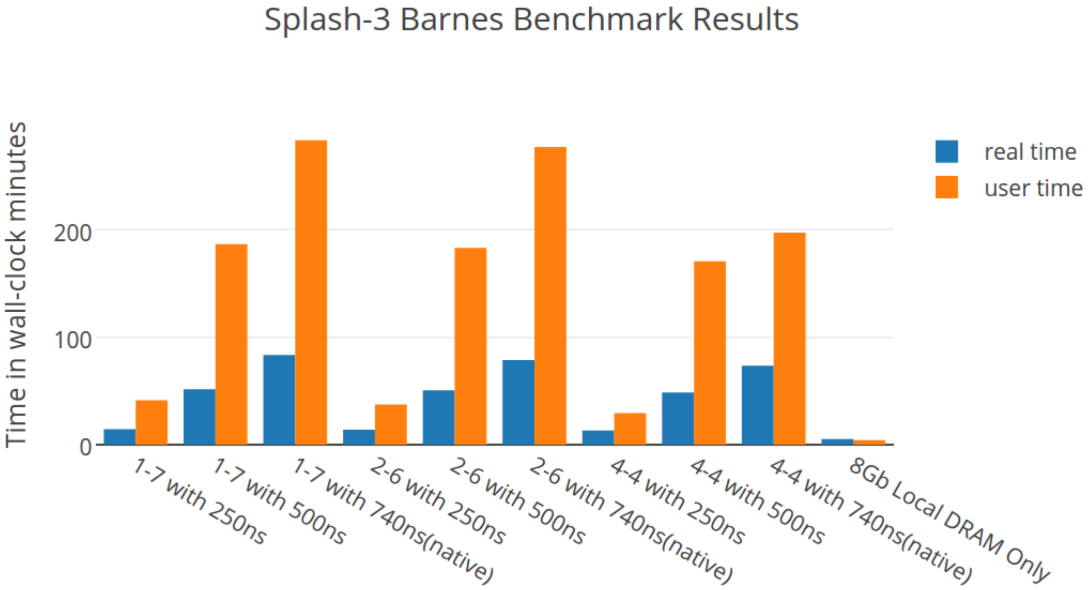
5.4 Splash-3 [Barnes]

Splash-3 (20) is a benchmark suite based on Splash-2 but without data races. It was first presented at ISPASS'16 [SLKR16]. The current Splash-3 version, referred to as Splash-3x, contains some small changes that enable support for running with a higher number of threads. From this benchmark suite, we decided to run the **Barnes** application. There is no need to install Splash-3. All you need to do is set the **BASEDIR** variable in codes/Makefile.config to point to the full path of the codes directory, and then, initiate 'make' in order to create the benchmark's applications.

The recommended inputs given inside the Splash-3 installation were not what we wanted. The input file's 'nbody' value was very small so I decided to search for a different value to incorporate as an input. After some research, I found that the **parsec-3.0** benchmark suite provided small, medium and large simulation inputs for the Barnes application, and I selected to use the 'simLarge' input values. So we ended up with the following input values: **nbody=262144, dtime=0.025, eps=0.05, tol=1.0, dtout=0.25, tstop=0.075, fcells=2.0, NPROC=4**, written inside one of the given input files.

The BARNES application implements the 'Barnes-Hut' method to simulate the interaction of a system of bodies (N-body problem). 'Barnes-Hut' simulation is an approximation algorithm for the execution of an n-body simulation. Using the 'Barnes-Hut' we get a complexity O(nlogn) compared to the 'direct-sum' algorithm complexity of O(n^2). The N-body problem in physics is the problem of predicting the individual movement of a group of celestial objects which interact gravitationally.

The reason behind this benchmark execution was to see if our simulated platform exhibited different behavior with different disaggregated memory and network delay configurations when running the same benchmark application. So, we improvised a series of combinations between remote and local DRAM memory configurations with altering network delay values between each test. We ended up with 10 tests, each with a different configuration. We tested the 'local-remote' memory combinations of 1Gb-7Gb, 2Gb-6Gb, and 4Gb-4Gb respectively with network delays altering between 250ns, 500ns and the native estimation of 740ns. We also issued a test with only 8Gb of local DRAM. The real time and user time results from these test runs are clearly visible in the following Figure.



Let us discuss these results. The first thing we notice is an intriguing difference between the user time and the real time. The user time is normally smaller than the real time. That is not the case in this series of tests except for the 8Gb local DRAM test. We know that the real time represents the wall-clock time from start to finish of a function/application call. The user time represents the time that the CPU spent outside the Kernel for the same function/application call. That is the reason why the user time is greater than the real one. **While the real time only shows how much time it would have taken to run this benchmark on this platform, the user time also depicts all the time the CPU or CPUs spent executing the benchmark alongside the simulation process and the simulation of the delay for the system interconnect**. The user time is analog to the total simulated delay. In other words, it increases due to the complexity of simulating all these Skip-Counts that were previously mentioned. **So, the user time for these tests can better be described using the term 'actual-wall-clock-time'**

Apart from that, an interesting pattern emerges. After running all 10 tests, the real time and also the user time values suggest that the execution time increases as we increase the delay interconnect, but decreases for memory configurations that have more local memory. This is exactly what we needed to see. This pattern suggests that our implementation of the simulated delay works great and we can create platforms that represent an actual 'local-remote' DRAM memory configuration. This verifies that all our work was on a correct path and that this disaggregated platform can be used for running more advanced benchmarks (like Cloud-Suite or Keyserver), especially for data centers. Also, it is a good baseline for future virtual platform development in simulating new disaggregated platform ideas.

