Building Extremely Fast, Specialized Unikernels The Easy Way

Alexander Jung <a.jung@lancs.ac.uk>
Felipe Huici <felipe.huici@neclab.eu>
Sharan Santhanam <sharan.santhanam@neclab.eu>
Simon Kuenzer <simon.kuenzer@neclab.eu>

FOSDEM’21

This work has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreements no. 871793 (“ACCORDION”) and 825377 (“UNICORE”). This work reflects only the author’s views and the European Commission is not responsible for any use that may be made of the information it contains.
Specialization = High Performance
Specialization = High Performance

Hardware

- TPUs
- Movidius
- FPGAs

Costly...

inherently scoped...
Networking


Specialization = High Performance

Language-specific runtime environments

- **MirageOS** \w Madhavapeddy, Anil, and David J. Scott. "Unikernels: Rise of the virtual library operating system." Queue 11.11 (2013): 30-44.

- **Erlang on Xen (LING)**  [http://erlangonxen.org](http://erlangonxen.org)

- **runtime.js**  [http://runtimejs.org/](http://runtimejs.org/)
Specialization in Virtualization = Unikernels

1. Small image size
2. Fast boot times
3. Low memory consumption
4. High throughput
5. Potentially more secure
Achieving Unikernel Performance

1. **Transparency**: applications are ported and automatically benefit from lower boot times, less memory consumption, etc.

2. **Modified**: applications are hooked into high performance APIs at the right level in the software stack
Doing it with Linux?
Doing it with existing unikernels?

1. They **require significant expert work to build** and to extract high performance; such work has to for the most part be redone for each target application.

2. They are **often non-POSIX compliant**, requiring porting of applications and language environments.

3. The (uni)kernels themselves, while smaller, are **still** monolithic and hard to customize.
1. The kernel should be fully modular in order to allow for the unikernel to be fully and easily customizable.

2. The kernel should provide a number of performance-minded, well-defined APIs that can be easily selected and composed in order to meet an application’s performance needs.
But is it possible to provide transparent application support?
### How does Binary Compatibility compare?

<table>
<thead>
<tr>
<th>Platform</th>
<th>Routine call</th>
<th># Cycles</th>
<th>nsecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux/KVM</td>
<td>System call</td>
<td>604.62</td>
<td>232.55</td>
</tr>
<tr>
<td></td>
<td>System call (no mitigations)</td>
<td>142.31</td>
<td>54.74</td>
</tr>
<tr>
<td>Unikraft/KVM</td>
<td>System call</td>
<td>85.0</td>
<td>32.69</td>
</tr>
<tr>
<td>Both</td>
<td>Function call</td>
<td>6.0</td>
<td>2.31</td>
</tr>
</tbody>
</table>
Transparently Building from Source?

App native build system → .obj and .a files → link (Unikraft Build system) → musl (POSIX), syscall shim → Unikraft stack
## Compile time

