System Architecture for Internet of Things with the Extensive Use of Embedded Virtualization

Tamás Kovácsházy[1,2], Gábor Wacha[1,2], Tamás Dabóczi[1,2], Csanád Erdős[1,2], Attila Szarvas[1,2]
[1]Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
[2]Inter-University Centre for Telecommunications and Informatics
Debrecen, Hungary
khazy@mit.bme.hu, wacha@mit.bme.hu, daboczi@mit.bme.hu, erdos@mit.bme.hu, attila.szarvas@gmail.com

***Abstract*— The proposed applications of Internet of Things (IoT) are abundant (smart buildings, smart cities, smart infrastructure, vehicle to vehicle, vehicle to road communications, smart utilities, wearable computing, etc.). However, the construction and operation (Constructability and Operability) of such large scale distributed embedded systems are not straightforward. The deployment of commercially viable IoT solutions is primarily limited by our capabilities of constructing and operating such complex and heterogeneous systems. In this paper we propose a system architecture using various delicately selected types of embedded virtualization approaches to keep the constructability and operability issues under control. We evaluate the proposed system architecture using a prototype built from commercially available single board computers, sensors, and actuators. The software components of the system are selected from open source, free software, for example, we use Linux, QEMU, python, etc. on the prototype. The paper also investigates how the use of various virtualization solutions influences performance and resource requirements.**

Keywords—Internet of Things, Embedded Virtualization, Constructability, Operability, Performance, Linux, QEMU, Cyber-Physical Systems, CPS

I. INTRODUCTION

The proposed applications of Internet of Things are abundant and the basic building blocks of IoT are available; however, wide scale installation of such technologies - except some trials - is to happen. The proposed applications are smart building, smart cities, smart infrastructure and utilities, vehicle to vehicle or vehicle to road communication, wearable computing, ambient assisted living, etc. Some say [1] that these new technologies may bring even the 4th industrial revolution drastically changing how we deal with the world, how we optimize our life. In this new world we envision nearly all information is acquired, collected, stored, analyzed and acted upon in a huge interconnected super-system, or system of super-systems.

Typical layered architecture of IoT solution is shown in Fig 1. On the lowest layer sensors and actuators are in contact with the embedding physical environment. The data collection layer controls the operation of the sensors and actuators by extracting or providing information with specified timing most of the cases. The data concentrator layer is responsible for storing and making available data in the system for all other layers, i.e., it connects the lower layers to the higher data analyses, presentation and decision making layers. The analyses, presentation, and the interpretation and decision making layers implement advanced functionalities to utilize the data present in the system. The management layer uses the vertical console layer to influence the operation of the system (the individual layers) for management.

A. Motivation

Unfortunately, constructing and operating such complex system are a tremendous achievement and we are not ready for that, our current capabilities are quite limited today; especially, if we take into account that these systems must be dependable and secure also. The deployment of commercially viable IoT solutions is primarily limited by our capabilities of constructing and operating such systems. We can say that IoT is not limited by hardware or low level software today; we can build sensors, actuators, networks, computers, etc., on an acceptable price; but it is limited primarily by software gluing all of things together into a working and maintainable solution. In essence,

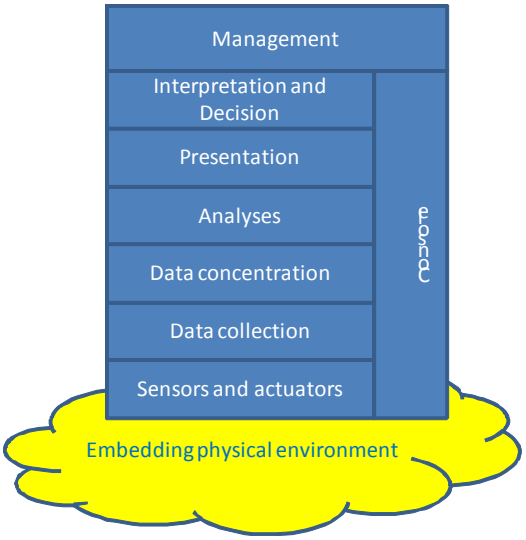

Fig. 1. Layered architecture of IoT solutions based on the distubuted monitor concept

978-1-4799-1546-0/13/$31.00 ©2013 IEEE

that means that the constructability and operability of our current IoT solutions are very low today.

