CEUR-WS.org/Vol-2135/SEIM_2018_paper_11.pdf

Design and testing of LXC-based virtualization system for resource-constrained MIPS devices

Maxim Menshchikov
Department of Applied Mathematics and Control Processes
Saint Petersburg State University
Saint Petersburg, Russia
maximmenshchikov@gmail.com

***Abstract*—Container-based virtualization became a standard de-facto for many environments, but it's applicability to resource-constrained systems is not as trivial question as it seems to be at first glance. Despite many projects aimed at bringing lightweight containers, such as Linux Containers (LXC), Docker, Kubernetes, etc, there are numerous problems in real-world development that might dramatically increase the impact on resources or render device unstable. In the paper we focus on Linux-based resource-constrained systems, studying the impact of LXC on MIPS CPU. Problems observed during the implementation of our system were studied and solutions for some of them, specifically slab overflow, provided. Our interpretation of obtained CPU, file system and other measures implies a negligible difference in comparison between host and isolated environment with only TCP performance reduced by up to 8-9%.**

***Index Terms*—virtualization, containers, lxc, linux, resource-constrained devices, embedded devices, internet of things, mips.**

I. Introduction

Embedded devices are used worldwide. They include routers, IoT/Smart Home devices, smartphones, players, set-top boxes etc. The way to add functionality to such devices hasn't changed over years: there is always a possibility to write a new module and build a firmware with it. It is assumed that such module is written by firmware developer and therefore a *right* place is taken in internal device infrastructure: it doesn't affect stability, performance, and security. The latter statement can only be taken with a pinch of salt: how can developers guarantee that there are no unforeseen outcomes of such functionality?

Mentioned product release schema is quite common, but what if software company decides to delegate some functionality to another company? Trusted partners can make a package nearly independent upon the generic firmware. Printing server (such as **CUPS**[1]), File server (e.g. **Samba**[2]) serve as good examples of almost independent components. However, vulnerabilities such as *SambaCry*[3] (a Samba vulnerability discovered in 2017) leave device open to new threats just due to *external* software.

Virtualization helps avoid these risks. Resource-constrained devices usually have too little raw performance to run a full-fledged virtualization and *containers* have a remarkable advantage over traditional virtualization methods. Container word relates to the environment made of multiple technologies inside the operating system kernel: PID namespaces [1], user namespaces [2], network namespaces [3], mount namespaces, etc. It is an isolated environment in which applications that are running inside are locked from accessing external resources unless explicitly or implicitly allowed doing so. Containers are running on top of main system kernel heavily exploiting isolation provided by it, which implies better performance and lower resource consumption.

However, even lightweight virtualization provided by kernel has serious limitations and consequences which must be addressed. In this paper we try to build a package virtualization supporting system and study performance, stability, security, networking and other effects of LXC (Linux Containers) virtualization on a specific MIPS-based embedded device and describe found problems with solutions to them.

A. Existing solutions

The first and the most obvious solution is Docker[4]. It is performing very well on desktop-class operating systems and more widespread embedded devices. Main disadvantages are in its binary size, the tendency to use Go runtime and compilers (that raises big questions about used toolchains, binary size, "maturity" of compilers) and its official incompatibility with MIPS processors.

Project **balena**[5] takes Moby project (the foundation for Docker) and tries to improve its weakest points. It features smaller single binary, multi-architecture support and more effective use of network bandwidth. The latter is quite exceptional and interesting, but suggested binary size is still quite big. MIPS architecture is also not supported officially.

If full-fledged virtualization is taken into account, then it is reasonable to check officially recommended [4] hypervisors for MIPS, such as prplHypervisor [5] (ex. Hellfire), SELTECH Fexerox [6], OmniShield [7]. However, these solutions tend to require more RAM, disk space and computational power for running isolated kernel alongside with operating system files.

[1]https://www.cups.org
[2]https://www.samba.org
[3]https://www.samba.org/samba/security/CVE-2017-7494.html
[4]https://www.docker.com
[5]https://github.com/resin-os/balena

B. Motivation

Existing solutions are more focused on more resource-rich desktop-class platforms. Our main intention is a creation of a system maintaining low overhead for boards used for more specific purposes and therefore lacking resources. We would like to run third-party packages of varying complexity while having those applications integrate to the device platform in a natural way. At the same time, the system we build is quite minimalistic and straight-forward, which enables quicker development of features.

While we test only MIPS devices, our expectation is that overhead measurement result will be quite generic among recent CPUs of various architectures. Still it requires an additional verification.

C. Positioning

Our system is positioned at a higher level than LXC itself since it deals with its setup rather than internal details. It shares the level with Docker/balena, which are far more user-friendly applications, but unlike **containerd**-based applications, we don't necessarily set up namespaces by our own means besides quotas and access restriction details. Virtualization by hardware or operating system level hypervisors is definitely on a lower level than our system.

II. System profile

The key components of our system are:

1) CPU: MIPS architecture 600 MHz dual core (in CPU parallel test similar single core CPU was used to ensure no core load differences).
2) Kernel: Non-preemptive SMP Linux (exact version can't be disclosed).
3) RAM: 256MB.
4) ROM: 128MB.
5) 2.4GHz 802.11bgn [8] and 5GHz 802.11ac [9] Wi-Fi antennas, traffic offloading hardware, cryptography acceleration, USB ports.

