"(c) 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works."

2015 IEEE 7th International Conference on Cloud Computing Technology and Science

IncludeOS: A minimal, resource efficient unikernel for cloud services

Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, Kyrre Begnum
Dept. of Computer Science
Oslo and Akershus University College of Applied Sciences
Oslo, Norway
alfred.bratterud@hioa.no

***Abstract*—The emergence of cloud computing as a ubiquitous platform for elastically scaling services has generated need and opportunity for new types of operating systems. A service that needs to be both elastic and resource efficient needs A) highly specialized components, and B) to run with minimal resource overhead. Classical general purpose operating systems designed for extensive hardware support are by design far from meeting these requirements.**

In this paper we present IncludeOS, a single tasking library operating system for cloud services, written from scratch in C++. Key features include: extremely small disk- and memory footprint, efficient asynchronous I/O, OS-library where only what your service needs gets included, and only one device driver by default (virtio).

As a test case a bootable disk image consisting of a simple DNS server with OS included is shown to require only 158 kb of disk space and to require 5-20% less CPU-time, depending on hardware, compared to the same binary running on Linux.

***Index Terms*—unikernel, library OS, full virtualization, virtio**

I. Introduction

While cloud computing is rapidly becoming a preferred platform for running services, a major concern is the increased energy consumption of cloud deployments worldwide. In reports by Green Peace from 2010 and 2014, the combined energy consumption of cloud computing surpassed countries such as Germany, Canada and Brazil [1], [2], making it the 6th larges energy consumer in the world. For many such deployments, much of the computing power is allocated to running virtual machines, which in turn are running general-purpose operating systems.

Todays major operating systems were designed to run a large number of programs in parallel, supporting a huge spectrum of different hardware devices ranging from Gigabit network interfaces to sound cards and last century printers. As a consequence, modern operating systems require a lot of disk- and memory for features the service it runs might not need. They also produce a steady flow of CPU and I/O usage. For example, for each Linux- or Windows VM, the hypervisor has to emulate the timer interrupt, causing the virtual machine to spend energy while doing nothing. This might seem negligible at first, but for elastic cloud services at scale, requiring a large number of virtual machines, it amounts to a significant resource waste and a serious limitation to the capacity of hypervisors [3].

In this paper we present IncludeOS, a single-tasking operating system designed for virtualized environments. IncludeOS provides a novel way for developers to build their C++-based code directly into a virtual machine at compile-time. The key contributions of IncludeOS are:

- **Extreme resource efficiency and footprint** IncludeOS will use minimal resources compared to standard operating systems. When idle, it uses no CPU at all. Only the parts of IncludeOS required for its single service gets included, which reduces waste and results in better network and memory performance.
- **Efficient deployment process** Using a custom GCC-based toolchain, writing `#include <os>` will literally Include the operating system. During link time the build-system will extract whatever the service needs from the pre-compiled OS-library and form a single binary. A boot sector is attached and it is all written to an image-file, resulting in a bootable disk image.

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.89

IEEE computer society

- **Virtualization platform independence**
  IncludeOS is written to run on virtualized x86 hardware. The resulting disk-image can be uploaded directly to openStack via the API or the web interface, or automatically formatted to fit most virtualization environments, such as VirtualBox.

The rest of the paper is organized as follows: Related work is described in Section 2, and the design and architecture of IncludeOS in section 3. Section 4 shows the resource usage and memory performance of IncludeOS compared to a standard operating system, and section 5 demonstrates the performance of a DNS-service running on IncludeOS, compared to the same binary running on Linux. Our results are discussed in Section 6 with conclusion and future work.

II. Related work

The idea of a "library operating system" dates at least back to the *exokernel* from 1995 [4], were the idea was that the applications in a multi process OS each provide their own abstractions over hardware. With the proliferation of virtualization technologies the idea of single tasking operating systems have regained relevance, and there has been a surge of activity to find the best adaptation for the context of cloud computing. A good overview of is provided by Madhavapeddy et. al. in [5], where they also present "Mirage OS", and dub this and other recent single-tasking OS'es *unikernels*. IncludeOS has many features in common with Mirage; it is single tasking, it draws only the required OS functionality from an OS library, and links these parts together with a service, to form a bootable virtual machine image, also called an *appliance*.