However, we have to discuss the efficiency of this platform on running this benchmark application. It is clearly visible that running this application on the disaggregated platform configurations takes more time as regards to running on a mainboard with only local DRAM. The real simulation time is the best for the 8Gb-local-DRAM-only configuration with 5.31 minutes, followed by the 4Gb-4Gb-local-remote(250ns) with 13.15 minutes, the 2Gb-6Gb-local-remote(250ns) with 14.01 minutes and the 1Gb-7Gb-local-remote(250ns) with 14.51 minutes. When increasing the network delay, these numbers climb even higher. On the contrary .

These results are far from alarming. On the contrary, they are extremely useful. From this graph it also is clearly visible that the use of remote DRAM alone is the key factor of this execution time spike. For the approximated 500ns delay, the execution time between the test configurations is negligible, indicating that the use of remote DRAM alone, means that the time statistics climb. So, for future systems that implement remote memory allocation, we could run a series of tests and evaluate if this increase in execution time is acceptable and can provide a better overall working database design in the future.

Despite this application's slower execution time on the disaggregated platform, we can only suggest that it is not ideal for this particular set of tests. Another application might present different simulation statistics and this gives us a good footing for future work. Furthermore, the DRAM pool that we tested was limited. The simulation results could be different if we had different local and remote memory configurations. All in all, these test runs are a good baseline for the future platform testing and evaluation. **As a matter of fact, we think that for a simulated platform running a simulated OS on a full system simulator, the results for the disaggregated memory tests with 250ns (best case scenario) delay were acceptable as they were less than 3-times slower than the mainboard configuration**. As I said, more testing is highly recommended.

6 Conclusions and Future work

6.1 Contributions and lessons learned

It's a GOOD start! It is a successfully implemented portion of the idea of the dRedBox disaggregated database platform on a virtual server blade platform. In this work, it is also highlighted that this certain approach for modeling the disaggregated memory configurations as well as the simulated delays is applicable and functional on this simulator.

It was a very time-consuming project that had a lot of hidden steps between the implementation of even the simplest of tasks. It has to be stated that this work took almost a whole year to complete, in which knowledge obtained was progressing along with the workload completed. The most lengthened part of this thesis work was the process of extracting functionality information about the components and the platform capabilities of the Imperas simulator. But, as the information kept stacking up, more and more functionality became visible and of course applicable. The end result was the first step in merging the dRedBox disaggregated data center project with Imperas virtual platform simulation. This was the main task from the beginning, but it ended up being so much more.

Besides all the lessons learned along the way, this was a successful simulation of a new virtual platform modeling idea. As we mentioned, the first part was to create the platform on the simulator. This was not the upper limit. However, as the work kept advancing, more and more ideas and of course obstacles came along the way. The difference is that now, an initial foothold has already been established and a certain amount of knowledge is acquainted with this kind of virtualization work.

Working on a virtual platform that has a fully functional OS is a big step for evaluation, analysis and debugging for future testers. With all our knowledge now available, more efficiency can be achieved for this particular field of work. In addition, it came to our attention that the Imperas simulator, and virtual platforms in general, can provide even more functionality, performance and evaluation tools. Now that this work is completed, it is possible to build from this point and with all the known features extract more from this simulator platform. For example, the simulator has the capability of running even faster through Quantum Leap (19), and that is a big step up if we decide to run even more complicated platform design or simulations.

All in all, this work provides a great baseline for future evaluation and testing and also has a variety of fields that can be studied and explored to make this initial model (or a new one) even better, or as close as possible to a real life prototype.

6.2 Is it worth it?

The question is then: is it worth it? Of course! With this approach, we enter the world of virtual platform emulation which is the fastest and most efficient way in the embedded development industry. It is the best way to make all the necessary evaluation tests a lot quicker and with highly accurate results in order to see if the product is worth putting in production. But, as I said, only the first step was made and this approach has way more capabilities.

Aside from that, we have an excellent behaving, well designed and highly efficient end result. It is great for evaluation testing, multi-core capable, it supports disaggregated platform designs and it is highly scalable. Its speed and efficiency are no new news and the performance has not yet reached its peak. It is also a technology that is advancing and upgrading rapidly so that means that more and more aspects of this will become better in the process. Also, we are now familiar with the virtual platform tools and we can use them not only with the dRedBox project but also in all sorts of other embedded system design platforms, that may or may not be related with dRedBox.