III. Classification of points of interest

1) **Kernel**. Operating system kernel is a key component for OS-level virtualization provider. We checked how it may generally prevent developers from developing embedded solutions running containers.
2) **Latency**. A device has to remain responsive no matter how many applications are running and how much do they exploit the CPU. Third-party applications *mustn't* spoil the main functionality.
3) **Raw performance**. Containers *shouldn't* suffer from decreased performance to extent set by quotas.
4) **Network performance**. Download and upload speeds shouldn't reduce significantly compared to host environment.
5) **RAM usage**. Main device's activity shouldn't be spoiled dramatically by the memory-consuming applications.
6) **File system**. A device should handle a reasonable number of containers with bound packages. There must be a possibility to avoid redundant copies of container files.
7) **Fault tolerance**. A device must be tolerant to boot failures and must be able to survive major panic situations.
8) **Security problems (throughout the paper)**. Containers shouldn't affect host's security.
9) **Board temperature**. Containers shouldn't have any influence on board temperature, nor they should lead to overheating. We consider it dependent upon processor load and power consumption. Since there is no intention to perform any serious computations on our CPU, board temperature wasn't a target of our research.
10) **Power consumption**. Use of applications in containers shouldn't lead to increased power consumption compared to runs within host namespace.
11) **Container activation time** This criterion is about time to start the container, which might be crucial in case of deeply distributed real-time computations. Our use case didn't imply any strict requirement for it, so it wasn't explored much. It will be indeed mentioned in file system overview.

In the next section we present the solutions on these aspects, show observed issues and suggest solutions.

IV. Aspects and solutions

A. Kernel

1) Embedded devices often can't do their job without hardware support: Operating system kernel has the largest influence on performance, stability, memory usage and other aspects. The main problem with embedded devices is in their dependency upon hardware supplier. Many manufacturers provide their own kernel for System-on-a-Chip (SoC) or some special hardware. With resource-constrained devices the situation is even more serious: often the device has *no technical possibility* to do its job without the help of hardware. In our case networking and cryptography accelerators required deep changes throughout the kernel to support shorter packet flow paths altering networking and netfilter cores very heavily. Justifying the need, CPU alone would have provided much less network bandwidth. In our experiments, 5GHz Wi-Fi antenna supporting IEEE802.11ac standard [9] was generally providing gigabit speeds, but could only demonstrate 40 mbit/s without overflowing the slab caches (we'll add on this problem later in RAM section). Summing up, resource-constrained embedded devices not only depend on the kernel for performance and correct operability but also do not have any workarounds if driver functionality does not work as expected.

2) Embedded devices have kernel version lag due to hardware supply: Generally, there is nothing to worry about regarding virtualization if patched kernel is based on recent enough kernel and tested properly, however, in our case the kernel was many versions behind the mainline: it was using not the latest LTS kernel, but LTS with far enough End Of Life time without any possibility to upgrade. Consequently, any desire to use container functionality was hitting at rough implementation in the kernel. To be specific, this version didn't have all required bits of user namespace support, therefore the UID/GID mapping was largely unsupported, making containers "privileged". Such containers have 'root' user identical to the 'root' outside the container, and while it had both advantages and disadvantages, it involved changing the policy for packages to *"trust only in specific cases"*. Of course, making a package out of *Samba* and other less-than-trusted software was no longer an option until the update from a hardware supplier. Migrating to the latest LTS was not an option as well due to development time constraints. We must also point out that the similar problem is common with Android smartphones [10].

B. Raw performance

Raw performance is worth a special check on resource-constrained embedded devices, a significant number of which stick to low-end CPUs. The wide range of provided services might require a reasonable computational power: VPN services using packet encryption and decryption without hardware cryptography acceleration, HTTPS servers (the most popular use case in our opinion), and quite a few other possible needs.

1) CPU quotas: CPU limits set up using cgroups, allowing containers to consume up to 50% of CPU, yet if responses from a main program were getting too rare, software watchdog was proactively decreasing the CPU share to some lower value until reasonable response time was achieved. This scheme has proven to be working well, however, a number of packages required almost none computational power, so the scheme wasn't widely adopted.

Our formula (1) is mathematically trivial. Consider R_n a normal response rate without additional load and R_c to be a current rate (both in *responses per second*). The upper threshold T_u for containers CPU share was 0.5 (which corresponds to 50% of CPU time and to I_n response interval), the lower threshold T_l was 0.05 (5% of CPU time).

$$s = min(T_l + \frac{R_c}{R_n} \cdot (T_u - T_l), T_u) \tag{1}$$

Without min, s can potentially grow more than T_u, which never happened in our experience, but in our testing 50% was always a reasonable upper limit to keep main device's activity running well.

2) Benchmarking methodology: **nbench** (the BYTE Magazine's BYTEmark) was used for benchmarking. When searching for properly supported utility, other utilities such as **sysbench** were tested, but most of them were problematic to compile on MIPS due to extensive use of assembler inlines. We consider **nbench** utility a reasonable choice because of its good cross compilation ability and its focus on single-threaded raw performance tests which fit our use case well.

CPU limits were disabled and the **nbench** executed 100 times in a row. The device was rebooted between tests.

Tests of CPU performance when running multiple benchmark processes were performed, but our prior investigation had shown that **nbench** test isn't much influenced by the number of processes (this fact is indeed good for the sanity of the first test). It was decided to measure time to complete the test. Two or mote processes were run in a row, an average time was recorded. To ensure no core load differences, another single core board with similar characteristics was used. The same software package was flashed to it.