Common for some of the existing solutions is that they are designed for supporting a certain high-level language, such as OCaml in the case of Mirage, or Java in the case of OSv [6]. Other solutions are limited to *partial virtualization* relying on the host to provide abstractions, such as threads and drivers, in the case of BSD rump kernels [7], or directly sharing its kernel, such as with Docker [8]. Others again are highly specialized, such as ClickOS, which is designed specifically to run "Click-based middleboxes", i.e. the Click modular router [9].

III. IncludeOS architecture

A. The Zero-overhead principle

For any service designed to scale by deploying more virtual machines, it is crucial that each such machine incurs minimal resource overhead. In contrast to classical operating systems, which include as many features as possible, IncludeOS aims for true minimality in the sense that nothing should be included by default that the service does not explicitly need. This corresponds to the *zero overhead principle* of e.g. C++; "what you don't use you don't pay for" [10, p. 10].

B. Statically linked libraries and GCC-toolchain

The mechanism used for extracting only what is needed from the operating system, is the one provided by default by modern linkers. Each part of the OS is compiled into an object-file, such as `ip4.o`, `udp.o`, `pci_device.o` etc., which are then combined using `ar` to form a static library `os.a`. When a program links with this library, only what's necessary will automatically be extracted by the linker and end up in the final binary. To facilitate this build process a custom GCC-toolchain has been created.

IncludeOS does not have a program loader, so there is no classical `main`-function, with parameters and return value. Instead, a `Service`-class is provided, and the user is expected to implement `Service::start` which will be called after the OS has completed initialization.

C. Standard libraries

RedHat's newlib has been chosen as C standard library implementation primarily because it is small, and designed to rely on only a handful of system calls, and because it compiles into a statically linked library. This way the linker will again naturally include only the parts of the standard library actually used by either the OS components or the service itself, leaving out the rest.

The C++ standard library is larger and trickier. Since the STL containers rely heavily on exceptions we have chosen not to use these inside the kernel, but instead an exception-free variant by Electronic Arts, EASTL [11]. While this implementation contains the most important parts of STL such as string, streams, *vector* and *map*, it is not complete so several features have been implemented, and some are still in the pipeline. Our version of this library will be available to the service (i.e. userspace), but future work is likely to include a port of a full-featured implementation such as GCC's libstdc++.

D. Virtio network driver

IncludeOS currently has only one device driver, namely a VirtioNet Device driver. The key benefit of virtio is that the hypervisor does not need to emulate a certain physical device, but instead can insert data directly into a queue in memory shared by the guest. While Virtio 1.0 has recently emerged as an OASIS standard [12], none of the hypervisors used during development supported any of the new features. Therefore the driver currently only implements *Virtio Legacy* functionality, but development has been done with future support for Virtio 1.0 in mind. Our current implementation uses the PCI bus and has not enabled MSI-x (Message signaled interrupts).

E. Modular, object oriented network stack

While existing network stacks were considered to be ported into IncludeOS, other implementations are usually tightly entangled with the operating systems to which they belong, relying heavily on local conceptions of drivers, threads, modules etc. With the additional constraint of the *no overhead principle*, it becomes necessary to heavily modularize the stack so that services which, for instance, only require a single interface with IPv4 UDP, doesn't need to pay for the overhead of routing, TCP, IPv6 etc. Fig.1 shows the design of the current implementation of the IncludeOS network stack, as used in the DNS service described below. While it is not yet complete, it will respond as expected to ARP-requests, answer *ping* and, as demonstrated below, fully support a functioning DNS server. Key design decisions include:

1) Completely modularized design: The objects shown in Fig.1 are again encapsulated into an ''Inet''-class, but this is just a thin wrapper. The key idea is that many different such IP-stacks can be formed, reusing some of the objects, and replacing others, such as for instance a stack with a different link-layer protocol, a dual IPv4-IPv6 stack, an IPv6-only stack etc.

2) Delegates as connections between modules: We use ''The fastest possible C++ delegates''[1] as connections between modules. The delegates were tested and found to yield identical performance to plain C function pointers. Connecting two network modules via delegates only, essentially means that each object only knows about the function signature of one relevant member function (i.e. method) in the class it connects to. For example, the UDP class only knows that it has a pointer to a class called ''network layer'', on which it can call the ''transmit'' function to pass outbound packets down stream. In the current implementation, each module is compiled separately, and are instantiated and connected to each other at run-time by the *Inet*-object. This *Inet* class has full control over this connection and could instead decide to reroute the UDP output to any other object as needed.