Constructability (or buildability) is the property of a system describing the hardness of the timely construction process of system with acceptable quality and resource usage. Most cases constructability is handled by hierarchic design, abstractions, and universal interfaces. Therefore, we need to devise a hierarchical design and proper abstraction levels for IoT solutions, and also provide universal patterns for interoperation (universal interfaces) as IoT is primarily about using the availability of information as much as possible and as many applications as possible to leverage costs and efforts.

Operability (or maintainability) is the property of a system describing the hardness of keeping the system in a safe and functional status during its lifetime. After constructing a system, it needs to be operated for the lifetime of the system, which tends to be 10-20 years or even more, for the proposed application area of IoT. This is very long time in case of electronics, both from the point of view of component lifetime, or from the point of view of spare part availability (most modern integrated circuits are manufactured for 5-10 years today, while earlier ones are available even after 30-40 years). Furthermore, software becomes obsolete even faster, in 2-5 years most cases.

Practically, both constructability and operability are about costs, that needs to be kept low for most of the applications of IoT because the return on investment is limited in this area, i.e., the systems are optimized even today, only utilizing higher level synergies may bring in lower expenses. While constructability is relatively easy to estimate and can be determined after installation; operability is very hard to be specified for systems under design, but needs to be taken into account during design (designed for operability) because otherwise maintenance costs would be unacceptable.

II. Embedded Virtualization

Constructability and operability of a software solution are closely related to (and influenced by) portability. In other words, if a software is portable, it is supposed to be easier to construct and build complex systems based on it, e.g., by using it as a component.

Embedded system software (firmware) is typically purpose built for an exactly specified hardware. In other words, the firmware is designed not to be portable; it is specific to the hardware, the used programming language (typically C or C++), and the software development environment. Some care is taken to write code in a portable way by restricting data type usage and language constructs, but it is done primarily for reliability purposes, not for portability. For example, MISRA C is such reliability centric solution. In addition, distributed embedded systems are also designed according to these principles limiting flexibility and reuse.

However, in the field of Information Technology, software is portable today, i.e., with a relatively low effort, it can be used on various hardware and under multiple operating systems. This is especially true for free, open source software. The prime example can be the Linux operating system (with a large number of hardware architectures it supports), and several similarly licensed applications (e.g. libreOffice, Gimp, Inkscape) that not only run under Linux, but several other operating systems as Windows and OSX. This kind of seamless portability seen in the IT world would allow us to construct and operate complex embedded systems, especially for IoT applications, where these properties are thought to be advantageous.

There is a large number of virtualization technologies used in the modern information technology for implementing virtual machines [2], and these technologies are slowly appearing in the embedded world also. The viable virtualization technologies for embedded systems are the following extending the list of technologies from [2] with virtual networking technologies:

- Portable native code in compiled languages (C/C++ with platform specific compiler like GCC),
- Virtual machine based languages (JAVA, python, perl, lua, bash, etc.),
- Multi-programmed operating systems employing process virtualization (fundamental architectural property of modern operating systems used in the IT world),
- Platform virtualization (VMWARE, KVM, Virtualbox, QEMU),
- Cloud computing (virtual IT infrastructure),
- Virtual networks (Virtual Local Area Networks, VLAN, Virtual Private Network, VPN),
- Sensor virtualization.

The fundamental question is that how these virtualization technologies can be utilized in IoT to ease the mounting constructability and operability problems.

A. *Portable native code in compiled languages*

It is possible to write such code in compiled languages (C, C++) that can be run on different hardware and software environments after compilation with a hardware and software environment specific compiler (transformation to native machine code). The basic architectural requirement for such code is that the code itself must be platform independent and the software interface to other platform specific software components (and through those to the hardware) must be present on all potential platforms. In addition, platform specific testing is also a requirement. While writing portable code is challenging, typically the best run-time performance can be achieved due to the native CPU specific machine code of the running program.

B. *Virtual machine based languages*

Virtual machine bases languages are developed to provide “the write once run anywhere” principle primarily. The task of the virtual machine is to hide the platform specific aspects of the hardware and underlying platform specific software by inserting a virtual machine layer. Some of these solutions are

typically interpreted (perl, python, php), while some others are based on intermediate byte code (Java). Due this run-time translation layer these solutions provide lower performance typically, and therefore they are less likely to be used in resource limited embedded systems using low end microcontrollers.