C. Latency

Latency is the "interval between stimulation and response". When evaluating operating system latency, we consider latency two-fold: scheduler latency and interrupt latency [11].

Scheduler latency is a time interval taken by an operating system before performing scheduling. Scheduler's response delay is based on two main parameters: minimal granularity and latency. Minimal granularity is a minimal time given to a process to execute. Latency is a period in which all tasks should run at least once. Consider T a number of executable tasks, L is a defined latency, G is a minimal granularity. Therefore scheduler period is based on the following formula.

$$Period = \begin{cases} L & \text{if } T < \frac{L}{G} \\ T \cdot G & \text{if } T \geq \frac{L}{G} \end{cases}$$

However, running non-preemptive kernel adds some complexities to the scheduler. The kernel cannot interrupt the process executing a kernel call until the return to user mode [11]. Therefore the absence of preemption brings additional security consideration: there *must not* be any serious way to interact with kernel besides regular kernel API. For example, if usage of some device driver can lead to infinite loops or long computations, such driver should not be allowed. **libusb** was determined to be causing a loop when using **DWC2** (DesignWare USB2 Core driver) driver[6]. The latter was actually fixed in the upstream kernel.

Also supplied kernel might have binaries precompiled for a preemptive kernel, but even if provided modules are recompilable, they still might have interlocking errors like *missing spin lock* or *dead lock*. They might render the kernel unstable in preemptive mode. So we had to stick with non-preemptive mode and provided kernel. Instead, **libseccomp**[7] was used in order to close unneeded kernel API and **AppArmor** [12] helped restrict access to all devices except really needed ones.

Interrupt latency is constructed of the time elapsed since the interrupt is first fired and interrupt is started servicing. The main contributors to this latency type are drivers disabling interrupts for a long time. It means containers generally contribute to interrupt latency in exactly the same way as the host does, mostly by initiating driver activity via networking, peripheral or any other kind of access. We haven't compared the difference between containers and host because there is almost no differentiation between namespaces on driver level.

[6]DWC2 source code: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/usb/dwc2

[7]https://github.com/seccomp

D. Networking

1) Virtualization approaches: There are multiple ways to get access to the network within the container. Methods were generalized to avoid discussing unnecessary details.

- **Shared networking**. Applications in container deal with the same set of network interfaces, routes, and rules. In certain circumstances, container can control device from networking stack. Guest can create netfilter rules to redirect traffic intended for another application or take control over internal APIs by manipulating IP addresses. Depending on the purpose, it might be desired or undesired behavior, but a right practice would be to avoid such networking type. To name few, one working scheme to take over the device is to perform Man-in-the-Middle [13] (MITM) attack against secure management protocol unless certificate pinning is used. Another possibility coming from MITM is an interception of HTTP/HTTPS Web Interface traffic involving stealing of end-user login/password.
- **Virtual Ethernet (VETH)** [14] devices might have performance penalty mostly due to implementation-specific problems of network namespaces. In our tests containers take a range of private IP addresses (such as **192.168.2.0/24**) and DHCP server manages IP leasing, although for rarely changing environment static IP approach can be even more effective.
  Hardware traffic processing engines often have fast paths for forwarded traffic, yet for Virtual Ethernet it will still go through the main CPU, which will dramatically decrease possible performance.

2) Benchmarking methodology: The overhead of containers networking was measured accordingly. The existence of overhead for containers and host in shared networking scenario was checked. The test had been performed by using Raspberry Pi 3 connected to our MIPS device by Ethernet cable. **iperf** was used in server mode on Pi side and in client mode on a device side. Shared networking setup for both host and guest is demonstrated in Fig. 1.

For isolated networking, setup was harder to accomplish. The main idea of whole VETH setup is to have a separate networking namespace, so we couldn't just repeat the first scenario. With VETH interface added to the same bridge as the generic LAN interface (Fig. 2) results were recorded again. However, in our case packet acceleration hardware, more specifically its driver, effectively broke TCP communication for the scenario. Once driver had been disabled we measured performance again, even though the host side was left essentially the same as in shared network test.

For TCP the following commands were issued:

```
iperf -s -p 5000 #server
iperf -c 192.168.2.3 -p 5000 -t 30 #client
```

For UDP, the bandwidth parameter was reduced down to 100 mbit/s, and socket size set to 10000KB, automatically limited to 352KB. There was 5-second interval between attempts to avoid band overflow. Only server's report was actually evaluated.

```
iperf -s -p 5000 -u -w 10000KB #server
iperf -c 192.168.2.3 -p 5000 -u -t 30 \
      -b 100MB -w 10000KB #client
```