Work is underway to completing TCP- and IPv6 support, and to make the stack completely standalone, runnable in Linux user space.

F. Asynchronous I/O and deferred IRQ

Currently, all IRQ handlers in IncludeOS will simply (atomically) update a counter, and defer further handling to the main event-loop, whenever there is time. This eliminates the need for a context switch, while also eliminating concurrency-related issues such as race conditions. The CPU is kept busy by having all I/O be asynchronous, so that no blocking occurs. This encourages a callback-based programming model, such as is common in modern Javascript applications.

IV. Resource footprint

A key benefit of the IncludeOS architecture is that only what the service actually needs gets included, thus minimizing the memory wasted on unused features. As shown in Fig. 2 it turns out that a fully virtualized Qemu-instance running a ''Hello world'' service *and the IncludeOS operating system* has a much smaller memory footprint than a *normal Java program* running on a PC (8.45 MB vs. 28.29 MB). In contrast, the minimal memory footprint of an Ubuntu 14.04 virtual machine, which is the reference operating

[1] Originally described by Don Clugston at CodeMonkeys (http://goo.gl/OW8g9P), with a C++11 update described on Stack Exchange code review, http://goo.gl/6TR44w

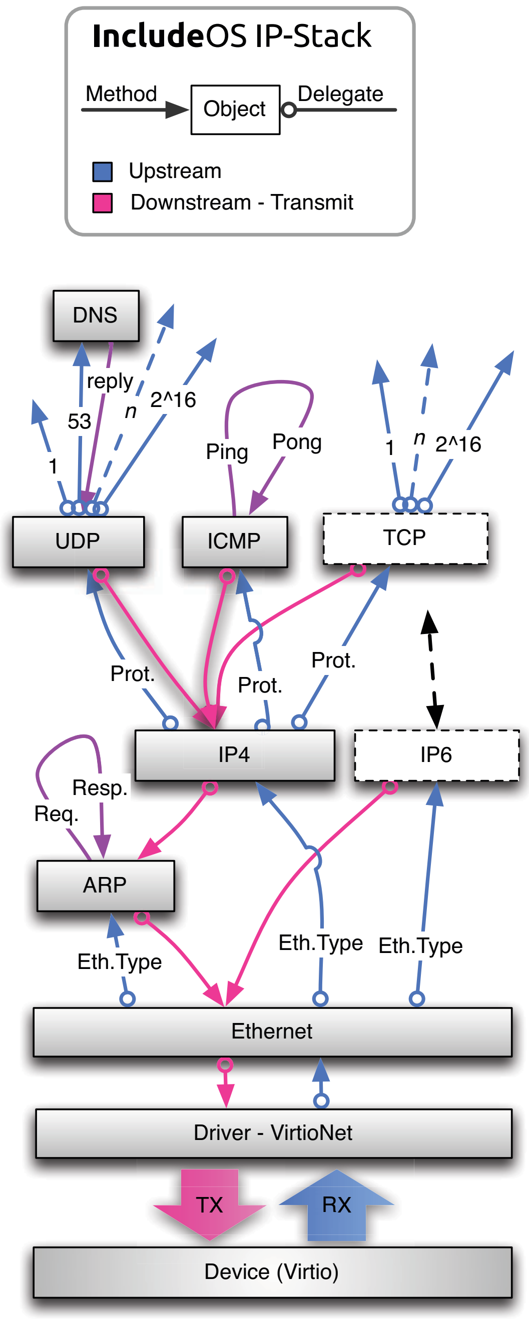

Fig. 1. *The design of the IncludeOS IP stack. Each box is an instantiation of a class, connected at runtime by fast delegates. The clean interface between modules makes makes it simple to replace parts, test modules independently or even completely rewire the IP stack during runtime. The classes have very few or no external dependencies, except for the standard library. Dashed lines represent modules yet to be completed.*

system for OpenStack guest systems, is around 300 MB.

The small memory footprint also has the effect of making IncludeOS exceptionally quick to boot - a minimal IncludeOS vm boots in about 0.3 seconds.

A. *Memory performance*

It is well known that hardware supported virtualization gives good overall performance. Nevertheless, the x86 architecture leaves much room for the operating system to influence this, by choice of CPU modes and memory protection schemes etc. While any general purpose operating system has to use hardware supported memory protection to ensure integrity among processes, a single-tasking operating system can chose to disable these features. The current IncludeOS prototype runs in 32-bit protected mode, but with virtual memory switched off. This eliminates the need for memory address translation inside the guest, which will reduce the complexity of memory access to a certain degree.