6.3 Plans for the future

As I mentioned earlier, there is a lot more room for improvement. The virtual platforms are a technology upgrading and so, new, better and faster simulations will be available really soon. Also, the current platform is capable of delivering more functionality, but as regards of my diploma thesis work, we didn't use all of it. There is the possibility that the Imperas-OVP platform is capable of simulating a multi-platform-multi-processor-multi-OS design that is as close as it gets to simulating the rack scale design of the dRedBox vision. Also, this simulator can be installed in a much faster host machine and-or on a 64bit OS to improve its performance even more.

Apart from that, the OS ready system can be expanded, including more capabilities that make it as close as possible to the real hardware design and behavior. On that, the user can then run all sorts of new or old simulations and benchmarks with the highest accuracy. Furthermore, a good idea is to apply all the VAP (Verification Analysis and Profiling) tools that Imperas M*SDK provides to get an even better idea of the efficiency of our design. Also, the simulator can be tuned in order for us to get results from simulations that have been executed with all the processor, memory and network delay combinations that are possible.

An approach that we could do in a different way would be as regards to the memory modules used. Imperas might provide, in the future, different DRAM memory modules or other kinds of memory. This instantly means that these memories can be implemented on this prototype virtual design and used to simulate the memory disaggregation with a different approach as the one using the intercept library with the approximation of skip-counts.

One thing that it could really boost this approach would be to implement some sort of energy efficiency approximation algorithm that extracts viable information from the simulated platform and translates them to energy consumption values. Lastly, we could use a finer tuned and complete rootfs OS image in order to have our platforms running at full capacity with minimum resource capability inefficiencies. A good example would be to set the image accordingly to set the most used cloud related benchmarks from Cloud-Suite!

THE FUTURE IS HERE AND ITS BRIGHT....

Appendix A: Imperas simulator tips and tricks

- **Always end an OS simulation by executing 'halt' from inside the UART console before closing it, otherwise, you can damage the Kernel image file. Afterward, terminate the UART and view the results printed on the host's Linux terminal.**

- **MIPS**: The parameter MIPS is used to set the instruction frequency of a particular processor instance. The specified parameter value sets the number of million instructions executed per second. OVP processor models have a default speed of 100 MIPS.

- **Wallclock**: We have seen examples of processors with a nominal speed of 100 MIPS that could be made to run at over their nominal MIPS (i.e 250). Although it is usually a benefit to having better-than-real-time simulation performance, there are some occasions when this is undesirable. For example, when simulating an OS such as Linux, processors are almost entirely idle when waiting at a login prompt. Unless told otherwise, the simulator will move simulated time rapidly forward when processors are idling. The effect of this is that it is impossible to log in interactively to the simulated Linux because of the log in times out instantly as simulated time shoots forward. It is possible to restrict maximum performance to any multiple of the real-time clock using the function icmSetWallClockFactor: **void icmSetWallClockFactor(double factor);**.
  Otherwise, you can activate the wallclock feature just by adding {–wallclock} when issuing an application run or an OS simulation. Also, you can add {–wallclockfactor <decimal>} in order to declare how much faster you allow the simulator to run as regards to the nominal MIPS! As you can see by now, –wallclock is the same as –wallclockfactor 1! For more information view the dedicated Imperas **OVP_Control_File_User_Guide.pdf** guide.

- A good way to start after you have finished installing the simulator, is by accessing and reading all the Imperas user guides located inside:
  $IMPERAS_HOME/doc/imperas/index.html

- Every new version of the Imperas simulator can be activated by each valid academic license that you have been using for the previous version. But, you have to keep in mind that most of your platforms may need rebuilding as there are many changes between patches.

- A useful feature for modifying and adding to the content of the rootfs images is to mount them and manage their directories directly, like a common folder. To do that, follow these steps:

  1. you have to have qemu and nbd-client installed.
  2. load the module: sudo modprobe nbd max_part=8
  3. share the disk on the network and create the device entries:
     sudo qemu-nbd –connect=/dev/nbd0 ubuntu16.04arm64.img
  4. mount the image:
     mkdir qemu_img
     sudo mount /dev/nbd0p1 qemu_img
     sudo chown -R $USER: $HOME

5. unmount when done:
   sudo umount qemu_img
   sudo nbd-client -d /dev/nbd0 { it shows : "disconnect, sock, done"}

- Some basic packages to install inside your OS image to help you set up and run all the benchmarks that you like are: **curl, gcc, git-core, nano, build-essential, wget, software-properties-common, lxc, aufs-tools, cgroup-lite, apparmor, lshw, tcl, linux-headers-generic**. Also, run a sudo apt-get update and upgrade before and after all the installations are finished.

- A very good idea is to add all the Imperas environment variables inside the .profile script so that you do not have to export them each time you restart your host machine. All, except the IMPERAS_TOOLS variable that manages the intercept library. This one is better to export every time you open a new terminal in order to avoid the use of this library if not necessary.

