Embedded Hypervisor Xvisor: A comparative analysis

Anup Patel
Individual Researcher,
Bangalore, India
anup@brainfault.org

Mai Daftedar
Ain Shams University,
Cairo, Egypt
mai.daftedar@ieee.org

Mohmad Shalan
American University of Cairo,
Cairo, Egypt
mshalan@aucegypt.edu

M. Watheq El-Kharashi
Ain Shams University,
Cairo, Egypt
watheq@engr.uvic.ca

***Abstract*—Virtualization technology has shown immense popularity within embedded systems due to its direct relationship with cost reduction, better resource utilization, and higher performance measures. Efficient hypervisors are required to achieve such high performance measures in virtualized environments, while taking into consideration the low memory footprints as well as the stringent timing constraints of embedded systems. Although there are a number of open-source hypervisors available such as Xen, Linux KVM and OKL4 Microvisor, this is the first paper to present the open-source embedded hypervisor eXtensible Versatile hypervISOR (Xvisor) and compare it against two of the commonly used hypervisors KVM and Xen in-terms of comparison factors that affect the whole system performance. Experimental results on ARM architecture prove Xvisor's lower CPU overhead; higher memory bandwidth; lower lock synchronization latency and lower virtual timer interrupt overhead and thus overall enhanced virtualized embedded system performance.**

Keywords-component; Embedded Systems; Virtualization; Xen, Linux KVM; Xvisor

I. INTRODUCTION

The increasing demands of multi core embedded systems, has recently lead researchers to make use of virtualization technology in embedded systems. Embedded systems usually feature resource limitations, real-time constraints, high performance requirements, and increasing application stack requirements that entail the use of limited peripherals [1].

Virtualization technology provides the means to run multiple operating systems (OSs) as virtual machines (VMs or guests) on single or multi core processing units. Each guest runs on one or more virtual processing units (vCPUs). Furthermore, all vCPUs of a guest are isolated from one another but share the same peripherals.

Accordingly, use of virtualization provides the following advantages [2]:

a) consolidating services, which were once running on different devices as separate VMs on the same device;
b) combining the OS's real-time and general purpose features on one device [3];
c) better fault tolerance;
d) providing isolation to highly secured applications;

Similarly, embedded virtualization has effectively proven its efficiency in a number of use cases:

1. Motorola Evoke, the first virtualized phone [4].
2. Industrial automation in which virtualization allows the addition of extra software application without the need to add more processing units [5].
3. Cars can consolidate multiple services on the same hardware, by running Infotainment OS, AUTOSAR OS, and RTOS on separate virtual machines [6].
4. Other use cases include devices used in retail and gaming.

The presented analytical research compares the new embedded hypervisor Xvisor to the two existing embedded open source hypervisors: KVM and Xen under the following considerations:

1. Research is based on ARM architecture since latest ARM processors have hardware virtualization extensions and ARM processors are widely used in embedded systems. In addition, the majority of the board support packages in common between the three hypervisors are for ARM architecture.
2. KVM and Xen are chosen for comparison due to: their support for ARM architecture [7, 8], their open source availability allows us to gather performance numbers without any restrictions [9], board support mapping with Xvisor's board support packages and finally their extension to the embedded systems.
3. Experimental results use micro-benchmarks, which test operations such as memory accesses, cache accesses, integer operations, and task handling on running guests. Performance enhancements in such operations, improves the overall performance of the system. Unlike macro-benchmarks, which are dedicated to test a specific workload such as web server, kernel compilation, graphics rendering, etc.
4. Experimental results run only General Purpose Operating System (GPOS) Linux as guest, since for the current date this is the only guest OS supported by all three hypervisors. There is no common real-time operating system (RTOS) available for all three hypervisors. Hence testing of commitments to timing constraints will be carried

out through measurements of these two factors: 1) low memory and CPU overhead from the hypervisor; 2) efficiency of the guest OS scheduling with minimal overheard from the hypervisor.

An explanation of Xen, KVM and Xvisor's implementation and a discussion of their limitations, is presented as follows: Section II explains the virtualization classification. Section III introduces Xvisor and provides an overview of the open source hypervisors Xen and KVM. Explaining the implementation details for the main components that affect the comparison factors are highlighted in the proceeding sections. The sections that follow will include a comparative analysis between these hypervisors in-context of ARM architecture [10]. We address the guest IO emulation in Section IV, while host interrupt processing, lock synchronization, and memory management and memory footprint are presented in Sections V, VI, VII, and VIII respectively. A brief description of the application benchmarks used for our analytical analysis is provided in section IX followed by experimental results in section X and a conclusion section.

II. VIRTUALIZATION CLASSFICATION

We classify hypervisors into five categories based on two aspects [11]: (1) Hypervisor design and (2) Virtualization mode as shown in Fig. 1. Accordingly, classification plays a big role in performance measures obtained by the guests.

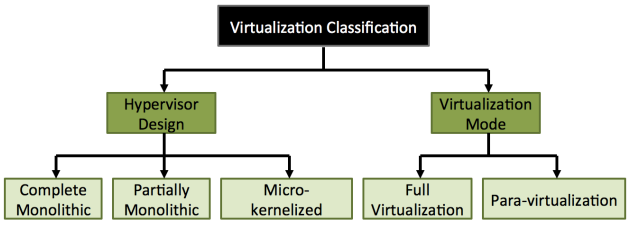

Figure 1. Virtualization classification.

A. Hypervisor Design

The hypervisor serves as an interface between the hardware and virtual machines. The manner by which these hypervisors are implemented determines how effectively they operate as virtualization managers. Based on their implementation, all hypervisors fall in one of these hypervisor design categories.

1) Complete Monolithic

Complete monolithic hypervisor is a single software layer responsible for host hardware access, CPU virtualization, and guest IO emulation. Examples of complete monolithic hypervisor are: Xvisor and VMware ESXi Server [12]

2) Partially Monolithic

Partially monolithic hypervisors are usually an extension of general purpose monolithic OS (e.g. Linux, FreeBSD, NetBSD, Windows, etc.). They support host hardware access and CPU virtualization in OS kernel and support guest IO emulation from user-space software. Examples of partially monolithic hypervisor are: Linux KVM and VMware Workstation [13].