The STREAM benchmark [13] was used to give an rough idea of the memory performance of IncludeOS, vs. Ubuntu, and to test the hypothesis of whether disabling paging inside a virtual machine would have an immediate effect on memory performance. In order to ensure fair timing on IncludeOS and Linux, the STREAM source code was equipped with a custom timekeeping function, based on the `rdtsc`-instruction, replacing `gettimeofday`.

To account for differences in virtualization technologies, the following experiments were performed on a Dell PowerEdge^{TM}R815, with 4 x 12-core AMD Opteron^{TM}6234 CPU's at 2.4GHz and 128 GB of memory[2], and a Fujitsu PRIMERGY^{TM}TX2540 M1, with one 6-core Intel® Xeon® E5-2420 v2 CPU's at 2.2GHz and 8 GB of memory[3].

We experienced a high degree of variability when running the memory bandwidth tests. As a means to assure the consistency of the results, a Welsh two sample t-test was carried out between the first and the second half of the samples. The data set was considered consistent only when

[2] 16 x 8GB DDR3 DIMM's at 1600 MHz

[3] 1 x 8GB DDR3 DIMM at 1600 MHz

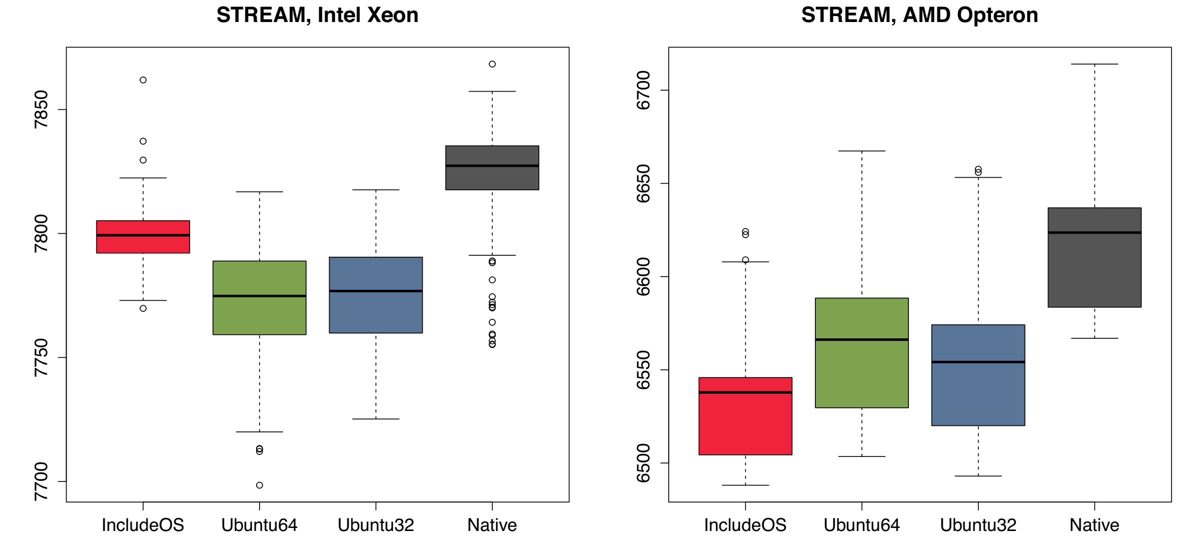

Fig. 3. *Memory bandwidth as reported by the STREAM "Triad" workload, for each of the test VM's, running on hypervisors with Intel Xeon and AMD opteron CPU's, respectively.*