C. Multi-programmed operating systems

Embedded system firmware is typically not extendable, i.e., it is not possible to start new (just downloaded) software components or stop running ones during operation (run-time), but the whole firmware must be updated and the system must be restarted to allow new software components to be executed. Multi-programmed operating systems; however, are capable to start executing new programs or stop running ones upon demand using the abstraction of process. Processes are virtualized environments for running programs allowing the hardware to run multiple programs in parallel without interferences and also sharing resources in a controlled manner. There are some hardware requirements to support processes, i.e., the CPU needs to have protection levels and a Memory Management Unit (MMU) to run operating systems with processes; therefore, this kind of operating systems can only now gain wide scale acceptance as more powerful embedded platforms appear due to the advances in chip manufacturing technologies [3].

D. Platform virtualization

Platform virtualization allows running multiple operating systems (guests) in a Virtual Machine Monitor (VMM) on a single hardware (host). Combining it with emulation, the guest operating systems and software can be even compiled for different hardware architectures, but can be executed on the host allowing maximum flexibility. Unfortunately, platform virtualization; and especially emulation; are resource intensive if no hardware virtualization support present on the host hardware, and even in case of hardware virtualization support on the host, performance may drop substantially. Today most of the embedded microcontrollers and embedded processors do not have hardware support for platform virtualization. However, this is changing as platform virtualization in embedded systems envisioned gaining wide scale acceptance in the near future [4].

E. Cloud computing

The term cloud computing refers to several technologies allowing utilizing the resources of distributed computing infrastructure in a virtualized, on-demand, scalable and reliable way. IoT solutions may deploy their data collection, analyses, decision making and presentation functionalities into the cloud, in essence, providing the real functionalities from a cloud solution with better scalability, reliability.

F. Virtual networks

Virtual networks, primarily Virtual Local Area Networks (VLAN) and Virtual Private Networks (VPN) allow creating an application centric virtual network infrastructure over the physical network infrastructure for better network separation and security. In this paper virtual networks are only mentioned as an additional virtualization technology helping wide scale use of virtualization, our primarily aim is to ease the development and maintenance of IoT solutions by using various virtualization solutions on the computing nodes of the system.

G. Sensor virtualization

Sensor virtualization is a technique allowing access to one or more physical properties of the system measured by one or multiple physical sensors through one or more virtual sensors independently of the scalability, reliability, implementation, and value domain problems of the individual physical sensors. Most cases sensor virtualization is a software layer between physical sensors and applications. For example, the Android mobile operating system employs sensor virtualization for providing access to the location, orientation, etc. properties of the mobile device to application programs based on simple physical sensors such as GPS, accelerometers, digital compasses, gyroscopes.

III. Applying Virtualization for IoT

In IoT solution virtualization allows us to detach implementation of functionality and execution of implementation in a dynamic (run-time), configurable way. Changes in the requirements, available execution units and sensors and/or actuators, new applications are implemented with the same hardware and software architecture just by detecting the changes and reconfiguring (extending) the software and functionalities run-time.

To implement these functionalities, the system stores implementations in the form of portable compiled language code, virtual machine based language code, and in platform specific native code in a repository of implementations. A functionality may have multiple implementations, for example, it may be available in native code only, or in multiple native codes and portable compiled language code (as it has been already compiled by a platform specific compiler for that platform), or in a form of virtual machine based language implementation. The system also stores dependencies of the implementations, such as tested platforms, required components, etc.

Upon a change is detected in the system, a new configuration is developed and mapped to execution resources based on implementation forms available for the required functionalities. Primarily, the following basic priority based rules may apply for mapping functionalities to execution resources:

- If a functionality is available in a platform specific native form and can be executed efficiently on a native execution resource, it is mapped to such resource and executed there.
- If a functionality is available in a portable compiled language code and that can be mapped to an execution resource for which a compiler exists, then the code is compiled specifically for that execution resource, and mapped to such resource and executed there.

- If a functionality is available in a virtual machine based language code and there exist a virtual machine for that language for an execution resource, then the functionality can be mapped to such resource and executed there.
- If the functionality exists only for a specific platform in native form, and there is no available execution resource for that platform, platform virtualization is taken as the last option. The functionality is mapped to an execution resource which has emulation capabilities for the specific platform and executed there.