3) Micro-kernelized

Micro-kernelized hypervisors are usually lightweight micro-kernels providing basic host hardware access and CPU virtualization in hypervisor micro-kernel. They depend on a management virtual machine for full host hardware access, guest IO emulation, and rest of the services. Some of these micro-kernelized hypervisors run each host device driver under a separate driver virtual machine instead of having it under common management virtual machine. Examples of micro-kernelized hypervisor are Xen, Microsoft Hyper-V [14], OKL4 Microvisor [15], and INTEGRITY Multivisor [16].

B. Virtualization mode

The virtualization mode determines the type of guests that can run on top of the hypervisor [17, 18].

1) Full virtualization

Unmodified guest OSs are allowed to run as guest by providing system virtual machines (i.e. emulating whole system resembling real hardware for guest).

2) Para-virtualization

Modified guest OSs are allowed to run as guest by providing hypercalls. This mode requires guest to use hypercalls for various IO operations (i.e. network send/receive, block read/write, console read/write, etc.) and sometimes in place of critical instructions. These hypercalls trigger a trap interrupt in the hypervisor, which based on the hypercall parameters, causes the hypervisor to provide the desired service to the guest.

III. OPEN SOURCE HYPERVISORS FOR EMBEDDED SYSTEMS

Following is a brief introduction of the two open source hypervisors Xen and KVM compared against the hypervisor under study, Xvisor.

Since our research is concerned with performance comparisons, we address the hypervisor implementation details for system components known to incur overheads for running guests, and hence affect the performance of the whole virtualized embedded system. This includes how each hypervisor handles CPU virtualization, guest IO emulation and host hardware accesses. Additionally, some insights concerning the main advantage characterizing each hypervisor are mentioned.

A. XEN

The Xen hypervisor shown in Fig. 2 is a micro-kernelized hypervisor that supports both fully virtualized and para-virtualized guests. The Xen hypervisor kernel is a lightweight micro-kernel which provides: CPU virtualization, MMU virtualization, virtual IRQ handling, and inter-guest communication [19]. Domain is the term used in Xen kernel to refer to a virtual machine or guest. Domain0 (Dom0) is a special type of domain that runs a modified version of Linux kernel. It must be running a priori any other virtual machine or guest and has full access to underlying host hardware.

Dom0's main purpose is to provide IO virtualization services and guest management services using Linux kernel. DomainU (DomU) refers to the guest virtual machine with a running guest OS.

Communication between DomU and Dom0 is required for handling emulated or para-virtualized guest IO events.

This is achieved using Xen event channel. The para-virtualized guest, referred to as DomU PVM uses Xen event channel for accessing Dom0 para-virtualized IO services. However, the fully virtualized guest referred to as DomU HVM uses QEMU running on Dom0 user space for emulating guest IO events. All user interactions to manage guests or domains are done using Xen toolstack running on Dom0 user space. Without Dom0, Xen cannot provide DomU PVM or DomU HVM.

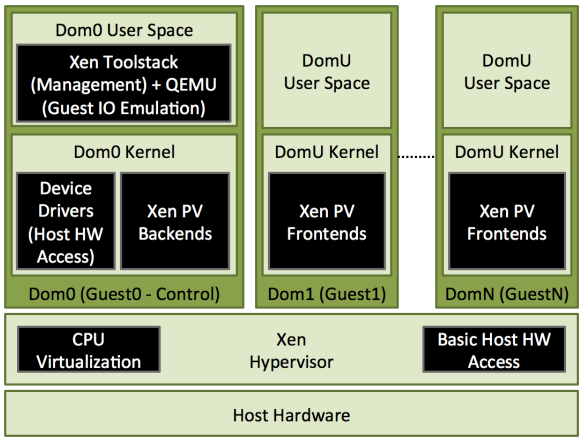

Figure 2. Xen architecture.

The most important advantage of Xen is the use of Linux kernel as Dom0 as it helps Xen reuse existing device drivers and other parts of Linux kernel. However, this advantage comes at a price of additional overhead as will be described in subsequent sections. Also, Linux kernel running as Dom0 performs slightly slower than Linux kernel running directly on hardware without Xen. This is because Dom0 is just another domain for Xen, having its own nested page table and can also be scheduled-out by Xen scheduler.

B. KVM

Kernel-based Virtual Machine (KVM) is a partially monolithic hypervisor that supports both full virtualization as well as para-virtualization. KVM primarily provides fully virtualized guest and provides para-virtualization in the form of optional VirtIO devices [20].

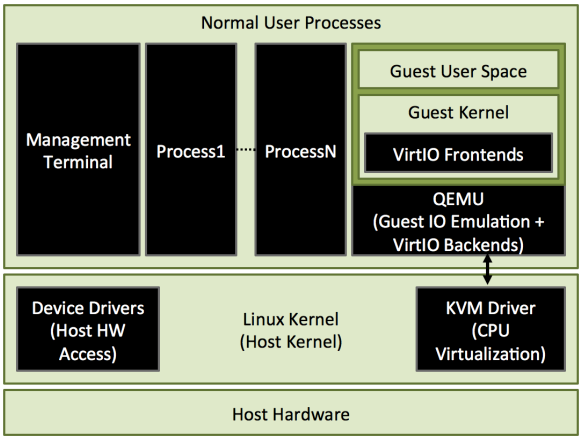

Figure 3. KVM architecture.

KVM extends Linux kernel's execution modes to allow its operation as a hypervisor. The new mode added to the two execution modes (kernel and user) is the guest mode. This allows guest OSs to run with the same execution modes as host OS except that certain instructions, register accesses, and IO accesses will be trapped to host Linux kernel. The host Linux kernel will view a virtual machine as a QEMU process. The KVM only virtualizes CPU in host kernel, and depends on QEMU running in user-space for handling emulated and para-virtualized guest IO events.

Fig. 3 shows KVM's building blocks comprising two main components: (1) Kernel-space character device driver, which provides CPU virtualization services and memory virtualization through a character device file /dev/kvm, and (2) Userspace emulator for guest hardware emulation (typically QEMU). Service request communications between these two components (such as the creation of virtual machine and vCPUs) is handled through the IOCTL system over /dev/kvm device file.