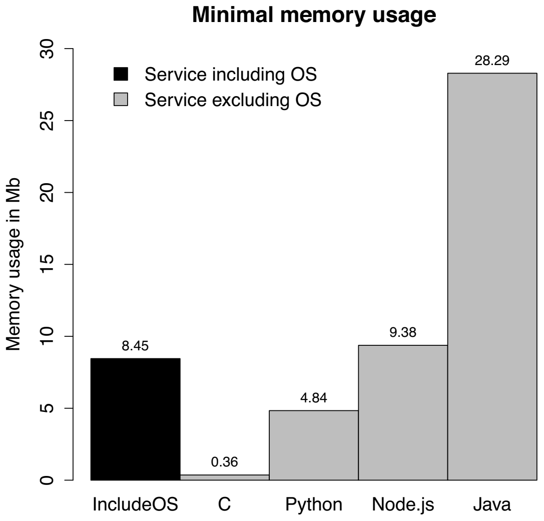

Fig. 2. *Showing the minimal memory usage of an IncludeOS virtual machine including a "Hello world" program, the necessary parts of the OS- and standard libraries, and a custom bootloader, in addition to the Qemu process itself. In comparison "Hello world"-programs in various language frameworks, excluding their operating systems, are shown.*

this t-test yielded a p-value of more than 0.1. In contrast, a t-test between the IncludeOS and Ubuntu64 data sets gave a p-value of 2.2×10^{-16} on both hypervisors.

Fig. 3 shows that IncludeOS has a small but consistent advantage over the Ubuntu VM's on the Intel Xeon server, but a small disadvantage of on the AMD server. However, on all servers, the difference between IncludeOS and the Ubuntu VM's were less than 0.5%.

V. Case: IncludeOS as a DNS server

In order to demonstrate the performance of IncludeOS running a real service, a simple DNS-service was written in such a way that the same binary could be run on both IncludeOS and Linux. The goal with this experiment is to demonstrate the resource overhead caused by the operating system, so it is not crucial to use feature-complete DNS-server, as long as both platforms run the same service. The service that was tested is essentially a partial implementation of the DNS protocol, allowing the service to answer real DNS-queries from tools such as `nslookup` and `dig`, but limited to A-records.

A. Experiment setup

Since IncludeOS does not have a program loader or a classical shell, but rather is a part of the program, the DNS-functionality was compiled into an object file that could be linked with IncludeOS and with a Linux executable respectively. On the Linux side a standard UDP socket was used, listening to port 53. All incoming data was then passed to the pre-compiled DNS-class for processing. The resulting reply was then passed back over the socket. On IncludeOS a lambda was registered as a callback for any packets entering UDP port 53. The lambda would simply pass the data buffer on to the pre-compiled DNS-service, and send the reply directly down the stack via the transmit-function in the UDP-class. The resulting disk image is only 158 Kb, including the kernel with interrupt handlers and event-loop, virtio-driver, network stack and bootloader.

The DNS-service was made to populate its registry with 10000 synthetic A-records, each corresponding to linearly increasing IP-addresses.

A program was written in C++ to run the experiment in a controlled fashion. The program would do the following:

1) Boot the DNS server as a subprocess, registering the PID.
2) When completely booted, record the contents of `/proc/<PID>/stat` at the hypervisor
3) Run `nslookup` 1000 times, each with a different query
4) Record the contents of `/proc/<PID>/stat` again, and calculate the difference in CPU-time.
5) Repeat the above for IncludeOS and Linux every other time, 100 times each.

The experiment was run on the same hardware as mentioned above, a 48-core AMD Opteron server, and a newer 6-core Intel Xeon server.

B. Experiment results

As shown in fig.4 the DNS service spent significantly less CPU time while running on IncludeOS, compared to when running on Ubuntu. On AMD IncludeOS uses 20% fewer CPU-ticks on average total, and 70% fewer ticks on average spent inside the guest CPU. On Intel IncludeOS uses 5.5% fewer CPU-ticks on average total, and 66% fewer ticks inside the guest.

In the figure, ”guest time” is the time spent inside the virtual CPU, and ”host time” is the time spent running the Qemu process, emulating the virtual hardware. The main factor affecting time spent in the host process, as opposed to inside the guest, is the number of *vm exits*, i.e. execution of protected instructions, which has to be forwarded to the hypervisor or emulated. For instance the instructions `out`, used for most bus communication, and `hlt` used to idle, are protected and will cause VM exits.

C. Limitations in the experiment

The network stack in IncludeOS is far from complete, and may not fully conform to RFC standards. It has been implemented using a pragmatic approach, providing only what needs to be in place for the DNS service to function in practice. For this reason, there might be certain networking features that are required by the RFC's, which if implemented would cause slowdown to IncludeOS. To the authors knowlege, no such features would significantly change the results. Additionally, IncludeOS does not yet implement the standard socket API, leaving more work for the application programmer, but also one less abstraction. At most, using sockets would incur a few extra cycles of processing time inside the guest CPU.