The functionalities are mapped to resources and executed there as processes (multi-programmed operating system) to allow parallel execution without interference between functionalities running on the execution units. Today, the price and the performance of hardware capable to run such operating system makes this option viable compared to the previously used microcontroller bases systems running non multi-programmed (with only thread support) operating systems.

IV. Prototype Implementation

We have developed a prototype implementation to evaluate the previously mentioned virtualization solutions in a real application. The selected problem can be categorized as an IT related smart building type application, in which the role of the IoT solution is to monitor and supervise the environment of several server rooms.

In the rooms we have various available sensors and actuators:

- Server computers with built in operational and environmental sensors running on x86 Linux. Operational sensors, such as CPU load, memory, disk and network usage sensors, can be accessed using the venerable SNMP protocol. Internal environmental sensors (temperature primarily) are accessed through custom interfaces developed for this project. They measure the temperature of internal components and the rotational speed of cooling fans.
- There are energy monitoring systems [5] measuring the power consumption of servers in the rooms. These devices communicate using ZigBee.
- There are temperature and humidity sensor nodes installed in the server room using custom built hardware and software to measure ambient environmental properties in specific locations in the room. These devices use Ethernet and TCP/IP for communication.
- There are external temperature and humidity sensors to take into account how these properties influence the internal temperature and the operation of the air-conditioning unit. These sensor also use Ethernet and TCP/IP.
- There are cameras observing the operational state and occupation of the server room. Based on live images image processing software running in the cloud can detect various state changes in the server rooms that influences the environmental status of the room. Such events are human occupation, operation of unmanaged equipment (for example, based on status LEDs available on the devices). This sensor type is included in the prototype also for evaluating streaming like sensors with relatively large bandwidth usage.

The various IoT related functionalities can be deployed on single board computers running Linux, or on the servers in the server rooms, or in the cloud. For the purpose of showing portability in a heterogeneous computing environment we use the following hardware:

- Standard x86 PCs running Ubuntu 12.04 LTS or Ubuntu 13.04 Linux.
- Raspberry-PI model B with an ARM11 architecture CPU (700MHz) and appropriate Raspbian Linux.
- Beagleboard-XM with an ARM Cortex-A8 architecture CPU (700 MHz) and appropriate Ubuntu 13.04 Linux.
- Atlys Xilinx Spartan-6 FPGA Development Kit with a soft core MicroBlaze CPU (75 MHz) running a custom built Linux.

The MicroBlaze CPU is included in the prototype to show how legacy software developed for a low performance (presumably outdated) limited capability platform can be used with emulation on a newer platform (Raspberry-PI or Beagleboard-XM).

We use USB (Silicon Labs Si7005 USB Dongle) and serial port based temperature and humidity sensors (Si7005 custom solution). The USB based sensors have a user-space driver written utilizing the libusb-1.0 API for user space USB access (not available on the emulated Microblaze platform). The serial port based temperature and humidity sensors use bash scripts to communicate. Visualization software is built using bash and gnuplot to present the data collected by the system. An example is shown in Fig 2.

The cameras are accessed by the mjpg-streamer video and still image streaming software. We use both USB cameras and the Raspberry-PI camera module. The software is compiled for all hardware platforms except for the Microblaze because of the limited computing resources that platform can provide.

A. Application of Cloud Computing in the Prototype

The data concentration, analyses, and presentation layer of the prototype is constructed in a Platform as a Service (PaaS) cloud computing environment. The data concentration is implemented using the Sensor Observation Service (SOS) developed by 52°North. There are two type of sensors:

- Scalar sensors: They measure a scalar physical property of the environment, for example CPU utilization, temperature, energy consumption, etc. The measurements of these sensors are stored by SOS; and therefore, other components can retrieve historical measurements also from SOS.
- Streaming sensors: They measure a large array of properties of the environment periodically, such as a

video stream. SOS is not capable to store the measured value of such sensors due to the limited storage and processing capabilities. Therefore, SOS only stores accessibility and availability information for such sensors, and based on this information, upper layer components may directly stream the measured (sensed) values from the sensor.

Image processing is done also in a PaaS based cloud environment by proprietary JAVA software. Its primary purpose is to detect changes and determine status of equipment based on LEDs independently of the illumination of the server rooms. The detected changes of the environment are stored in the SOS component as data coming from virtual sensors, i.e., the streaming data is not stored in SOS, but the result of the analyses of the streaming data is stored for future use.