- If you want to improve the efficiency of the simulator running on your PC visit the Imperas website or contact them about the QUANTUM LEAP(19)!

- MIPS: You set a nominal MIPS number when a processor model is implemented. The verbose MIPS output provides information to indicate how many instructions were executed on the OVP Processor model and the equivalent simulated Millions of Instructions Per Second (MIPS) that were executed. This can be used as an indication of how well the simulator performed on the host machine running a particular application.

- The **VB_DRIVE_DELTA** option on the Virtio block device can be set to '1' to prevent the model from modifying the image on the host disk drive. If this option is set to '0', the image file will be modified by the simulation. **When VB_DRIVE_DELTA is set to '0', you must be careful to shut Linux down cleanly with the "halt" command to prevent corruption of the disk image. Once the kernel halts just close the uart0 window and the simulation will terminate**.

- The simulated CPU can also be changed from inside the device tree. As we said, the frequency inside the device tree must match the MIPS set inside the model declaration file, which in my case was the module.op.tcl file. Note that if we change the MIPS value, the simulated delay value will also change, because it is calculated using the MIPS at a given moment!

- Each time we open a new terminal, we should point the IMPERAS_TOOLS environment variable to the location of the lib.ic file, in order to activate the binary interception. The simulator requires that the lib.ic and also the library's model.so must be located in the same directory as the RUN.sh script.

- To enable the M*SDK feature of the VAP tools, just add the following line to the starting script:
  –enabletools –symbolfile ../Linux/vmlinux –extlib cpu=linux64OsHelper –override ${MODULE}/cpu/enableSMPTools=1

- From the 'RUN.sh' script we can also use other CPU models like the 53 and the 72 with all possible x1,x2,x3 and x4 combinations.

- **Internet access**, using the virtual bridge virbr0 on the host machine:
  -Repeat the following commands every time you restart the PC-

  ```
  sudo brctl addif virbr0 enp3s0
  sudo dhclient virbr0
  sudo service network-manager restart
  ```

  Finally, inside the UART console, execute:
  ifconfig eth0 up and dhclient.

Appendix B: Imperas Small Starting Guide

More information that compliments this small guide is given in the extensive and very detailed Imperas Installation guides.

- **OS**: Aside from all the recommended Operating Systems, all this thesis work has been implemented on an Ubuntu Linux 32bit OS. Using a 32bit OS will make it easier to work with as all the model, peripheral and tool-chain installers from both the Imperas and OVP websites come as 32-bit executable files. However, there are libraries that make it possible to install 32-bit on a 64-bit Linux and Imperas assures that this approach works fine.

- **The license activation**: Imperas provides a certain **Academic License** for research institutes and facilities. The OVP and Imperas products are licensed using FLEXlm. Running the Imperas license server within a virtual machine is not permitted. The IMPERASD_LICENSE_FILE environment variable should be set to point to the license server. For this work, a floating license was used, described in the Imperas installation guide. There are also other ways to set up the licensing that can be found in the installation tutorials.

- **Installation**: The Linux versions are provided as pre-built binaries in a self-extracting executable file. Imperas products are developed to work on x86 hardware, 32-bit or 64-bit. Also, GCC compilers versions on all available OS versions are used by Imperas for generation of host code. Imperas SDK as Imperas_SDK.<version>.<dot release>.Linux32 .exe. As I said, Linux 64-bit hosts are also supported by Imperas. First, we need to change the installer first so, for example, we execute:

  chmod +x Imperas_ SDK.20161005.0.Linux32.exe

  Then, we execute the self-extracting installer and accept the license agreement that pops:

  ./Imperas_ SDK.20161005.0.Linux32.exe

  Once the license has been accepted, you will be asked to provide the directory in which you want to install the Imperas tools.

- **Set up the Linux Environment Variables**: If for example, the installation directory is: $HOME/Desktop/imperas.20161005, then you can add the following to the .profile script:
  export IMPERAS_HOME = $HOME/Desktop/imperas.20161005
  export IMPERAS_ARCH = Linux32
  export IMPERAS_UNAME = Linux
  export IMPERAS_SHRSUF = so
  export IMPERAS_VLNV= $IMPERAS_HOME/lib/$IMPERAS_ARCH /ImperasLib
  export IMPERAS_RUNTIME = CpuManager
  export PATH = $PATH $IMPERAS_HOME/bin/$IMPERAS_ARCH
  export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:$IMPERAS_HOME /bin $IMPERAS_ARCH:$IMPERAS_HOME/lib/$IMPERAS_ARCH /External/lib
  export IMPERASD_LICENSE_FILE = <server_port>@<cite>.gr
  Keep in mind that the Windows installation follows a different procedure and requires the use of MSYS/MinGW Environment.