VI. Discussion

The main benefits of IncludeOS, compared to traditional Linux VM's are

- Extremely small disk- and memory footprint
- Overall performance increase, due to simplistic design
- No host- or software dependencies, other than virtual x86 hardware, and standard virtio for networking.
- Very fast boot time - less than 0.3 seconds.
- No system call overhead; the OS and the service are the same binary, and system calls are simple function calls, without passing any memory protection barriers
- No unnecessary overhead from timer interrupts
- No I/O-waiting, but rather an event-based asynchronous I/O-model
- No overhead of emulating the Programmable Interrupt Timer (i.e. no periodic timer interrupts and no preemptive scheduling)
- Reduced number of VM exits by keeping the number of protected instructions very low.

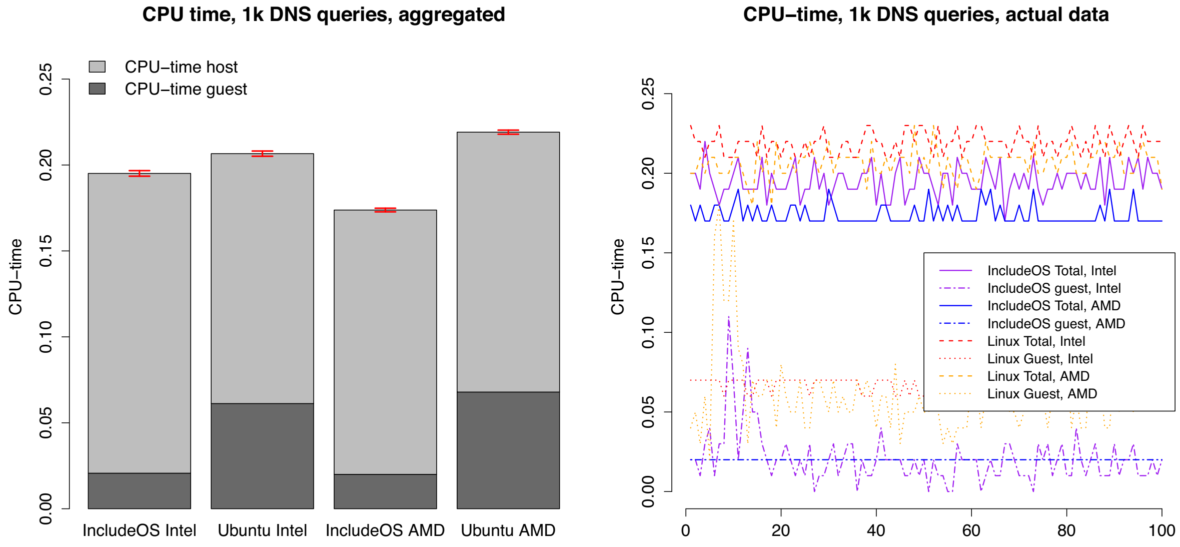

Fig. 4. *CPU-time required to execute 1000 DNS-requests, on IncludeOS and Ubuntu, running on Intel- and AMD systems. IncludeOS and Ubuntu are running the same DNS sevice binary. On AMD, IncludeOS uses 20% fewer CPU-ticks on average total, and 70% fewer ticks on average spent inside the guest CPU. On Intel IncludeOS uses 5.5% fewer CPU-ticks on average total, and 66% fewer ticks inside the guest. Error bars represent the 95% confidence interval over 100 samples.*

- A simple and highly modular zero-copy IP stack

A. *Deferring all interrupts*

That being so, IncludeOS in its current form is not fit for every task. In particular, deferring all IRQ's will cause the VM to seem unresponsive (i.e. not answer `ping`) under workloads requiring a lot of CPU activity pr. request (this is not the case for DNS) For many services, a non-preemptive kernel can work well, but services with realtime requirements are likely to require time-based interrupts. For DNS, and any other service with only a small CPU-load pr. request, a simple `ping` to the server would reply promptly, even at high loads, without any preemption. For services requiring several seconds of CPU-processing for each request, ICMP-packets would simply be queued until the virtio-queue was full, at which point they would be dropped, giving the impression of an unresponsive service.