B. Examining Protability

The user-space USB driver for the USB temperature and humidity sensors and the mjpg-streamer are built in C, and compiled for the used platforms (except for the Microblaze for the mjpg-streamer). Due to the similar operating system kernels (various Linux versions), kernel components (video for Linux, user space USB, etc.), the portable code, and the GCC compiler the code was portable without problem. Based on these test, portable C code and the Linux software environment together assures portability among various hardware platforms even in the case of hardware specific programs such as the user-space USB driver or video streaming.

The bash script based temperature and humidity sensor communication over the serial port (using also a USB to serial converter on the Raspberry-PI and the Beagleboard-XM due to the absence of real serial port) is fully portable except that the device file of the serial port needs to be specified on all devices manually (serial port is not plug and play). This test demonstrates that virtual machine based languages can provide exceptional portability.

The implementation, portability properties and performance of the platform virtualization solution is detailed in the next section.

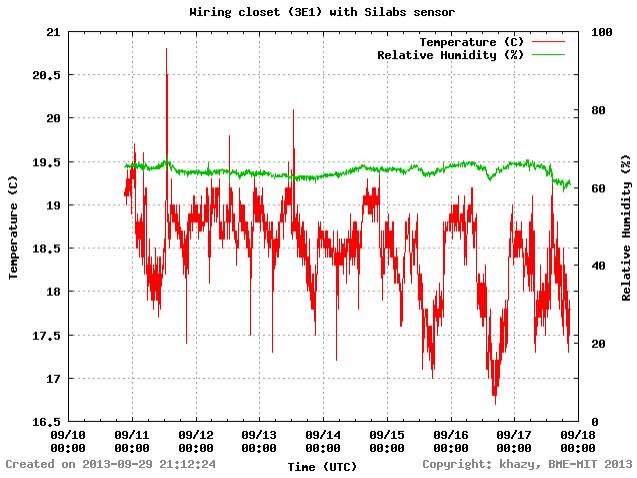

Fig. 2. One day measurement of the ambient temperatue and humidity of the wiring closet measured by one of the USB based sensors.

V. Benchmark Results for Platform Virtualization

A. QEMU emulator

For processor virtualization, the QEMU emulator can be used. One of the benefits of QEMU is the great number of host and guest architectures it supports via its TCG (Tiny Core Generator) binary translation mechanism. One of the greatest advantages of QEMU in embedded systems is the Linux user mode emulation, which allows the user to spare computing power and storage space.

In Linux user mode emulation, QEMU executes only the user program on the guest architecture, every Linux system call and IO control is passed to the host Linux kernel, thus allowing greater computation speed and better interaction between the host and guest systems. One can use QEMU to access the host hardware IO space without having to create the appropriate virtual hardware for QEMU.

B. Benchmarking QEMU emulation

To measure the computing performance of a QEMU-emulated embedded platform, two different methods were used, reflecting the usual tasks of an embedded system. The first benchmark measures the computing performance of the QEMU user-space emulation, the second simple benchmark measures the IO performance. The benchmarked system is a Raspberry Pi board with Raspbian Linux distribution. The QEMU guest architecture emulated on the Raspberry Pi is a little endian MicroBlaze (microblazeel).

C. Benchmarking the computing performance

To benchmark the performance of the emulated architecture, three measurements were used. The first measures the native ARM processor performance, the second measures the emulated MicroBlaze processor performance, the third measures the native performance of a 75 MHz MicroBlaze processor.

Since the goal of this benchmark is to expose the theoretical upper limit of the computing system, and not the measurement of the performance of a specific task (for example video stream compression) the NBench synthetic computing benchmark program was used. One of the advantages of the NBench is its simplicity. The tests included in NBench are simple enough to be portable, but complex enough to test the processing power of the system under test. Also, NBench is designed to be scalable: it includes a dynamic workload adjustment, which allows the tests to adjust to the capabilities to the system, while providing consistent results. NBench does not test the multitasking capabilities of the system. Since an embedded system rarely relies on heavy multitasking, this deficiency of NBench can be neglected

As the results show, there's an approximately 95% performance reduction with using QEMU, but comparing to the native performance of a MicroBlaze running on a 75 MHz clock the two measurements are comparable. Here, we must also note that the native results are produced on the Raspberry-PI having a floating point arithmetic unit, while the Microblaze

is capable only fixed point computations, and floating point computations are emulated only.