- **Installing additional tools**: Many tools are provided inside the Linux or Windows installers. More are available from the OVP website www.OVPworld.org. It is very important to state that the Cross Compiler and Peripheral Simulation Engine toolchains are provided only as 32-bit native executables. **It is crucial that the installed tools are for the exact version of the installed simulator that we are using**. Each additional tool from the Imperas-OVP library is marked and has its own version that highlights this compatibility. Also, each tool must be installed inside the same folder that the simulator is located. Both PSE and Cross Compiler Toolchains have a dedicated Makefile in order to build PSE and processor application behavioral code respectively. The installation of the appropriate Cross Compiler Toolchains and Peripheral Simulation Engine (PSE) Toolchain comes first, followed by all the packages for testing or evaluating that can be used as a guide and also as a baseline for many new designs.

- **Installation Check**: To verify the installation after an OVPsim or an Imperas install one of the Demos that are downloaded in both installations can be executed. These Demos will work on both the OVPsim and the Imperas Simulators. A good example is to execute the 'simpleCpuMemory' platform that is located inside $IMPERAS_HOME/Examples /PlatformConstruction/simpleCpuMemory. This will show if the simulator is working correctly and it will introduce you into platform construction, the harness and the application execution on the simulated platform.

Appendix C: The Simulated Platform

- Whenever we start a new design using a demo as a baseline, it is a good idea to create a new directory inside the IMPERAS_HOME folder and copy all the files that are included in the demo directory. This is helpful so that the initial demo directory remains intact as we configure the test/copied one.

- **Setting the Platform parameters**:

  ```
  set vendor arm.ovpworld.org
  set library module
  set name ARMv8-A-FMv1
  set version 1.0
  ```

- **Initiate Platform Creation**: I utilize the parameters I set before and also initiate the platform creation. 'ihnew' is a TCL command that initiates the platform design.

  ```
  ihwnew -name $name -vendor $vendor -library $library -version $version -stoponctrlc -purpose module -releasestatus ovp
  iadddocumentation -name Licensing -text "Open Source Apache 2.0"
  iadddocumentation -name Description -text $desc
  iadddocumentation -name Limitations -text $limitations
  iadddocumentation -name Reference -text "ARM DUI 0677C"
  ```

- **The Common Bus**: The 40bit address value was taken directly from the Foundation v8 platform model.

  ```
  ihwaddbus -instancename pBus -addresswidth 40
  ```

- **Platform processor**: Here we have the complete declaration of the platform CPU.

  ```
  ihwaddprocessor -instancename cpu -type arm -vendor arm.ovpworld.org -version 1.0 -endian little -simulateexceptions -mips 500
  ihwconnect -instancename cpu -bus pBus -busmasterport INSTRUCTION
  ihwconnect -instancename cpu -bus pBus -busmasterport DATA
  ihwsetparameter -handle cpu -name variant -value Cortex-A57MPx4 -type enum
  ihwsetparameter -handle cpu -name compatibility -value ISA -type enum
  ihwsetparameter -handle cpu -name UAL -value 1 -type boolean
  ihwsetparameter -handle cpu -name override_CBAR -value 0x2c000000 -type Uns32
  ihwsetparameter -handle cpu -name override_GICD_TYPER_ITLines -value 4 -type Uns32
  ```

  After setting the 'version', the 'endian', activating the 'simulateexeptions' and placing the nominal mips at 500, we first connect the 'DATA' and 'INSTRUCTION' master ports to the pBus. All the 'ihwsetparameter' commands are used to configure the arm processor module. Here I set as default the Cortex-A57MPx4 processor but I can change that outside this file as I will demonstrate in later sections. All the other configurations are made using the dedicated Imperas guide, and set the ISA compatiblility, the CIGs and other stuff as well.

  We will also use the 'override_timerScaleFactor' parameter:

  ```
  ihwsetparameter -handle cpu -name override_timerScaleFactor -value 5 -type Uns32
  ```

  Here the value is set to 5 thus it sets the processors Generic Timers to

100 MHz (500MIPS/5), to match the specified speed in the device tree (cpu)! The Kernel is not checking CNTFRQ register value so you must match what is in the device tree!

- **CPU Interrupt ports:** After initializing the processor we have to connect the supported interrupt request nets to the processor's Shared Peripheral Interrupt ports. In this allocation of the Interrupt ports, we select which to use and which to remain idle.

  ```
  # unused { 2 3 4 9 10 12 13 14 }
  foreach i { 5 6 7 8 15 42 } {
  ihwaddnet -instancename ir${i}
  ihwconnect -net ir${i} -instancename cpu -netport SPI[expr $i + 32] }
  ```

- **ARMv8 Foundation Model #3 RAM modules:** These 3 ram modules are connected to the pBus to the addresses: 0x00000000->0x03ffffff, 0x04000000->0x0403ffff and 0x06000000->0x07ffffff. 'ihwaddmemory' command, adds the ram module and the 'ihwconnect' links it to the common pBus. One of these 3 connection is shown below:

  ```
  # 0x00000000 0x03ffffff RAM
  ihwaddmemory -instancename RAM0 -type ram
  ihwconnect -instancename RAM0 -bus pBus -busslaveport sp1 -loaddress 0x00000000 -hiaddress 0x03ffffff
  ```