The most important advantage of KVM (just like Xen) is its use of Linux kernel as host kernel, which thus helps KVM reuse existing Linux device drivers and other parts of Linux kernel. Nevertheless, this advantage comes at a price that essentially degrades KVM's overall performance, due to operations such as KVM's world-switch from Guest mode to Host mode upon nested page faults, special instruction trap, host interrupts, guest IO events and another world-switch from Host mode to Guest mode to resume the Guest execution.

C. Xvisor

Xvisor shown in Fig. 4, is a complete monolithic hypervisor that supports both full virtualization and para-virtualization. It aims to provide a lightweight hypervisor that can be used within embedded systems with less overhead and small memory footprint. Xvisor primarily provides fully virtualized guest and provides para-virtualization in the form of optional VirtIO devices [20].

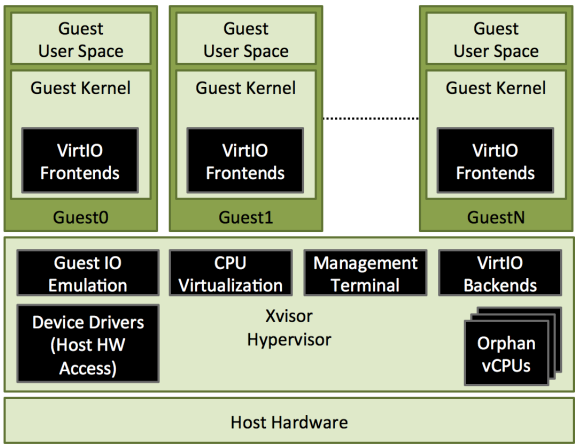

Figure 4. Xvisor architecture.

All core components of Xvisor such as: CPU virtualization, guest IO emulation, background threads, para-virtualization services, management services, and device drivers run as a single software layer with no pre-requisite tool or binary file.

The guest OS runs on what Xvisor implementers call Normal vCPUs, having a privilege less than Xvisor. Moreover, all background processing for device drivers

and management purposes run on Orphan vCPUs with highest privilege. Guest configuration is maintained in the form of a tree data structure called device tree [21]. This facilitates easier manipulation of guest hardware through device tree script (DTS). In other words, no source code changes are required for creating a customized guest for embedded systems.

The most important advantage of Xvisor is its single software layer running with highest privilege, in which all virtualization related services are provided. Unlike KVM, Xvisor's context switches are very lightweight (refer to section V) resulting in fast handling of nested page faults, special instruction traps, host interrupts, and guest IO events. Furthermore, all device drivers run directly as part of Xvisor with full privilege and without nested page table (unlike Xen) ensuring no degradation in device driver performance. In addition, the Xvisor vCPU scheduler is per-CPU and does not do load balancing for multi-processor systems. The multi-processor load balancer is a separate entity in Xvisor, independent of the vCPU scheduler (unlike KVM and Xen). Both, vCPU scheduler and load balancer are extensible in Xvisor.

Xvisor's only limitation is its lack of rich board and device driver support like Linux. To tackle this limitation Xvisor provides Linux compatible headers for porting device driver frameworks and device drivers from Linux kernel. Albeit not completely solving the problem, porting efforts are greatly reduced.

IV. GUEST IO EMULATION

Embedded systems will need to run legacy software as virtual machine or guest. This legacy embedded software might expect particular type of hardware that hypervisors have to emulate. It is thus imperative that hypervisors have minimum overhead in emulating guest IO events.

The following subsections explain the lifecycle of emulated guest IO events on the ARM architecture for the aforementioned hypervisors. It is important to note that the flow of emulated guest IO events in the indicated hypervisors is the same on all architectures including ARM.

A. Xen ARM

Fig. 5 shows the lifecycle of emulated guest IO events on Xen ARM for DomU HVM. Beginning at (1), a guest IO event is triggered which is then forwarded to Dom0 kernel using Xen event channel as shown at (2) and (3). Subsequently, QEMU running in Dom0 user space emulates guest IO events as shown at (4). Lastly at (5), control returns to DomU.

The shown flow incurs a number of overheads. First, the Xen event channel-based inter-domain communication (though optimized over time) has non-zero overhead. Secondly, context switches from Dom0 kernel to user space and vice versa also adds overhead to handling emulated guest IO events.

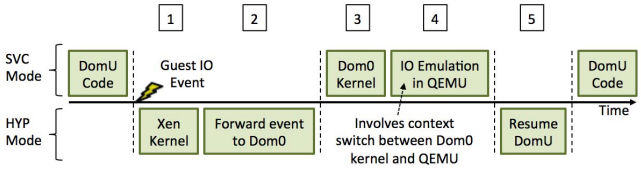

Figure 5. Emulated guest IO event on Xen ARM.

B. KVM ARM

Fig. 6 shows how an emulated guest IO event is handled in KVM ARM. The shown scenario starts at (1) when a guest IO event is triggered, causing a VM-exit to switch from guest mode to host mode. Guest IO events are then handled by QEMU running in user space as shown at (2) and (3). Finally at (4), VM-enter occurs which switches host mode to guest mode.

The sustained overhead is mainly due to VM-exit and VM-enter context switches, which are known to be heavily taxing for KVM.

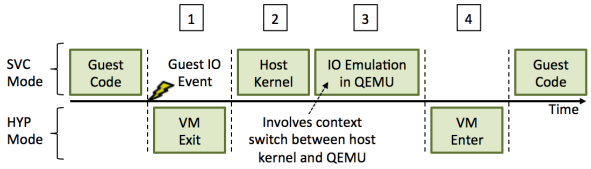

Figure 6. Emulated guest IO event on KVM ARM.

C. Xvisor ARM

Unlike other hypervisors, Xvisor ARM does not incur any additional scheduling or context switch overhead in emulating guest IO events. As presented in Fig. 7, the scenario starts at (1) when a guest IO event is trapped by Xvisor ARM and (2) handles it in a non-sleepable normal (or emulation) context. The non-sleepable normal context ensures fixed and predictable overhead.

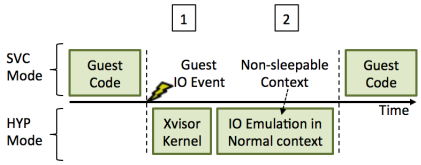

Figure 7. Emulated guest IO event on Xvisor ARM.