<table>
<thead>
<tr>
<th></th>
<th>musl</th>
<th></th>
<th>newlib</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size (MB)</td>
<td>std</td>
<td>compat. layer</td>
<td>Size (MB)</td>
</tr>
<tr>
<td>lib-axtls</td>
<td>0.336</td>
<td>✔</td>
<td>✔</td>
<td>0.432</td>
</tr>
<tr>
<td>lib-bzip2</td>
<td>0.296</td>
<td>✔</td>
<td>✔</td>
<td>0.364</td>
</tr>
<tr>
<td>lib-c-ares</td>
<td>0.304</td>
<td>✔</td>
<td>✔</td>
<td>0.432</td>
</tr>
<tr>
<td>lib-duktape</td>
<td>0.700</td>
<td>✔</td>
<td>✔</td>
<td>0.772</td>
</tr>
<tr>
<td>lib-farmhash</td>
<td>0.232</td>
<td>✔</td>
<td>✔</td>
<td>0.276</td>
</tr>
<tr>
<td>lib-fft2d</td>
<td>0.356</td>
<td>✔</td>
<td>✔</td>
<td>0.396</td>
</tr>
<tr>
<td>lib-helloworld</td>
<td>0.232</td>
<td>✔</td>
<td>✔</td>
<td>0.256</td>
</tr>
<tr>
<td>lib-libucontext</td>
<td>0.232</td>
<td>✔</td>
<td>✔</td>
<td>0.276</td>
</tr>
<tr>
<td>lib-libunwind</td>
<td>0.232</td>
<td>✔</td>
<td>✔</td>
<td>0.276</td>
</tr>
<tr>
<td>lib-lighttpd</td>
<td>0.796</td>
<td>✗</td>
<td>✔</td>
<td>0.916</td>
</tr>
<tr>
<td>lib-lighttprepy</td>
<td>0.256</td>
<td>✔</td>
<td>✔</td>
<td>0.296</td>
</tr>
<tr>
<td>lib-memcached</td>
<td>0.524</td>
<td>✔</td>
<td>✔</td>
<td>0.672</td>
</tr>
<tr>
<td>lib-micropython</td>
<td>0.527</td>
<td>✔</td>
<td>✔</td>
<td>0.628</td>
</tr>
<tr>
<td>lib-nginx</td>
<td>1.13</td>
<td>✗</td>
<td>✔</td>
<td>1.20</td>
</tr>
<tr>
<td>lib-open62541</td>
<td>0.248</td>
<td>✗</td>
<td>✔</td>
<td>0.804</td>
</tr>
<tr>
<td>lib-openssl</td>
<td>2.98</td>
<td>✗</td>
<td>✔</td>
<td>3.01</td>
</tr>
<tr>
<td>lib-pcre</td>
<td>0.344</td>
<td>✔</td>
<td>✔</td>
<td>0.380</td>
</tr>
<tr>
<td>lib-python</td>
<td>4.75</td>
<td>✗</td>
<td>✔</td>
<td>4.81</td>
</tr>
<tr>
<td>lib-redis-client</td>
<td>0.640</td>
<td>✗</td>
<td>✔</td>
<td>0.801</td>
</tr>
<tr>
<td>lib-redis-server</td>
<td>1.26</td>
<td>✗</td>
<td>✔</td>
<td>1.42</td>
</tr>
<tr>
<td>lib-ruby</td>
<td>6.84</td>
<td>✗</td>
<td>✔</td>
<td>6.93</td>
</tr>
<tr>
<td>lib-sqlite</td>
<td>1.22</td>
<td>✔</td>
<td>✔</td>
<td>1.31</td>
</tr>
<tr>
<td>lib-zlib</td>
<td>0.348</td>
<td>✔</td>
<td>✔</td>
<td>0.404</td>
</tr>
<tr>
<td>lib-zydis</td>
<td>0.276</td>
<td>✔</td>
<td>✔</td>
<td>0.328</td>
</tr>
</tbody>
</table>
How much syscall support is enough?

A study of modern Linux API usage and compatibility: what to support when you're supporting.
Tsai et. Al, Eurosys 2016
What Unikraft *Could Transparently* Support

Syscalls required by a set of 30 popular server apps vs. Syscalls currently supported by Unikraft.
If all else fails – Manual Porting

![Bar chart showing the total porting time in working days for different categories across quarters: Libraries, Library dependencies, OS primitives, Build system primitives.]

- **Q2 2019**: Libraries 132, Library dependencies 60, OS primitives 31, Build system primitives 16
- **Q3 2019**: Libraries 88, Library dependencies 22, OS primitives 21, Build system primitives 18
- **Q4 2019**: Libraries 43, Library dependencies 1, OS primitives 46, Build system primitives 0
- **Q1 2020**: Libraries 24, Library dependencies 0, OS primitives 4, Build system primitives 0

© NEC Corporation 2021
What Unikraft Supports

On-going: R, OpenJDK, JS
Base Performance Evaluation
Unikernel image size compared to other projects
Boot time of Unikraft with different VMMs

QEMU 1NIC: 64.7ms
QEMU MicroVM: 84.7ms
Solo5: 18.6ms
Firecracker: 3.4ms
SQLite: 1.2ms
NGINX: 1.4ms
Minimum memory needed to run an application

![Bar chart showing minimum memory requirements for different applications and environments.](chart.png)
NGINX performance with wrk

<table>
<thead>
<tr>
<th>System</th>
<th>Average Throughput (x1000 req/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rumpun</td>
<td>18.4</td>
</tr>
<tr>
<td>OSv</td>
<td>36.3</td>
</tr>
<tr>
<td>Linux KVM</td>
<td>58.9</td>
</tr>
<tr>
<td>Docker</td>
<td>73.1</td>
</tr>
<tr>
<td>Unikraft</td>
<td>90.0</td>
</tr>
</tbody>
</table>
Redis performance tested with redis-benchmark