- **The Ethernet module:** Imperas provides the model of the SMSC 91C111 Ethernet device model. Location: $IMPERAS_HOME/ImpeasLib/source/smsc.ovpworld.org /peripheral/LAN91C111. The implementation is described with the tcl code that follows. One thing to notice is that this is connected to the common pbus and also to the interupt port ir15.

  ```
  ihwaddperipheral -instancename eth0 -type LAN91C111 -vendor smsc.ovpworld.org
  ihwconnect -instancename eth0 -bus pBus -busslaveport bport1 -loaddress 0x1a000000 -hiaddress 0x1a000fff
  ihwconnect -instancename eth0 -net ir15 -netport irq
  ```

- **System registers:** Suggested by the ARMv8 Foundation model. Location: $IMPERAS_HOME/ImpeasLib/source/arm.ovpworld.org/peripheral /VexpressSysRegs

  ```
  ihwaddperipheral -instancename sysRegs -type VexpressSysRegs -vendor arm.ovpworld.org
  ihwconnect -instancename sysRegs -bus pBus -busslaveport bport1 -loaddress 0x1c010000 -hiaddress 0x1c010fff
  ihwsetparameter -handle sysRegs -name SYS_PROCID0 -value 0x14000237 -type Uns32
  ```

- **UART:** ARMv8 Foundation model suggests 4 but for this thesis, we used only one. They are the interface to the actual hardware by providing the user with a Linux terminal console after loading an OS image. Location: $IMPERAS_HOME/ImpeasLib /source/arm.ovpworld.org/peripheral/UartPL011 . They have to be connected to an interrupt port (i.e ir5,6,..). In addition, their parameters 'variant', 'outfile' and 'finishOnDisconnect' should be set to 'ARM', 'uart0.log' and '1' respectively.

```
# 0x1c090000 0x1c090fff UartPL011 pl011
ihwaddperipheral -instancename uart0 -type UartPL011 -vendor arm.ovpworld.org
ihwconnect -instancename uart0 -bus pBus -busslaveport bport1 -loaddress 0x1c090000 -hiaddress 0x1c090fff
ihwconnect -instancename uart0 -net ir5 -netport irq
ihwsetparameter -handle uart0 -name variant -value ARM -type string
ihwsetparameter -handle uart0 -name outfile -value uart0.log -type string
ihwsetparameter -handle uart0 -name finishOnDisconnect -value 1 -type boolean
```

- **The Virtio block device:** Location: $IMPERAS_HOME/ImpeasLib /source/ovpworld.org/peripheral /VirtioBlkMMIO.

  ```
  ihwaddperipheral -instancename vbd0 -type VirtioBlkMMIO -vendor ovpworld.org
  ihwconnect -instancename vbd0 -bus pBus -busmasterport dma -loaddress 0 -hiaddress 0xffffffffff
  ihwconnect -instancename vbd0 -bus pBus -busslaveport bport1 -loaddress 0x1c130000 -hiaddress 0x1c1301ff
  ihwconnect -instancename vbd0 -net ir42 -netport Interrupt
  ```

- **The SmartLoaderArm64Linux:** Location: $IMPERAS_HOME/ImpeasLib/source /arm.ovpworld.org/peripheral/SmartLoaderArm64Linux. One thing to note is that the physical base address starting point must be set to 0x80000000.

  ```
  ihwaddperipheral -instancename smartLoader -type SmartLoaderArm64Linux -vendor arm.ovpworld.org
  ihwconnect -instancename smartLoader -bus pBus -busmasterport mport -loaddress 0 -hiaddress 0xffffffff
  ihwsetparameter -handle smartLoader -name physicalbase -type Uns64 -value 0x80000000
  ihwsetparameter -handle smartLoader -name command -type string -value "console=ttyAMA0 earlyprintk=pl011,0x1c090000 nokaslr"
  ```

- **The DRAM modules:** As an example a (2GB - 6GB) configuration is presented in the thesis. At that point, the memory is uniform. The local and remote attributes are given when we configure the network interconnect delay for this platform. For example, we can add a certain amout of delay when accessing the 6GB memory block, thus making it work like a remote memory module. This delay feature can be inserted via an intercet library that is described in a separate chapter.

- **The module Makefile:** We only have to focus on the one inside the module folder. This Makefile has the task of including a designated Makefile that is capable of creating this module platform. and is located at: $IMPERAS_HOME/ImpeasLib /buildutils/Makefile.module.

Appendix D: The Intercept Library

- **vmirtAddSkipCount**: VMI API Run Time function. Full definition at : OVP_VMI_Run_Time_Function_Reference.pdf. The full declaration: **void vmirtAddSkipCount(vmiProcessorP processor, Uns64 skipCount);**

- For simulating the network delay, the BinaryInterception Demo, located in the Imperas installation, is a great baseline. Also, the intercept library is written in C.