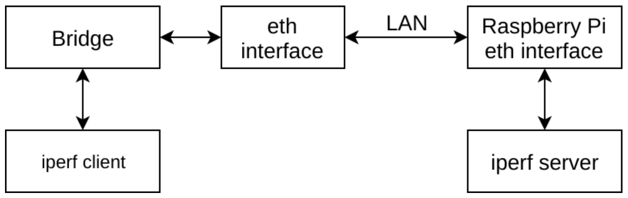

Fig. 1. Shared networking setup

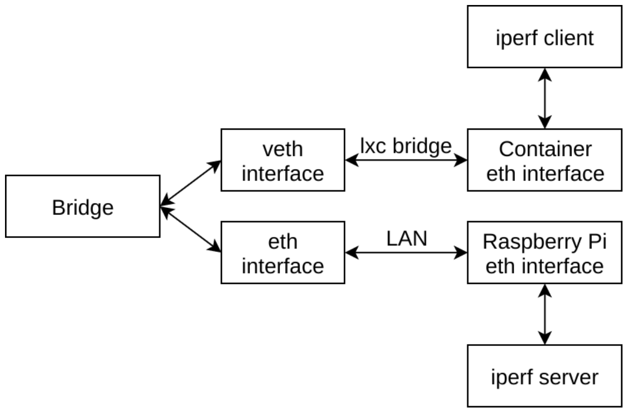

Fig. 2. Isolated networking setup

E. RAM usage

During experiment RAM usage was generally good and to some extent couldn't be any better. The process running in container consumed exactly the same amount of memory as the one which was running outside the container. However, measuring overhead in a kernel space is an additional problem which requires a separate study.

1) RAM quotas: One of the interesting questions was what the system would do when the generic RAM usage would have risen above the bar so that it wouldn't be possible to reclaim any memory for host OS. RAM limit through **cgroups** was generally a good way to ensure stability of the host in such situation. Unlike CPU, RAM quota wasn't something we could change dynamically as swap partition was unused. If some container would have used e.g. 30% of all RAM, we wouldn't have any way to squeeze it to 15% in the runtime. The solution we used was simple: leave 30% of RAM to containers. However, if generic free RAM counter would go below some threshold (e.g. 30MB) for some reason, the kill command was issued to the process in a container using the most of memory. Obviously, this scheme is only correct when containers aren't ought to do any sensitive work. As our applications were using a fair minimum of memory, this quota rule wasn't widely adopted.

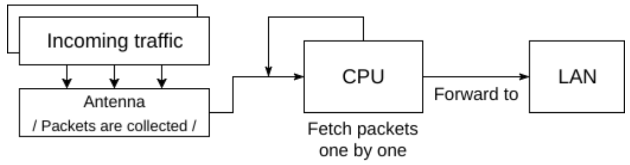

Fig. 3. Slab exploitation on non-preemptive kernel

2) Slab injection: In a quest for predictable and stable operability of a device, yet another issue was faced. Linux kernel uses slab subsystem [15] to decrease memory fragmentation when allocating objects of the same size. The outcome of it is barely visible, but if container gets a possibility to cause a kernel to allocate virtual memory, it might get problematic because the kernel cannot reclaim memory for slab in atomic context (reclaiming is done for cached pages — those used for file caching or other purposes yet essentially free). Drivers sending a lot of packets in short time frames might be unable to allocate memory, and further attempts to do so will lead to hardware queue growth. One of such examples was found in the Wi-Fi driver which under certain circumstances, such as build configuration or due to technical need, could be unable to pass traffic through fast path, and would have had to allocate regular sockets (note that IEEE802.11ac [9] standard extends the bandwidth to at least 433mbps ≈ 54MB/s). The exploitation method is shown in Fig. 3.

It was investigated that having a lot of empty slabs was barely possible to achieve. Virtual memory tuning through procfs means, such as $/proc/sys/vm/min\_free\_kbytes$ [16] and other entries, didn't have any influence — Wi-Fi driver could still overfill buffers and get caught in an infinite loop, which would have caused system reboot after watchdog detected the situation. Our solution included a technique called **slab injection**, which was used to fulfill caches with objects of the same size, selectively leaving one of the objects per page. This resulted in increased capacity for selected types of objects which couldn't be gracefully controlled any other way.

Slab injection must be performed in the kernel mode. Consider s an object size, p to be page size. To make the algorithm more precise, it is desired to know the number of $active$ and the $total$ number of objects in the cache ($total - active$ shows the number of free objects within existing slabs which can make algorithm ineffective if real $active \neq 0$ and $total \neq 0$). The method is shown in Fig. 4. We have also created a GitHub project for it[8].

This technique was proven to be working well on small size objects, which were of primary interest, however it *couldn't inject pages to slab caches with object size close to P, i.e. when page can only fit one object of that size*. For such cases a patch disabling shrinking for certain user-defined caches was created.

RAM (and to some extent ROM) usage was actually one of reasons why **Docker** hadn't been used. At the moment of writing Docker was a multi-binary application written in Go.

[8]https://github.com/mmenshchikov/slab_inject

```
function PREFILLCACHE($cache, active, total$)
    $c \leftarrow 1$
    for $i \leftarrow active, total$ do
        $O_c \leftarrow$ KMEM_CACHE_ALLOC($cache$)
        $c \leftarrow c + 1$
    end for
    return $O$
end function
procedure FREEPREFILLED($cache, O$)
    $c \leftarrow 1$
    while $\exists O_c$ do
        KMEM_CACHE_FREE($cache$, $O_c$)
        $c \leftarrow c + 1$
    end while
end procedure
procedure INJECTPAGE($cache, active, total, s, p$)
    $P \leftarrow$ PREFILLCACHE($cache, active, total$)
    $n \leftarrow p/s$
    for $i \leftarrow 1, n$ do
        $O_i \leftarrow$ KMEM_CACHE_ALLOC($cache$)
    end for
    for $i \leftarrow 1, n - 1$ do
        KFREE($O_i$)
    end for
    FREEPREFILLED(cache, P)
end procedure
```

Fig. 4. Slab injection algorithm

Docker didn't support MIPS in its build system and source files. While the build support was fixed (it involved rewriting of make files, adding **.mips.go** files, adding 32 bit support whenever applicable, and fixing gccgo/go selector to choose target compiler more properly), shared libraries couldn't be generated, so we ended up with Docker binaries bigger than flash size. In our case, even if shared libraries could work, Go runtime potentially could still be too big, leaving too large working set for the running process. That were the reasons Docker porting had been stopped.