V. HOST INTERRUPTS

Embedded systems have to comply with the stringent timing constraints when processing host interrupts. In virtualized environments, hypervisors can have additional overheads in processing host interrupts, which in turn affects host IO performance. It is important to note that the flow of host interrupt handling in indicated hypervisors are the same for all architectures including ARM.

A. Xen ARM

In Xen, host device drivers run as part of Dom0 Linux kernel. Hence all host interrupts are routed to Dom0. As shown in Fig. 8, the scenario starts at (1) when a host IRQ is triggered which is then routed to Dom0 at (2). All host interrupts are handled by Dom0 as shown in (3) and (4). If a host interrupt is triggered while DomU is running, then it will be processed only after Dom0 is scheduled-in hence, host interrupt handling incurs scheduling overhead.

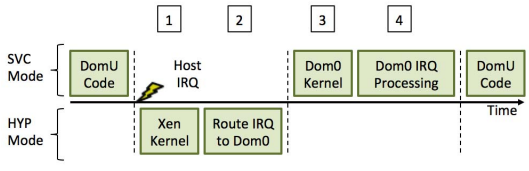

Figure 8. Host interrupt handling on Xen ARM.

B. *KVM ARM*

Fig. 9 shows host interrupt handling in KVM ARM while guest is running [22]. Each host interrupt triggers VM-exit as shown at (1). Once the interrupt is serviced by the host kernel as shown at (2) and (3), KVM resumes guest by undergoing VM-entry at (4). The VM-exit and VM-entry add considerable overhead to process host interrupts while a KVM guest is running. Further, scheduling overhead occurs if host interrupt is to be routed to KVM guest.

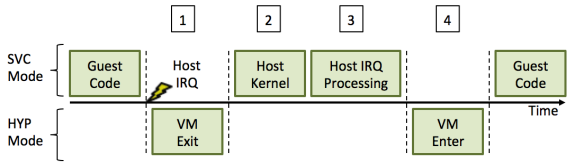

Figure 9. Host interrupt handling on KVM ARM.

C. *Xvisor ARM*

Xvisor's host device drivers generally run as part of Xvisor with highest privilege. Hence, no scheduling or context switch overhead is incurred for processing host interrupts as shown in Fig. 10. A scheduling overhead only incurs if the host interrupt is routed to guest, which is not running currently.

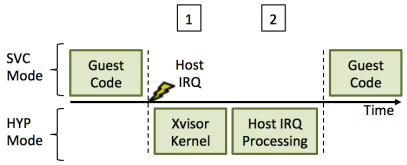

Figure 10. Host interrupts handling on Xvisor ARM.

VI. LOCK SYNCHRONIZATION LATENCY

In virtualized environments, the lock synchronization latency issue is a well-known problem mentioned in literature as in [23]. It takes place due to the presence of two schedulers: the hypervisor scheduler and the guest OS scheduler. Here, both schedulers are not aware of each other, causing the guest vCPUs to be preempted by the hypervisor at any time. We present a brief description about the implications of such latencies and how all three hypervisors handle them.

Inefficient handling of lock synchronization between vCPUs of the same guest results in two problematic scenarios (as presented in Fig. 11 and Fig. 12): vCPU pre-emption and vCPU stacking issues. Both issues could lead to prolong waiting time in acquiring locks for vCPUs of the same guest.

The vCPU preemption issue is initiated when a vCPU running on a certain host CPU holding a lock is preempted while another vCPU running concomitantly on another host CPU is waiting for that lock. Additionally, the vCPU stacking issue takes place due to a lock scheduling conflict that occurs on a single host CPU running various vCPUs. That is, a vCPU (vCPU1) accessing a lock is scheduled prior to the vCPU (vCPU0) that is already holding the lock on the same host CPU.

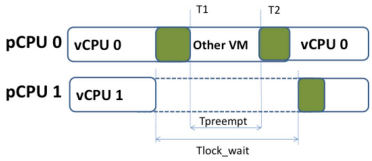

Figure 11. vCPU preemption issue.

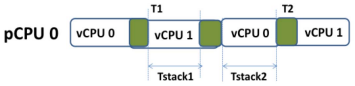

Figure 12. vCPU stacking issue.

On ARM architecture, OSs typically use Wait For Event (WFE) instruction while waiting to acquire a lock and use Send Event (SEV) instruction when releasing a lock. The ARM architecture allows WFE instruction to be trapped by the hypervisor but SEV instruction cannot be trapped. To solve the vCPU stacking issue, all three hypervisors (Xen ARM, KVM ARM and Xvisor ARM) trap WFE instruction for yielding vCPU time-slice. The vCPU preemption issue on ARM can only be solved using paravirtualized locks which requires source level changes in guest OS.

VII. MEMORY MANAGEMENT

Embedded systems require efficient memory handling. The overhead sustained by memory management is an important consideration with embedded hypervisors.

The ARM architecture provides two-staged translation tables (or nested page tables) for guest memory virtualization. Fig. 13 shows the two-staged MMU on ARM. The guest OS is responsible for programming stage1 translation table which carries out guest virtual address (GVA) to intermediate physical address (IPA) translation. The ARM hypervisors are responsible for programming stage2 translation table to achieve intermediate physical address (IPA) to actual physical address (PA) translation.

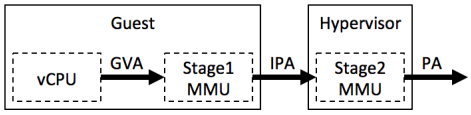

Figure 13. Two-staged MMU on ARM.

Translation table walks are required upon TLB misses. The number levels of stage2 translation table accessed through this process affect the memory bandwidth and overall performance of virtualized system. Such that N levels in stage1 translation table and M levels in stage2 translation table will carry out NxM memory accesses in worst-case scenarios. Clearly, the TLB-miss penalty is very expensive for guests on any virtualized system. To reduce TLB-miss penalty in two-staged MMU, ARM hypervisors create bigger pages in stage2 translation table.

A. *Xen ARM*

Xen ARM creates a separate three level stage2 translation table for each guest or domain (Dom0 or DomU). It can create 4KB or 2MB or 1GB translation table entries in stage2. Xen ARM also allocates guest memory on-demand and attempts to construct the biggest

possible translation table entry in stage2 based on IPA and PA alignment.