- **Includes**:
  ```
  #include "vmi/vmiMessage.h"
  #include "vmi/vmiOSAttrs.h"
  #include "vmi/vmiRt.h"
  #include "vmi/vmiVersion.h"
  #include "stdio.h"
  ```

- **Structure type containing local data for this plugin**:
  ```
  typedef struct vmiosObjectS {
  Uns32 totalReads;
  Uns32 totalReadBytes;
  Uns32 totalWrites;
  Uns32 totalWriteBytes;
  } vmiosObject;
  ```

- **Constructor**: With these callbacks, we can simulate the interconnect delay and create the remote DRAM effect.
  ```
  static VMIOS_CONSTRUCTOR_FN(constructor) {
  memDomainP domain = vmirtGetProcessorPhysicalDataDomain(processor);
  vmirtAddReadCallback(domain, 0, 0x0880000000, 0x09ffffffff, readCB, object);
  vmirtAddWriteCallback(domain, 0, 0x0880000000, 0x09ffffffff, writeCB, object);}
  ```

- **Destructor**:
  ```
  static VMIOS_DESTRUCTOR_FN(destructor) {
  if(flag==0)
  {
  printf(\nread callbacks %lld, write callbacks %lld and total system delay %lld ns\n\n",r,w,((r+w)*740) );
  flag=1;
  }
  vmiPrintf(" processor's %s final program counter : PC->"FMT_64u" \n", vmirtProcessorName(processor), vmirtGetPC(processor));
  vmiPrintf(" issued instructions: "FMT_64u" \n", vmirtGetICount(processor));
  vmiPrintf(" skipped instructions: "FMT_64u" \n", vmirtGetSkipCount(processor));
  vmiPrintf(" Calculated executed instructions estimation: "FMT_64u" \n\n", (vmirtGetICount(processor)-vmirtGetSkipCount(processor))); vmiPrintf(" Actually executed instructions based on vmirtGetExecutedICount function: "FMT_64u" \n", vmirtGetExecutedICount(processor));
  }
  ```

- **Read Callback**:
  ```
  static VMI_MEM_WATCH_FN(readCB) {
  if(processor) {
  vmirtAddSkipCount(processor,74);}
  }
  ```

- **Write Callback**:
  ```
  static VMI_MEM_WATCH_FN(writeCB) {
  if(processor) {
  vmirtAddSkipCount(processor,74);}
  }
  ```

- **Makefile.intercept**:
  After all this is done, the intercept library is completed. Now we only need to copy the **Makefile.intercept** from the BinaryIntercept example inside our platform directory. The Makefile.intercept includes the **$(IMPERAS_HOME)/ImperasLib/buildutils /Makefile.host** provided by Imperas.

Definitions and Useful links

1. The ramification is a complex or unwelcome consequence of an action or event.

2. Horizontal and Vertical Architecture: Horizontal scaling means that you scale by adding more machines into your pool of resources whereas Vertical scaling means that you scale by adding more power (CPU, RAM) to an existing machine.

3. `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ipmi-second-gen-interface-spec-v2-rev1-1.pdf`

4. A Serializer/Deserializer (SerDes pronounced sir-deez) is a pair of functional blocks commonly used in high-speed communications to compensate for limited input/output. These blocks convert data between serial data and parallel interfaces in each direction. The term "SerDes" generically refers to interfaces used in various technologies and applications. The primary use of a SerDes is to provide data transmission over a single/differential line in order to minimize the number of I/O pins and interconnects.

5. `http://www.linux-kvm.org`

6. In computing, paravirtualization is a virtualization technique that presents a software interface to virtual machines that are similar, but not identical to that of the underlying hardware

7. `http://searchservervirtualization.techtarget.com/definition/memory-ballooning`
   Virtual memory ballooning is a computer memory reclamation technique used by a hypervisor to allow the physical host system to retrieve unused memory from certain guest virtual machines (VMs) and share it with others. Memory ballooning allows the total amount of RAM required by guest VMs to exceed the amount of physical RAM available on the host. When the host system runs low on physical RAM resources, memory ballooning allocates it selectively to VMs.

8. `http://www.imperas.com/msdk-advanced-multicore-software-development-kit`

9. `http://www.imperas.com/dev-virtual-platform-development-and-simulation`

10. `http://www.ovpworld.org/welcome`

11. `http://www.ovpworld.org/technology`

12. `http://www.imperas.com/articles/imperas%E2%84%A2-delivers-next-generation-embedded-so`

13. `http://infocenter.arm.com/help/topic/com.arm.doc.dui0677c/DUI0677C_foundation_fast_model_ug.pdf`

14. More information about the Imperas VMI API can be found inside the simulator installation located inside:
    $IMPERAS_HOME/doc/api/vmi/html/index.html

15. **bootstrap**:
    In general, **bootstrapping** usually refers to a self-starting process that is supposed to proceed without external input.
    Furthermore, **bootstrapping** in computer science is the process of writing a compiler (or assembler) in the source programming language that it intends to compile. Applying this technique leads to a self-hosting compiler. An initial minimal core

version of the compiler is generated in a different language (which could be assembly language); from that point, successive expanded versions of the compiler are run using the minimal core of the language.
Moreover, **bootstrapping** in computer technology (usually shortened to booting) usually refers to the process of loading the basic software into the memory of a computer after power-on or general reset, especially the operating system which will then take care of loading other software as needed.
But also, **Bootstrap** is a free and open-source front-end web framework for designing websites and web applications.