F. Fault tolerance and debugging

The important part of any consumer device is a failure tolerance. While our main software goes through very intensive testing proactively detecting both kernel crashes and userland problems, third-party packages might be less tested and, what's even harder to check, built without proper knowledge of hardware- or software-specific weaknesses leading to kernel panics.

Our measures against runtime loops employ a *watchdog* which operates at kernel and userland levels and proactively reboots the device if any long-standing lockup is detected.

The other measure is a *safe mode*. If a device fails booting for 5 times, the partition with external packages is not loaded at all.

In all failure cases, the software collects configuration, core files, and logs, compresses them to a single archive and makes

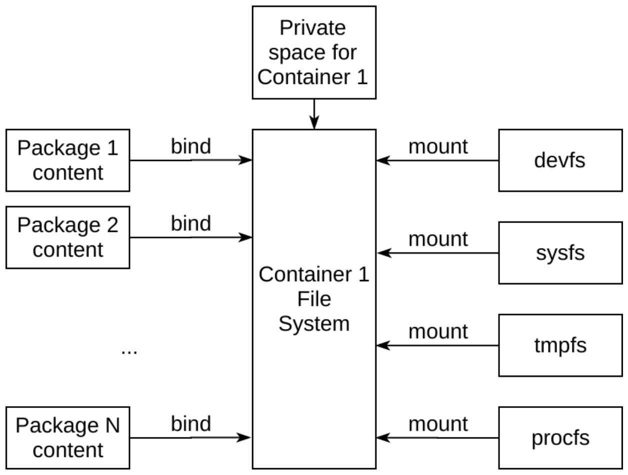

Fig. 5. Container file system sources

it available for an investigation through service provider's means.

G. File system

1) Single repository policy: Lack of disk space was the largest problem in our system. The task was to get a file system (FS) configurator spawning FS with the following properties:

- Consistent.
- Allows no duplicates.
- Free of "DLL hell" and versioning problems.

Following qualities can only be achieved by using a single repository policy. The software can not actually receive a new software package without getting update availability status. Once the new software is available and loaded, it is assumed that the package is compatible with other packages installed to the system. In our case, the presence of the Internet on the device was essential to functionality and user was not given an ability to install packages without it, so the system was quite stable in that aspect.

2) System file security: Another important point is how to deal with system files: there *must be an ability to load additional packages with libraries*, yet *packages must have no capability to write to files they don't own*. A read-only bind mount sounds like an adequate response to the problem.

Container file system is built from read-only bind mounts of package content (allowing containers share packages), generic file systems such as **devfs**, **procfs**, **sysfs** and **tmpfs**. Whole container file system is physically located in a separate folder inside package partition, so free space is directly taken from it. The scheme of container file system sources is shown on the Fig. 5. No problem had been found maintaining such a system, nor any sign of security problems it may bring (besides the case when such mount is read/write) was detected.

3) Performance and benchmarking methodology: Our file system is an **OverlayFS** [17] built on top of **squashfs** [18] (read-only part) and **ubifs** [19]. The **dd** utility was used to measure the impact of containers on file system performance:

```
for i in `seq 1 100` ; do
/tmp/busybox dd if=/dev/zero     \
    of=/test/file bs=4096        \
    count=10000 2> write$i.txt
done
```

The following script was used for testing read performance.

```
for i in `seq 1 100` ; do
echo 3 > /proc/sys/vm/drop_caches
/tmp/busybox dd if=/test/file    \
    of=/dev/null bs=4096         \
    count=10000 2> read$i.txt
done
```

Cache dropping was used to prevent the operating system from filling file system caches, significantly improving performance in consequent tests.

4) Container activation time: When the container is not enabled and some application is about to start inside it, activation is fairly quick: it is essentially the same as few **mount** calls and **fork** with **exec**. However, if the container is running and some package is ought to be bound to it, the only option is to use *shared folders* to transfer bind mounts from host to container. This process is indeed not that trivial, big packages required up to few seconds to bind. Fortunately, the whole scheme in which container can't be shared among packages however greatly minimizes the possibility of such a long delay down to zero.

In case of Docker that time could be seriously different: as docker images tend to include operating system templates, mount time would be less proper.

H. Container API

Software packages should have an ability to perform system tasks such as "stop container" request, **cron**-like application start management, listing of computers in LAN and so on.

With container networking it could become troublesome to distinguish the container from others and define access control rules. Consider two use cases.

- *Shared networking* was determined to provide no real possibility to determine the container, granted that container always has a possibility to change its network configuration. Spoofing at container's side is generally too trivial for real-world usage, it can be done through **iptables** or **ip**.
- *Isolated networking* (through **Virtual Ethernet (VETH)** devices) provides a full-fledged IP management. With VETH the determination of IP spoofing within the container is trivial just by comparing of IP addresses within the container with predefined/leased value and specific **netfilter** rules. Any newly created networking interface won't let the traffic flow outside the container.