B. KVM ARM

The KVM user space tool (QEMU) pre-allocates guest RAM as user space memory and informs its location to KVM ARM kernel module. The KVM ARM kernel module creates a separate three-level stage2 translation table for each guest vCPU. Normally KVM ARM will create 4KB stage2 translation table entries but it can also create 2MB stage2 translation table entries using HugeTLB optimization.

C. Xvisor ARM

Xvisor ARM pre-allocates contiguous host memory as guest RAM at guest creation time. It creates a separate three level stage2 translation table for each guest. Xvisor ARM can create 4KB or 2MB or 1GB translation table entries in stage2. Additionally, it always creates the biggest possible translation table entry in stage2 based on IPA and PA alignment. Finally, the guest RAM being flat/contiguous (unlike other hypervisors) helps cache speculative access, which further improves memory accesses for guests.

VIII. MEMORY FOOTPRINT COMPARISON

Embedded systems require small memory footprint [24]. Tables I, II and III below show the required installations and their minimum memory consumption on Cubieboard2 [25]. The following questions will thus be answered:

1. What is required to get Xen ARM, KVM ARM and Xvisor ARM running on the system?
2. What is the minimum memory consumed by Xen ARM, KVM ARM and Xvisor ARM?

TABLE I. XEN ARM INSTALLATIONS

Xen ARM	**Installations**		
	Xen Kernel	***Dom0 Linux zImage***	***Toolstack***[a]
Size	600+ KB	3-4 MB	20 MB <
Memory on cubieboard2	160+ MB[b]	20+ MB	---

[a] Xen toolstack is installed alongside other dependent libraries. This toolstack allows users to manage virtual machine creation, destruction and configuration. It is driven by command line console and a graphical interface [10].

[b] Xen reserves lot of memory for event channels.

TABLE II. KVM ARM INSTALLATIONS

KVM ARM	**Installations**	
	Host Linux zImage	***QEMU***
Size	3-4 MB	20 MB <
Memory on cubieboard2	20+ MB	---

TABLE III. XVISOR ARM INSTALLATIONS

Xvisor ARM	**Installations**
	Xvisor kernel[c]
Size	1-2 MB
Memory on cubieboard2	4+ MB out of 16MB[d]

[c] Xvisor kernel covers the complete virtualization functionality in a single binary and is expected to increase in the future with the additions of new features to (2-3 MB).

[d] Xvisor restricts its own memory consumption using compile time option which is set to 16MB as default for Xvisor ARM

IX. BENCHMARK APPLICATIONS

The benchmark applications used in our experiments focus on comparing hypervisors in-terms of CPU overhead, memory bandwidth and lock synchronization.

A. Dhrystone

Dhrystone is a simple benchmark that is used to measure the integer performance of processors [26]. Total iterations per second of Dhrystone benchmark are referred to as Dhrystones per second. Further, another representation of Dhrystone result is Dhrystone Million Instruction Per Second (DMIPS), which is Dhrystones per second divided by 1757. The DMIPS is nothing but relative performance of a computer system compared to VAX 11/780, nominally a 1 MIPS machine [27].

B. Cachebench

Cachebench is designed to evaluate performance of the memory hierarchy of a computer system [28]. Its focuses on parameterizing the performance of possibly multiple levels of cache present on and off the processor. The Cachebench does different tests such as: memory set, memory copy, integer read, integer write, and integer read-modify-write with varying buffer size. For our experiments, we will report results for memory copy and integer read-modify-write in MB/sec.

C. Stream

Memory bandwidth has been viewed to affect the system's performance [29]. STREAM is a simple, synthetic benchmark designed to measure sustainable memory bandwidth (in MB/s). The STREAM benchmark is specifically designed to work with datasets much larger than the available cache on any given system. The STREAM version used in our experimental results is v5.10 with 2000000 array size (approx. 45.8 MB).

D. Hackbench

Hackbench measures the system scheduler performance by determining the time taken to schedule a given number of tasks. The main job of Hackbench is to schedule threads and processes. These schedulable entities communicate via sockets or pipes to send data back and forth. The datasize and number of messages can be set when the running the application [30].

X. EXPERIMENTS

The following experiments aim to evaluate the newly proposed embedded hypervisor Xvisor's efficiency in comparison to KVM and Xen. Four benchmark applications were tried on guest Linux running on Cubieboard2 [25]. The Cubieboard2 is an ARM Cortex-A7 dual core 1GHz board with 1GB RAM. The following hypervisor versions are used in our experiments:

1. **KVM:** Latest Linux-3.16-rc3 is used as Host KVM kernel. The guest kernel is Linux-3.16-rc3.
2. **Xen:** Latest Xen-4.5-unstable kernel dated 3rd August 2014 is used as hypervisor. The Dom0 kernel is Linux-3.16-rc3 and DomU kernel is also Linux-3.16-rc3.
3. **Xvisor:** Latest Xvisor-0.2.4+ dated 18th July 2014 is used as hypervisor. The guest kernel is Linux-3.16-rc3.

Experimental results are obtained with two test vectors. The first runs over a single core, while the second runs over a dual core. The systems under test (SUTs) are: (1) Host without any hypervisor; (2) Xvisor guest; (3) KVM guest; (4) KVM guest with HugeTLB and (5) Xen guest.

In order to ensure that only CPU overhead, memory bandwidth and lock synchronization latency are taken into consideration, both test vectors have one paravirtualized guest with two vCPUs. Moreover, all hypervisors have the following optimizations: No maintenance interrupt from generic interrupt controller, Super pages support for Xen ARM, and Trap-and-yield vCPU on WFE instruction.

Tables IV and V, present Dhrystone results in DMIPS. The DMIPS obtained on Xvisor guest are around 0.2% higher than KVM guest, 0.19% higher than KVM guest with HugeTLB, and 0.46% higher than Xen DomU. The Dhrystone benchmark is small in size and mostly fits in cache at runtime hence memory access overhead does not affect it. Despite obtaining improvement of 2 DMIPS, this still improves the overall system performance because 1 DMIPS equals 1757 iterations-per-second. Therefore, actual improvement will be thousands of Dhrystone iterations (typically few million machine cycles).