![Graph showing Redis performance with various setups.](image-url)
Specialization Performance Evaluation
Unikraft image sizes

- Default configuration
- + Dead Code Elim. (DCE)
- + Link-Time Optim. (LTO)
- + DCE + LTO

<table>
<thead>
<tr>
<th>Application</th>
<th>Default</th>
<th>DCE</th>
<th>LTO</th>
<th>DCE + LTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>helloworld</td>
<td>256.7KB</td>
<td>256.7KB</td>
<td>192.7KB</td>
<td>192.7KB</td>
</tr>
<tr>
<td>nginx</td>
<td>832.8KB</td>
<td>328.8KB</td>
<td>1.1MB</td>
<td>1.1MB</td>
</tr>
<tr>
<td>redis</td>
<td>832.8KB</td>
<td>328.8KB</td>
<td>1.1MB</td>
<td>1.1MB</td>
</tr>
<tr>
<td>sqlite</td>
<td>832.8KB</td>
<td>328.8KB</td>
<td>1.1MB</td>
<td>1.1MB</td>
</tr>
</tbody>
</table>
Unikraft NGINX throughput \w diff mem allocators

Average Throughput (x1000 req/s)

- mimalloc: 90.0
- tinyalloc: 37.7
- TLSF: 88.8
- Binary Buddy: 88.4
Unikraft NGINX boot time \w diff mem allocators

- alloc
- plat
- ukbus
- lwip
- pthreads
- vfscore
- misc
- rootfs
- virtio

Total Boot Time (ms)

Binary Buddy mimalloc bootalloc tinyalloc TLSF
Execution speedup of SQLite relative to mimalloc
Throughput with Redis using redis-benchmark

![Graph showing throughput with Redis using redis-benchmark](image_url)

- **GET**: mimalloc (1184), tlsf (959), bbuddy (849), ta (462)
- **SET**: mimalloc (597), tlsf (862), bbuddy (295), ta (173)
- **LPUSH**: mimalloc (338), tlsf (298), bbuddy (295), ta (173)
- **MSET**: mimalloc (218), tlsf (185), bbuddy (182), ta (69)

**Mean throughput (Kreq/s)**

- Redis request type: GET, SET, LPUSH, MSET
Filesystem Specialization for web caching

![Graph showing average TSC for different filesystems and operating systems.](chart.png)
## Key-value store application specialization

<table>
<thead>
<tr>
<th></th>
<th>Linux Baremetal</th>
<th>Linux KVM Guest</th>
<th>Unikraft KVM Guest</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single</strong></td>
<td>509K/s</td>
<td>985K/s</td>
<td>250K/s</td>
</tr>
<tr>
<td><strong>Batch</strong></td>
<td>105K/s</td>
<td>276K/s</td>
<td>6M/s</td>
</tr>
<tr>
<td><strong>DPDK</strong></td>
<td>6.1M/s</td>
<td>276K/s</td>
<td>6.1M/s</td>
</tr>
<tr>
<td><strong>LwIP</strong></td>
<td>250K/s</td>
<td>276K/s</td>
<td>6M/s</td>
</tr>
<tr>
<td><strong>Netdev</strong></td>
<td>6M/s</td>
<td>6.1M/s</td>
<td></td>
</tr>
<tr>
<td><strong>DPDK</strong></td>
<td>6.1M/s</td>
<td>6.1M/s</td>
<td></td>
</tr>
</tbody>
</table>
TX throughput of Unikraft vs. Linux KVM VM

Throughput (Mpps)

Packet Size (Bytes)

- Unikraft with vhost-user
- Unikraft with vhost-net
- Linux DPDK with vhost-user
- Linux DPDK with vhost-net
Future directions on Specialization

- **Compartmentalization**
  1. write critical micro-libs in memory safe, race condition safe or statically verifiable languages
  2. compile and link them together
  3. use HW assisted memory separation (CHERI, Intel MPKs, etc.) to retain languages’ properties

- **Code reduction**

- **Sealing** (hypervisor call to set pages as read-only or execute-only after boot)

- **Upstream standard features** (ASLR, stack protection etc.)

- **Fuzzing** (for verification of above)
Find us online

https://github.com/unikraft

http://unikraft.org

<minios-devel@lists.xenproject.org>
<unikraft@listserv.neclab.eu>

@UnikraftSDK
Thanks