B. *Performance across platforms*

An initial hypothesis motivating the memory performance experiment was that disabling paging inside a cloud guest would give a performance benefit. While we are confident in the data we presented, we have also experimented with other memory benchmarks, giving up to 8% difference between IncludeOS and Ubuntu, some times in favor of IncludeOS, some times in favor of Ubuntu. A large number of factors affect memory performance, such as CPU cache, page size, data alignment, the use of SSE etc. and especially on the 48-core server with many numa nodes, we found great variability in results. Further research is required to fully account for the effect of virtual memory inside guests. While it is possible that this has played a role in our DNS experiment, our conclusion so far is that the main reason for the increased CPU-performance of IncludeOS is due to the overall simplicity of the system.

C. *CPU-time outside the guest*

It might seem puzzling that an IncludeOS VM spends more time than Ubuntu, ”outside the guest CPU”. This means that the Qemu process has to do a bit more work to run an IncludeOS VM, as opposed to a Linux VM. IncludeOS however, more than makes up for this, by spending 66-70% less time inside the guest CPU - which is most likely due to the fact that IncludeOS simply does less work; there are no timer interrupts interrupting, and no scheduler that has to consider other processes. It is harder to explain why IncludeOS spends more time outside the guest, but one possibility is that the virtio driver is more optimized in Linux, causing fewer vm exits and interacting more efficiently with the host driver.

D. Concluding remarks

While many other projects are related, IncludeOS is different: where systems such as Mirage and OSv aims to provide a platform for a high-level language-runtimes, which impose significant resource penalties in themselves, IncludeOS aims to represent absolute minimality. As shown in Fig.2, ”Hello World” in Java, *without an operating system* require more than 3 times the memory of IncludeOS *with operating system*. Where systems such as Docker and Rumkernels depend heavily on the hypervisor kernel, IncludeOS will run on any virtualization platform, depending only on x86 virtualization.

While IncludeOS is still in its infancy we believe the results so far to be very promising. The small memory footprint will in itself allow more VM's to share a single server, and have significant impact on the cost of migration, which in turn can make much studied server consolidation schemes more feasible in practice. As an added benefit, the simplistic design of IncludeOS can also give an overall reduction in CPU usage.

E. Future work

A near-future use case for IncludeOS will be running high-performance web-applications, written in an asynchronous programming style similar to Node.js, but in a maximally efficient and minimal-overhead C++ language framework. These services will have no host kernel dependencies, running directly on top of virtual hardware, in any IaaS cloud.

Further goals include full IPv6 capabilities, file systems, 64-bit and multi-core support. All these will however be optional features; with IncludeOS you shouldn't pay for what you don't use.

IncludeOS is open source software and available from www.includeos.org.

REFERENCES

[1] (2010) Make it green: Cloud computing and its contribution to climate change. Green Peace. [Online]. Available: http://www.greenpeace.org/international/en/publications/reports/make-it-green-cloud-computing/

[2] Y. W. Gary Cook, David Pomerantz and T. Dowdall, “Clicking green: How companies are creating the green internet,” Tech. Rep., 2014. [Online]. Available: http://www.greenpeace.org/international/en/publications/reports/make-it-green-cloud-computing/

[3] A. Bratterud and H. Haugerud, “Maximizing hypervisor scalability using minimal virtual machines,” in *IEEE CloudCom 2013*, 2013, pp. 218–223.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., “Exokernel: An operating system architecture for application-level resource management,” in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266. [Online]. Available: http://doi.acm.org/10.1145/224056.224076

[5] A. Madhavapeddy and D. J. Scott, “Unikernels: Rise of the virtual library operating system,” *Queue*, vol. 11, no. 11, pp. 30:30–30:44, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2557963.2566628

[6] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, “Osv—optimizing the operating system for virtual machines,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[7] A. Kantee *et al.*, *Flexible operating system internals: The design and implementation of the anykernel and rump kernels*. Aalto University, 2012. [Online]. Available: http://urn.fi/URN:ISBN:978-952-60-4917-5

[8] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[10] B. Stroustrup, *The C++ Programming Language (4th. edition)*. Addison-Wesley, 2013.

[11] P. Pedriana, “EASTL - Electronic Arts Standard Template library,” Open Standards, Tech. Rep., Apr. 2007. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html

[12] R. Russell, M. S. Tsirkin, C. Huck, and P. Moll, “Virtual I/O Device (VIRTIO) Version 1.0,” OASIS Standard, OASIS Committee Specification 02, January 2015. [Online]. Available: http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html

[13] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.