Results (MB/s) in Table VI, VII, VIII and IX show Cachebench results, through two operations: memory copy and integer read-modify-write. The memory copy results of Xvisor guest are around 18% higher than KVM guest, 1.2% higher than KVM guest with HugeTLB, and 0.67% higher than Xen DomU. Also, integer read-modify-write results of Xvisor guest are around 1.14% higher than KVM guest, 1.2% higher than KVM guest with HugeTLB, and 1.64% higher than Xen DomU.

Results presented in Tables X and XI, show sustainable memory bandwidth in Xvisor guest are around 0.72% higher than KVM guest, 1.57% higher than KVM guest with HugeTLB and 1.2% higher than Xen DomU.

Hackbench results presented in Tables XII and XIII show that task dispatch latency on Xvisor guest is around 12.5% lower than KVM guest, 5.62% lower than KVM guest with HugeTLB and 6.39% lower than Xen DomU.

TABLE IV. DHRYSTONE WITH THE FIRST TEST VECTOR

SUT	Dhrystone (DMIPS) (Host with 1 CPU and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
Run 1	2558600	2549738	2543283	2542991	2531857
Run 2	2558695	2549818	2543335	2542969	2531820
Run 3	2558864	2554009	2543443	2542946	2530996
Run 4	2558631	2548488	2543510	2542657	2531254
Run 5	2559466	2549626	2543640	2542866	2531471
Avg.	2558851	2550336	2543442	2542886	2531480
DMPIS	1456.3	1451.52	1447.60	1447.28	1440.79

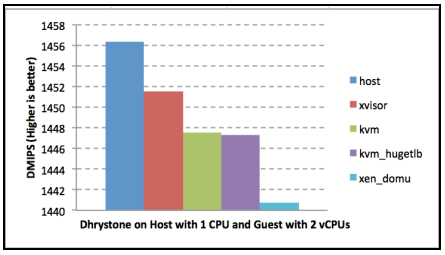

Figure 14. Dhrystone with the first test vector.

TABLE V. DHRYSTONE WITH THE SECOND TEST VECTOR

SUT	Dhrystone (DMIPS) (Host with 2 CPUs and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
Run 1	2558559	2558553	2555134	2555882	2553904
Run 2	2559327	2558656	2555213	2555849	2553737
Run 3	2558820	2558574	2555038	2555765	2553626
Run 4	2559140	2558698	2555159	2555890	2553842
Run 5	2559035	2558592	2555133	2555814	2553590
Avg.	2558976	2558615	2555135	2555840	2553740
DMPIS	1456.44	1456.24	1454.26	1454.66	1453.46

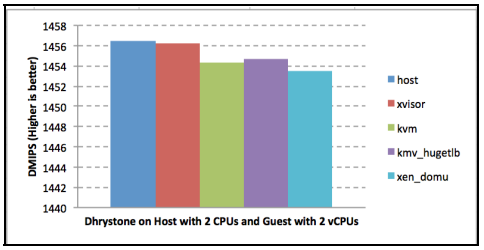

Figure 15. Dhrystone with the second test vector.

TABLE VI. CACHEBENCH MEMCOPY WITH THE FIRST TEST VECTOR

Size of data	Cachebench memory copy (MB/s) (Host with 1 CPU and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
1.5M	1748.4	1729.86	1333.2	1702.2	1705.8
2M	1748	1733.1	1399.6	1699	1696.41
3M	1746.4	1728.65	1483.8	1701.8	1686.23
4M	1749.6	1706.99	1524.9	1702.2	1706.99
6M	1746.77	1705.39	1542.5	1702.99	1702.99
8M	1748	1708.37	1571.3	1702.2	1708.58
12M	1744.62	1735.22	1576.0	1701.99	1705.39
16M	1738.25	1711.33	1501	1701.99	1708.37
Avg.	1746.25	1719.86	1491.55	1701.79	1702.59

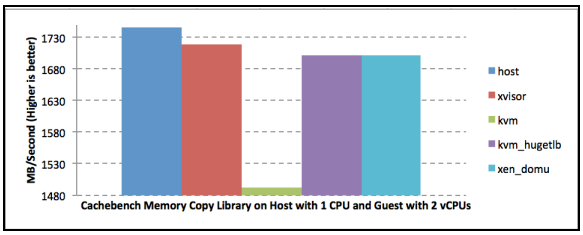

Figure 16. Cachebench memcopy with the first test vector.

TABLE VII. CACHEBENCH INT RMW WITH THE FIRST TEST VECTOR

Size of data	Cachebench int Read-Modify-Write (MB/s) (Host with 1 CPU and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
1.5M	565.8	549.8	544.91	546.11	548.5
2M	564.47	560.26	544.51	546.11	546.11
3M	564.07	558.45	544.51	546.11	547.41
4M	563.67	558.49	545.02	545.53	549.3
6M	564.14	559.74	543.71	546.11	548.5
8M	565.27	550.3	543.94	546.03	547.12
12M	564.14	561.85	543.94	546.11	549.8
16M	565.08	550.2	543.87	546.03	549.11
Avg.	564.58	556.13	544.30	546.01	548.23

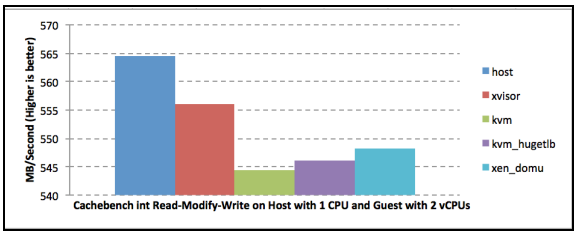

Figure 17. Cachebench int RMW with the first test vector.

TABLE VIII. CACHEBENCH MEMCOPY WITH THE SECOND TEST VECTOR

Size of data	Cachebench memory copy (MB/s) (Host with 2 CPUs and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
1.5M	1746.2	1742.63	1634.13	1727.09	1736.53
2M	1746.8	1743.43	1381.24	1727.49	1734.93
3M	1747.6	1742.63	1271.71	1728.29	1738.92
4M	1774.45	1745.31	1284.46	1730.94	1740.52
6M	1785.63	1742.63	1309.74	1731.74	1742.63
8M	1768.06	1743.14	1341.83	1730.68	1740.52
12M	1766.47	1745.02	1367.33	1727.24	1743.71
16M	1748.8	1743.14	1377.78	1730.42	1743.71
Avg.	1760.50	1743.49	1371.02	1729.23	1740.18

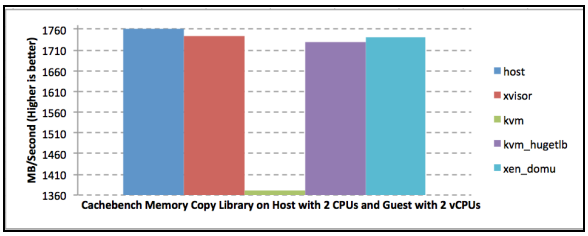

Figure 18. Cachebench memcopy with the second test vector.