Consequently, due to the usage of isolated networking, the container API server had been set up to listen at specific VETH interface.

TABLE I
CPU PERFORMANCE MEASURES FOR HOST AND GUEST

Test	*Avg.Host*	*Avg.Guest*	Diff.	Diff. (%)
Numeric Sort	157.8933	157.9457	−0.0524	−0.0333
String Sort	6.1465	6.1243	0.0222	0.3618
Bitfield	50020270	50159090	−138820	−0.2775
FP Emulation	39.7115	39.7496	−0.0381	−0.096
Fourier	51.0747	51.1460	−0.0713	−0.1395
Assignment	2.8199	2.8622	−0.0423	−1.5
IDEA	755.0928	753.0582	2.0346	0.2695
Huffman	171.1702	171.0864	0.0838	0.049
Neural Net	0.0635	0.0636	−0.000095	−0.1495
LU Decomp.	2.0483	2.0388	0.0095	0.4616

TABLE II
TEST TIME MEASURES IN PARALLEL MODE

Test	*Host (sec)*	*Guest (sec)*	Diff. (sec)	Diff. (%)
1 process	333.59	335.800	−2.210	−0.662
2 processes	669.86	668.345	1.515	0.226
4 processes	1338.323	1337.288	1.035	0.077
8 processes	2676.929	2681.940	−5.011	−0.187

I. Power consumption

Power consumption was measured during parallel runs of **nbench** utility for both dual core and single core variants of MIPS CPU. The biggest power consumers such as Wi-Fi antenna were disabled, power consumption in an idle state was measured. Absolute watt numbers are irrelevant: since our hardware is not publicly available, it can't serve as a baseline for future tests except ours, the difference between power consumption in an active state and the idle state is provided. Of course, since idle state boundary might be floating, it was measured before every test.

V. RESULTS

A. CPU performance

Average results (calculated using bytemark_counter utility[9]) are shown in Table I and Table II. All results were uploaded to GitHub[10] for convenient interpretation.

In 60% of tests the guest is faster than the host (we give our explanation of this fact later). One test shows difference more than 1% (1.5%), while in 90% of tests measures in absolute numbers are well lower than 0.5%. 30% of numbers are even lower than 0.3. The maximum difference is 1.5%, it is reached in assignment test.

In the second test results seem to be more sporadic, ranging from 0.662% win of the host over the guest to 0.226% win of the guest. It is notable that there is no generic trend.

B. Network performance

Raw results had been uploaded to GitHub[11]. Average results are shown in Table III.

[9]https://github.com/mmenshchikov/bytemark_counter

[10]https://github.com/mmenshchikov/bytemark_counter_results

[11]https://github.com/mmenshchikov/lxc_iperf_comparison

TABLE III
NETWORK PERFORMANCE MEASURES FOR HOST AND GUEST

Test	Network	*Avg.Host (mbit/s)*	*Avg.Guest (mbit/s)*
iperf (TCP)	Shared	93.26	85.14
iperf (TCP)	Isolated	86.75	79.64
iperf (UDP)	Shared	94.92	95.22
iperf (UDP)	Isolated	93.60	94.17

TABLE IV
FILE SYSTEM PERFORMANCE MEASURES FOR HOST AND GUEST

Test	*Avg.Host (MB/s)*	*Avg.Guest (MB/s)*
dd write (no sync)	18.504	18.634
dd write (sync)	0.897	0.919
dd read (dropped cache)	7.241	7.207

TABLE V
POWER CONSUMPTION MEASURES FOR HOST AND GUEST

Test	*Host (Watts, diff.)*	*Guest (Watts, diff.)*
nbench (1 process)	−0.1 / 0.3	−0.1 / 0.3
nbench (2 processes)	−0.1 / 1.2	−0.1 / 1.2
nbench (4 processes)	−0.1 / 1.2	−0.1 / 1.2
nbench (8 process)	−0.1 / 1.2	−0.1 / 1.2
iperf	0.8 / 0.9	0.8 / 0.9
iperf+nbench	0.6 / 0.6	—

TCP tests for the guest show 91.29% and 91.8% of host's performance. In UDP tests host gets 99.68%/99.39% of guest's speed. While not outlined, the impact on CPU was comparable.

C. File system performance

Average results[12] are demonstrated in Table IV.

The difference is low: 0.13 MB/s, 0.022 MB/s and 0.034 MB/s. Therefore the maximum difference is about 130 KB/s (which is alone not quite small, but incomparable to generic performance).

D. Power consumption

Average results are shown in Table V. Note that we show results for single core and dual core counterparts in form **single core result / dual core result**. **iperf** was tested in both shared and isolated modes, but results were mixed because they don't differ.

Overall, no measurable difference in power consumption between host and guest was found. Two and more processes indeed had shown the rise of dual core processor consumption by 0.9W.

VI. DISCUSSION

A. Benchmark

Our interpretation of results led us to a conclusion that the impact of container-based virtualization is low. Raw CPU performance is better for a guest than for a host, which contradicts to common sense (although the similar statistics

[12]https://github.com/mmenshchikov/lxc_dd_comparison

is provided in another research [20]). In our understanding, it is scheduling which is to blame since scheduler's choice of quantum might substantially differ for the process in another PID namespace, and we tend to believe the implicit CPU share between namespaces is quite fair. Another possible reason might come from a difference between device runs, as the one added by varying service start time, but it shouldn't have any serious impact.