16. **KVM hypervisor**:
    http://whatis.techtarget.com/definition/KVM-hypervisor
    KVM hypervisor is the virtualization layer in Kernel-based Virtual Machine (KVM), a free, open source virtualization architecture for Linux distributions. A hypervisor is a program that allows multiple operating systems to share a single hardware host.

17. **Clos network**:
    In the field of telecommunications, a Clos network is a kind of multistage circuit switching network, which represents a theoretical idealization of practical multi-stage telephone switching systems. In general, Clos networks are required when the physical circuit switching needs to exceed the capacity of the largest feasible single crossbar switch. The key advantage of Clos networks is that the number of crosspoints (which make up each crossbar switch) required can be far fewer than would be the case if the entire switching system were implemented with one large crossbar switch. Nowadays, the advent of complex data centers, with huge interconnect structures, each based on optical fiber links, means that Clos networks are again important.

18. **Network simulated delay values**: bidirectional (glue logic + switching delay) 60ns + 300ns SerDes pipeline delay + network delay (with one switching layer it is approximately 20ns) + 300ns SerDes pipeline at reception + 20ns switching + 40ns memaccess (AXI delay for remote mem included previously) = 740ns.
    This means that a good baseline is to try and reference this 740ns delay into a SkipCount value to add as a parameter in the vmirtAddSkipCount function. **If we have the nominal MIPS=100 (clock=100MHz) and also WALLCLOCK function activated we know that the simulator is capable of delivering 10ˆ8 instructions per second. That means that it can deliver 0.1 instructions per 1ns. So in the case of the remote memory accesses (reads or writes) the simulator is capable of executing 0.1*740=74 instructions. So we decided to set the SkipCount value at 74, to simulate our delay effect**. Of course, if we change the MIPS, we also have to change the delay that we are going to use!!!
    It is worth noting that I tried setting the MIPS=500 and also the CPU clock at 500MHz from inside the device tree. Frankly, the simulated delay value became 370. The result was that the simulator could not boot the OS in all of my tries so I stuck with the default 100MHz design!!!

19. http://www.imperas.com/articles/imperas-delivers-quantumleap-simulation-synchronizatio
    E2%80%93-industrys-first-parallel-virtual

20. https://github.com/SakalisC/Splash-3

21. https://community.arm.com/processors/b/blog/posts/programmable-interrupt-controllers-a

22. https://www.ibm.com/developerworks/library/l-virtio/

23. https://github.com/mami-project/KeyServer/wiki

24. https://tools.ietf.org/html/draft-cairns-tls-session-key-interface-01

References

[Clo53] Charles Clos. A study of non-blocking switching networks. *Bell Labs Technical Journal*, 32(2):406–424, 1953.

[HAK$^+$14] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. Simulating dram controllers for future system architecture exploration. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 201–210. IEEE, 2014.

[Has14] Kobi Hasharoni. High bw parallel optical interconnects. In *Photonics in Switching*, pages PT4B–1. Optical Society of America, 2014.

[HBG$^+$14] Kobi Hasharoni, Shuki Benjamin, Amir Geron, Stanislav Stepanov, Gideon Katz, Itai Epstein, Niv Margalit, and Michael Mesh. A 1.3 tb/s parallel optics vcsel link. In *SPIE OPTO*, pages 89910C–89910C. International Society for Optics and Photonics, 2014.

[KSP$^+$16] Kostas Katrinis, Dimitris Syrivelis, D Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K Hasharoni, D Raho, C Pinto, F Espina, et al. Rack-scale disaggregated cloud data centers: The dredbox project vision. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 690–695. IEEE, 2016.

[KYM16] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[Lim10] Kevin Te-Ming Lim. *Disaggregated memory architectures for blade servers.* PhD thesis, Hewlett-Packard Labs, 2010.

[SLKR16] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 101–111. IEEE, 2016.

[ZGY10] Jing Zhang, Huaxi Gu, and Yintang Yang. A high performance optical network on chip based on clos topology. In *Future Computer and Communication (ICFCC), 2010 2nd International Conference on*, volume 2, pages V2–63. IEEE, 2010.

- Professor Dionisios Pnevmatikatos

  ____

- Professor Apostolos Dollas

  ____

- Faculty Researcher Dimitrios Theodoropoulos

  ____

- Konec -