TABLE IX. CACHEBENCH INT RMW WITH THE SECOND TEST VECTOR

Size of data	Cachebench int Read-Modify-Write (MB/s) (Host with 2 CPUs and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
1.5M	565.2	561.16	553.29	556.97	559.88
2M	564.8	560.96	552.19	558.08	559.68
3M	564.07	561.08	551.09	558.08	559.36
4M	563.67	560.96	551.89	557.77	559.36
6M	564.14	561.75	552.19	557.14	560.48
8M	564.14	560.79	552.38	557.77	559.84
12M	564.14	560.79	551.29	557.14	559.36
16M	563.96	561.52	552.38	557.62	559.84
Avg.	564.26	561.12	552.08	557.57	559.72

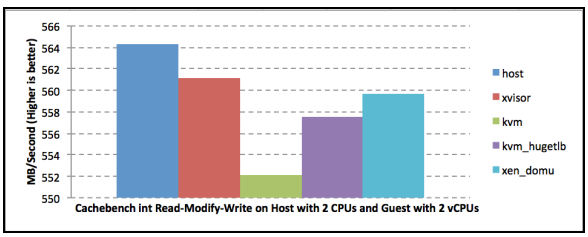

Figure 19. Cachebench int RMW with the second test vector.

TABLE X. STREAM v5.10 SCALE WITH THE FIRST TEST VECTOR

SUT	Stream v5.10 Scale (MB/s) (Host with 1 CPU and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
Run 1	289	287.5	280.4	280.6	282.2
Run 2	289	287.5	279.6	280.5	282.4
Run 3	288.9	287.5	279.8	280.6	282.4
Run 4	288.9	287.6	279.8	280.4	282.3
Run 5	289	287.6	279.5	280.5	283.2
Avg.	288.96	287.54	279.82	280.52	282.5

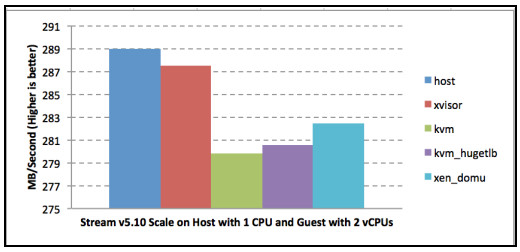

Figure 20. Stream v5.10 with the first test vector.

TABLE XI. STREAM v5.10 SCALE WITH THE SECOND TEST VECTOR

SUT	Stream v5.10 Scale (MB/s) (Host with 2 CPUs and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
Run 1	289	287.6	284.3	285.8	286.7
Run 2	289.1	287.8	284.3	285.8	287.3
Run 3	289	287.9	284.5	285.9	287.1
Run 4	289	287.9	284.2	285.8	287.3
Run 5	289	288	284.4	285.8	287.2
Avg.	289.02	287.84	284.34	285.82	287.12

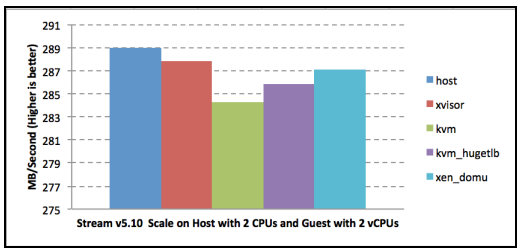

Figure 21. Stream v5.10 with the second test vector.

TABLE XII. HACKBENCH WITH THE FIRST TEST VECTOR

SUT	Hackbench (seconds) (Host with 1 CPU and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
1*40 tasks	0.2928	0.341	0.449	0.4146	0.496
2*40 tasks	0.5998	0.678	0.769	0.7802	0.7932
4*40 tasks	1.2042	1.375	1.5494	1.4606	1.4556
6*40 tasks	1.8218	2.065	2.314	2.1784	2.2532
8*40 tasks	2.4162	2.798	3.126	2.8426	2.8488
10*40 tasks	3.0532	3.609	4.0098	3.6044	3.7934

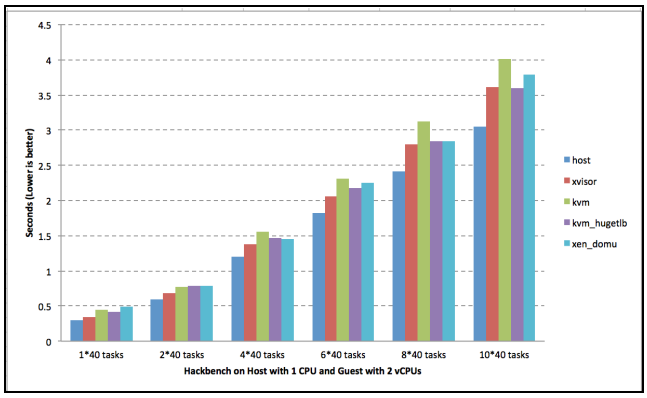

Figure 22. Hackbench with the first test vector.

TABLE XIII. HACKBENCH WITH THE SECOND TEST VECTOR

SUT	Hackbench (seconds) (Host with 2 CPUs and Guest with 2 vCPUs)				
	Host	*Xvisor*	*KVM*	*KVM HugeTLB*	*Xen*
1*40 tasks	0.1652	0.180	0.1976	0.188	0.182
2*40 tasks	0.3294	0.339	0.406	0.353	0.344
4*40 tasks	0.6678	0.666	0.7686	0.705	0.684
6*40 tasks	0.991	0.995	1.1402	1.045	1.018
8*40 tasks	1.324	1.324	1.519	1.372	1.345
10*40 tasks	1.655	1.669	1.898	1.710	1.698

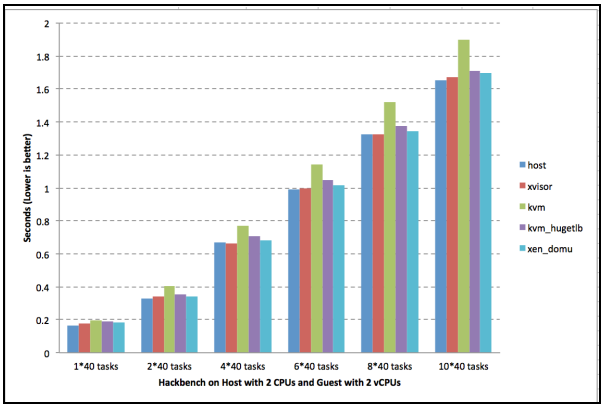

Figure 23. Hackbench with the second test vector.