The following table shows the benchmark results:

TABLE I. QEMU BENCHMARKS RESULTS

Computing task	Native Raspberry-PI (1/s)	QEMU MicroBlaze on Raspberry-PI) (1/s)	Native MicroBlaze (1/s)
Numeric sort	237	10	15
String sort	38	1	1
Bitfield manipulation	9.324e7	1.45e6	8.41e5
Floating point	47	2	4
Fourier transformation	2612	1	1
IDEA encryption	791	3	12
Huffman coding	487	7	9
LU decomposition	84	1	1

D. Benchmarking the IO performance

Since the QEMU Linux user mode emulation passes the guest system calls to the host kernel, software which relies on system calls and host kernel functions (for example an embedded software doing IO) can perform better than a computational software. The measurement simply tests the maximum achievable GPIO signal frequency (square wave) by continuously toggling a GPIO bit of the BCM2835 chip used on the Raspberry Pi board.

Problems of this measurement method are that the TCG used in QEMU works especially good with tight loops, so we are measuring something which can be called the "best-case" scenario. But since the most common IO tasks are built up from small loops, the measurement reflects the common use cases.

The results of the measurements are the following:

- With a native software running on the processor, a frequency of 15-16 MHz can be achieved,
- On the other hand, the emulated MicroBlaze is capable to generate a 5-6 MHz square wave.

It needs to be mentioned that the relative inaccuracy (approximately 20-25%) of the frequency of the square wave generated by the virtualized system is much greater than the relative inaccuracy (approximately 6-8 %) of the frequency of the native code generated signal, mainly because of the inaccuracy of the running time of the dynamic recompilation done by QEMU. In some applications, this inaccuracy is not tolerable.

VI. CONCLUSION

In this paper we presented our virtualization based approach for IoT solutions to handle the constructability and operability issues present in such systems. By the prototype implementation it is shown that various virtualization techniques; such as portable code, virtual machine based languages, platform virtualization, and multi-programmed operating systems are viable options even for embedded systems today; however, performance issues are to be taken into account for platform virtualization.

We investigated platform virtualization with QEMU emulator in embedded systems in detail also. Two disadvantages were found using QEMU as a Linux user mode emulator. The first issue is the much lower performance compared to native implementation, and the second one is that the QEMU Linux user mode is somewhat underdeveloped compared to recent Linux kernels, e.g., we were not able to use our user-space USB solution due to some missing kernel interface implementations. Nonetheless, for simple tasks where portability and direct hardware access is important, QEMU can be an option for virtualization even in IoT solutions; furthermore, with hardware virtualization support, performance problems may be decreased [6].

ACKNOWLEDGMENT

The publication was supported by the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project. The project has been supported by the European Union, co-financed by the European Social Fund.

Silicon Laboratories Hungary Kft. has supported this work by donating Si7005 temperature and humidity sensor chips and Si7005 USB dongles.

REFERENCES

[1] John Donovan, "The 4th Industrial Revolution is upon us", Electronic Component News, 10/29/2013, Availabel: http://www.ecnmag.com /articles/2013/10/4th-industrial-revolution-upon-us

[2] SCOPE Alliance, „Virtualization: State of the Art," 2008. [Online]. Available: http://scope-alliance.org/sites/default/files/documents/SCOPE-Virtualization-StateofTheArt-Version-1.0.pdf.

[3] Hanák, P., N. Kiss, T. Kovácsházy, B. Pataki, M. Salamon, Cs Seres, Cs Tóth, and J. Varga. "System Architecture for Home Health and Patient Activity Monitoring." In *5th European Conference of the International Federation for Medical and Biological Engineering*, pp. 945-948. Springer Berlin Heidelberg, 2012.

[4] Heiser, Gernot. "Virtualizing embedded systems: why bother?." In Proceedings of the 48th Design Automation Conference, pp. 901-905. ACM, 2011.

[5] Kovácsházy, Tamás, Gábor Fodor, and Csaba Bernát Seres. "A distributed power consumption measurement system and its applications." In *Carpathian Control Conference (ICCC), 2011 12th International*, pp. 224-229. IEEE, 2011.

[6] Motakis, Antonios, Alexander Spyridakis, and Daniel Raho. "Introduction on performance analysis and profiling methodologies for KVM on ARM virtualization." In SPIE Microtechnologies, pp. 87640N-87640N. International Society for Optics and Photonics, 2013.