While generally incomparable, **dd** write test (unsynchronized) outperforms read test solely due to *writeback* mechanism. A synchronized test is slower due to *writethrough*, the direct write to flash. The filesystem cache was dropped before every reading test, so the result can be considered "pure". Since the difference is too small, we assume that containers don't add any serious overhead for the file system in our use case.

As for network performance, UDP protocol tests demonstrate guest outperforming host — it looks like an effect of scheduling. At the same time TCP protocol is generally more consuming and it adds some overhead to container environment (about 8-9% of host speed).

Power consumption for host and containers was equal. It didn't come as surprise, our theoretical understanding also implied this. It is very notable that for single core processor the power consumption is even less for **nbench** testing. Scheduling and different computational power needed for specific purposes seem to be the reasons for it. Dual core processor faced the rise of power consumption when second core woke up. In general, this result shows there is no any difference in energy consumption between host and lightweight containers, granted they don't bring any additional computational costs besides ones introduced by kernel namespaces.

Overall we can conclude that advantages of containers overweigh negative impact of additional namespaces in the kernel. File system performance and raw CPU performance are almost unaffected by it. Network performance went down, however, 8-9% is a reasonable price for extra security and comfort. Power consumption doesn't rise. Therefore Linux Containers technology is mature enough for use on resources constrained devices according to our tests.

Generic latency, network latency, temperature and other important points were overlooked due to low interest. Those are definitely worth examination in the future.

B. System design

The suggested system design works well for our use cases, however, there are problems in each core aspect which might be crucial for specific purposes. The system is not well-protected against kernel lockups: maintainers must be extra careful when allowing specific kernel APIs or access to kernel drivers. That's mostly vendor-specific limitation than the design flaw. CPU quotas are operating properly, but a stricter control over container's quotas might be needed: otherwise limits might get reduced too much based on a very imprecise measure of "application response time". At the same time, RAM quotas aren't flexible at all, and the solution for this problem is unclear.

File system architecture is providing low overhead (as seen in tests), however, it is still important to audit **overlayfs** for possible **inode** problems [21]: there is a strong need to know it for sure if there is an intention to build reliable platform.

Container API needs strict enforcement of IP addresses inside the container. Our system doesn't have an access restriction for certain containers, so this is again not a problem. If it was the case, we could have employed some other mechanism for communication, e.g. UNIX sockets.

VII. Related work

The container-based virtualization was first explored with [22] work, continued to embedded devices with Cells [23], the virtualization system for Android smartphones.

Service-hosting gateways [24] were one of the first attempts to define requirements and build a proof-of-concept of a more constrained device with containers running. A. Krylovskiy [20] examines containers on Raspberry Pi, the embedded device built upon ARM CPU. We expand the topic further by using MIPS CPU with even fewer resources available. The notable thing is that a lot of studies focus on proof-of-concepts, while our approach was tested in production. One of the other examples of a system used in real-world applications is resinOS [25], but it is still tailored for more resource-rich devices, most notably due to more RAM and disk space.

ParaDrop [26] is a multi-tenant platform for third-party services on wireless gateways. While ParaDrop's goal is to "push services from data centers to the network edge" [27], our goal differs in a way to allow partners further extend device's software package with the ease and adequate level of security.

Paper [28] provides a high-level overview of lightweight container problematics on IoT devices, which we partially cover.

The framework for block-streaming application execution [29] demonstrates a method of loading software binary chunks to resource-constrained hardware. Our device is not that limited by RAM and ROM, it is running Linux and full-fledged executable files, yet it provides a foundation for even more constrained devices.

Aimed at performance, another research [30] is concentrating on Raspberry Pi and Odroid (ARM) single-board computers running Docker images. Generally, our study differs from it as we are not focused on performance but rather building a system from scratch. Also, all tested systems are considerably faster and have a lot more built-in RAM. Obtained results are indeed similar, showing a negligible impact of container virtualization on these systems.

As for full-fledged virtualization, many efforts [31]–[33] had been put into making it work for embedded MIPS CPUs in various ways. Moratelli, Zampiva, and Hessel [31] presented an embedded hypervisor for real-time execution of applications. Low overhead was observed for single processor platforms, and a certain penalty detected for multiprocessor systems. There were updates to the research, e.g. [34], dealing with hardware-assisted interrupt delivery. The approach looks promising, but just as any virtualization requires slightly more disk space and computational power. KVM-Loongson [33] authors develop processors based upon MIPS with an efficient hypervisor, with only I/O being a serious bottleneck. Still it is a hardware solution, which is not what most users can integrate. Hypervisor OmniShield [7] is CPU vendor-backed, so it is assumed to have a good support in recent MIPS processors. Also, it brings improvements to hardware-based domain isolation. All it makes it a good alternative once sufficient resources are present.

VIII. Conclusion and future work

We managed to create a system for running containers on resource-constrained embedded devices. Working design for different aspects was suggested. Major problems found during the development process, namely slab overflow, packet acceleration engine troubles, container API considerations, were explained. We compared the performance of host and containers and found the difference in most of the tests quite negligible with host winning only in TCP networking tests. Speed test result is confirming that LXC environment is ready for container-based third-party applications. Power consumption measurement doesn't show any difference between containerized and native environments.