XI. CONCLUSION

This paper presents the new embedded hypervisor Xvisor as a solution for the performance shortcomings present in open source hypervisors Xen and KVM. The implementation advantages of Xvisor were shown in-terms of comparison factors such as: guest IO emulation, host interrupt handling, lock synchronization latency, memory footprint and memory management. Furthermore, we supported the implementation advantages of Xvisor by showing results of four different benchmarks.

Experimental results show that Dhrystone, Cachebench and Stream benchmarks yield higher rates on Xvisor ARM guest. This indicates that Xvisor ARM guest has lower CPU overhead and higher memory bandwidth compared to KVM ARM guest and Xen ARM DomU. Moreover, Hackbench yields lower numbers on Xvisor ARM guest, showing lower lock synchronization latency and virtual timer interrupt overhead compared to KVM ARM guest and Xen ARM DomU. Such results imply that OSs (GPOS or RTOS) running as guest under Xvisor ARM will have near native performance when compared to KVM ARM and Xen ARM. Finally, lower memory footprint of Xvisor ARM allows it to efficiently utilize limited memory available on embedded systems.

Xvisor allows addition of board support packages hence; more multi-core (more than dual cores) experiments are possible. Furthermore based on its proven improved performance in the aforementioned measures, it's anticipated that future implementation of Xvisor in areas of network and storage virtualization would yield enhanced performance.

REFERENCES

[1] "Motivation for running a Hypervisor on Embedded Systems,"[Online].Available: http://www.linuxplanet.com/linuxplanet/reports/6490/1

[2] R.Kaiser, "Complex embedded systems-A case for virtualization," in Intelligent solutions in Embedded Systems, 2009 Seventh Workshop on, pp. 135-140. IEEE, 2009.

[3] Heiser, Gernot. "The role of virtualization in embedded systems," in Proceedings of the 1st workshop on Isolation and integration in embedded systems, pp. 11-16. ACM, 2008.

[4] Heiser, Gernot. "The Motorola Evoke QA4-A Case Study in Mobile Virtualization." Open Kernel Labs (2009).

[5] "Case Study: The Use of Virtualization in Embedded Systems," white paper,2013.

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

[7] Dall, Christoffer, and Jason Nieh. "KVM/ARM: Experiences Building the Linux ARM Hypervisor." (2013).

[8] Rossier, Daniel. "EmbeddedXEN: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization." White paper, Switzerland (2012).

[9] Soriga, Stefan Gabriel, and Mihai Barbulescu. "A comparison of the performance and scalability of Xen and KVM hypervisors." In Networking in Education and Research, 2013 RoEduNet International Conference 12th Edition, pp. 1-6. IEEE, 2013.

[10] "ARMv7-AR architecture reference manual," [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html

[11] "Xvisor open-source bare metal monolithic hypervisor,"[Online]. Available: http://www.xhypervisor.org/

[12] "The Architecture of VMware ESXi,"[Online].Availaible: http://www.vmware.com/files/pdf/ESXi_architecture.pdf

[13] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing Virtualization to the x86 Architecture the Origiinal VMware Workstation," ACM Transactions on Computer Systems, 30(4):12:1-12:51, Nov 2012.

[14] "OKL4 Microvisor,"[Online]. Available: http://www.ok-labs.com/_assets/download_library/OK_Labs_Product_Datasheet.pdf

[15] H. Fayyad-Kazan, L. Perneel and M. Timerman, "Benchmarking the Performance of Microsoft Hyper-V server, VMware ESXi, and Xen Hypervisors", Journal of Emerging Trends in Computing and Information Sciences, Vol. 4, No. 12, Dec 2013.

[16] "INTEGRITY Multivisor,"[Online].Availble: http://www.ghs.com/products/rtos/integrity_virtualization.html

[17] "Full Virtualization," [Online] Available: http://en.wikipedia.org/wiki/Full_virtualization

[18] "Paravirtualization," [Online] Available: http://en.wikipedia.org/wiki/Paravirtualization

[19] "Xen Project Beginners Guide,"[Online]. Available: http://wiki.xenproject.org/wiki/Xen_Project_Software_Overview

[20] R. Russell. Virtio PCI Card Specification v0.9.5 DRAFT, May 2012.

[21] G.Likely, and J.Boyer, "A symphony of flavours: Using the device tree to describe embedded hardware," in Proceedings of the Linux Symposium, vol. 2, pp. 27-37. 2008.

[22] R.Ma, F.Zhou, E.Zhu, and H.GUAN, "Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System." Journal of Information Science and Engineering 29, no. 5 (2013): 1021-1035.

[23] X.Song, J.Shi, H.Chen, and B.Zang, "Schedule processes, not vcpus," in Proceedings of the 4th Asia-Pacific Workshop on Systems, p. 1. ACM, 2013.

[24] "Memory Footprint,"[Online]. Available: http://en.wikipedia.org/wiki/Memory_footprint

[25] "Cubiboard,"[Online]. Available: http://cubieboard.org/

[26] RP.Weicker, "Dhrystone: a synthetic systems programming benchmark," Communications of the ACM 27, no. 10 (1984): 1013-1030.

[27] "Dhrystone,"[Online]. Available: http://en.wikipedia.org/wiki/Dhrystone

[28] PJ.Mucci, K.London, and J.Thurman. "The cachebench report."University of Tennessee, Knoxville, TN 19 (1998).

[29] "Stream,"[Online]. Available: http://www.cs.virginia.edu/stream/ref.html

[30] "Hackbench Ubuntu Manuals,"[Online]. Available:http://manpages.ubuntu.com/manpages/precise/man8/hackbench.8.html