Our future work would be concentrated on further examination of container capabilities on embedded devices, as well as the practical work towards improving disk space and RAM usage. Additionally, we have an intention to make a security audit of the described system.

References

[1] "Namespaces in operation, part 3: PID namespaces," (Last accessed 12-April-2018). [Online]. Available: https://lwn.net/Articles/531419

[2] "Namespaces in operation, part 5: User namespaces," (Last accessed 12-April-2018). [Online]. Available: https://lwn.net/Articles/532593

[3] "Namespaces in operation, part 7: Network namespaces," (Last accessed 12-April-2018). [Online]. Available: https://lwn.net/Articles/580893

[4] "The MIPS architecture and virtualization," (Last accessed 12-April-2018). [Online]. Available: https://www.mips.com/blog/the-mips-architecture-and-virtualization

[5] A. Aguiar and F. Hessel, "Virtual hellfire hypervisor: Extending hellfire framework for embedded virtualization support," in *2011 12th International Symposium on Quality Electronic Design*, March 2011, pp. 1–8.

[6] "SELTECH debuts new FEXEROX hypervisor for MIPS." [Online]. Available: https://www.mips.com/blog/seltech-debuts-new-fexerox-hypervisor-for-mips-mcus

[7] "Parallelization, virtualization, security for smart gateways and smart "things"." [Online]. Available: https://www.mips.com/blog/parallelization-virtualization-security-for-smart-gateways-and-smart-things

[8] M. Gast, *802.11n: A Survival Guide: Wi-Fi Above 100 Mbps*. O'Reilly Media, 2012.

[9] ——, *802.11ac: A Survival Guide: Wi-Fi at Gigabit and Beyond*. O'Reilly Media, 2013.

[10] "Ars Technica: Android execs get technical talking updates, Project Treble, Linux, and more," (Last accessed 12-April-2018). [Online]. Available: https://arstechnica.com/gadgets/2017/05/ars-talks-android-googlers-chat-about-project-treble-os-updates-and-linux

[12] M. Bauer, "Paranoid penguin: an introduction to Novell AppArmor," *Linux Journal*, vol. 2006, no. 148, p. 13, 2006.

[11] P. Raghavan, A. Lad, and S. Neelakandan, *Embedded Linux System Design and Development*. CRC Press, 2005.

[13] Y. Desmedt, "Man-in-the-middle attack," in *Encyclopedia of cryptography and security*. Springer, 2011, pp. 759–759.

[14] S. Subramanian and S. Voruganti, *Software-Defined Networking (SDN) with OpenStack*. Packt Publishing, 2016.

[15] B. Fitzgibbons, "The Linux slab allocator," 2000.

[16] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[17] "Overlay Filesystem," (Last accessed 12-April-2018). [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

[18] P. Lougher and R. Lougher, "SQUASHFS - a squashed read-only filesystem for Linux," 2006.

[19] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif, "Abstract specification of the UBIFS file system for flash memory," in *International Symposium on Formal Methods*. Springer, 2009, pp. 190–206.

[20] A. Krylovskiy, "Internet of things gateways meet linux containers: Performance evaluation and discussion," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Dec 2015, pp. 222–227.

[21] "OverlayFS and containers." [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/overlayfs-and-containers-vault-2017-miklos-vivek.pdf

[22] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 361–376, 2002.

[23] C. Dall, J. Andrus, A. Vant Hof, O. Laadan, and J. Nieh, "The design, implementation, and evaluation of cells: A virtual smartphone architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 3, p. 9, 2012.

[24] J. Whiteaker, F. Schneider, R. Teixeira, C. Diot, A. Soule, F. Picconi, and M. May, "Expanding home services with advanced gateways," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 37–43, 2012.

[25] "Introducing resinOS 2.0, the host OS to run containers on embedded devices," (Last accessed 12-April-2018). [Online]. Available: https://resin.io/blog/introducing-resinos/

[26] D. Willis, A. Dasgupta, and S. Banerjee, "Paradrop: a multi-tenant platform to dynamically install third party services on wireless gateways," in *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture*. ACM, 2014, pp. 43–48.

[27] P. Liu, D. Willis, and S. Banerjee, "ParaDrop: Enabling Lightweight Multi-tenancy at the Network's Extreme Edge," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct 2016, pp. 1–13.

[28] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, Jan 2018.

[29] X. Peng, J. Ren, L. She, D. Zhang, J. Li, and Y. Zhang, "Boat: A block-streaming app execution scheme for lightweight iot devices," *IEEE Internet of Things Journal*, vol. PP, no. 99, pp. 1–1, 2018.

[30] R. Morabito, "Virtualization on internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[31] C. Moratelli, S. Zampiva, and F. Hessel, "Full-virtualization on mips-based mpsocs embedded platforms with real-time support," in *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design*. ACM, 2014, p. 44.

[32] C. Moratelli, S. Johann, and F. Hessel, "Exploring embedded systems virtualization using mips virtualization module," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016, pp. 214–221.

[33] Y. Tai, W. Cai, Q. Liu, and G. Zhange, "Kvm-loongson: An efficient hypervisor on mips," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, July 2013, pp. 1016–1022.

[34] C. Moratelli, F. Hessel *et al.*, "Hardware-assisted interrupt delivery optimization for virtualized embedded platforms," in *Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 304–307.