H2020-ICT-2018-2-825377

UNICORE

UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments - Intermediate

Horizon 2020 - Research and Innovation Framework Programme

D4.2 Design & Implementation of Tools for Unikernel Deployment - Intermediate

Due date of deliverable: June 30, 2020

Actual submission date: June 30, 2020

Start date of project	January 1, 2019
Duration	36 months
Lead contractor for this deliverable	University POLITEHNICA of Bucharest (UPB)
Version	1.0
Confidentiality status	”Public”

Abstract

The goal of the EU-funded UNICORE project is to develop a common code-base and toolchain that will enable software developers to rapidly create secure, portable, scalable, high-performance solutions starting from existing applications. The key to this is to compile an application into very light-weight virtual machines – known as unikernels – where there is no traditional operating system, only the specific bits of operating system functionality that the application needs. The resulting unikernels can then be deployed and run on standard high-volume servers or cloud computing infrastructure. The objective of this deliverable is to describe the current architecture of the toolchain as well as the different tools that have been developed to build and orchestrate unikernels. The current toolchain contains two fully-functional tools (dependency analysis tool and automatic build tool) that have been tested with different schemes and configurations. Others tools are still in research and will be released in a next milestone.
This deliverable contains updated report detailing the progress since the release of D4.1. It includes design of tools and host environment. We include measurements of current unikernel builds augmented by the use of current tools and discuss the development and deployment environment and support for extended features. A second release of the source code is also part of the deliverable.

Target Audience

The target audience for this document is **public**.

Impressum

Full project title	UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments
Title of the workpackage	WP4 - Toolstack Implementation
Editor	University POLITEHNICA of Bucharest (UPB)
Project Co-ordinator	Emil Slusanschi, UPB
Technical Manager	Mike Rapoport, IBM
Copyright notice	© 2020 Participants in project UNICORE

Executive Summary

The UNICORE D4.2 deliverable, ”Design & Implementation of Tools for Unikernel Deployment - Intermediate”, describes the current architecture of the UNICORE toolchain as well as the different tools that have been developed to build and orchestrate unikernels. The approach taken is to describe the general structure of the UNICORE toolchain and then explain each tool in a separate way, including support for extended features and a description of the development and deployment environment.

Dependency analysis and automatic build tools shorten the time taken to add new components to the UNICORE software set with binary compatibility filling the spot of proprietary or difficult to configure and build applications. UNICORE builds run on multiple platforms and multiple architecture, with a modular platform API paving the way to extend that. Verification and validation support will strengthen the security of unikernel applications, more than the inherent reduction in attack surface. The integration of modern orchestrators will increase the use cases and availability of UNICORE unikernel builds.

Development is done following an open source model, with code releases being available.

The document presents performance measurements highlighting the benefits of highly specialized UNICORE images, the ever-reducing time in porting new applications and the increasing automation of parts of the development, configuration, build and deployment process.

List of Authors

Authors	Răzvan Deaconescu and Costin Raiciu (UPB), Felipe Huici and Simon Kuenzer (NEC), Gaulthier Gain and Cyril Soldani (ULiège), Xavier Peralta (CSUC), Mike Rappoport (IBM)
Participants	UPB, NEC, ULiège, CSUC, IBM
Work-package	WP4 - Toolstack Implementation
Security	PUBLIC
Nature	R
Version	1.0
Total number of pages	38

Contents

List of Figures

List of Tables

1 Introduction

UNICORE relies on its core and external libraries, build system and toolchain to create specialized (fast and secure) unikernel images. Images selectively incorporate core APIs, libraries and application code into a singular binary image that is loaded and run on an underlying platform. UNICORE supports multiple virtualization platforms running on several CPU architectures.

UNICORE toolchain assists in the development, porting, building, configuration, securing and deployment of UNICORE applications. This document highlights the current design of UNICORE tools, use cases and evaluation. It's a continuation of the D4.1 deliverable.

The development efforts since D4.1 have sprung a diverse set of tools that will be presented throughout the paper. Each tool is valuable in automating, easing, hardening and evaluating the development and deployment of UNICORE unikernel solutions.

Chapter 2 details the tools in the toolchain, their use cases and benefits. Chapter 3 focuses on the multi-target and verification and validation support for UNICORE, as WP4 tasks. Chapter 4 presents the development and deployment environment, showing the two approaches of building already ported source code or running existing binary applications via a new binary compatibility layer. Chapter 5 gives a first view of integrating UNICORE into modern orchestration solutions. Chapter 6 presents code release information. Chapter 7 concludes.

2 Tools

UNICORE toolstack is used for automating the building, verification, configuration and deployment of unikernel images. The toolstack relies on the inherent modularity of UNICORE components (core APIs, external libraries) to select required software components and build a final specialized unikernel instance.
This chapter presents the state of the following tools:

- **decomposition tool**: used to break larger software components into smaller building blocks
- **automatic build tool**: used to build software components (semi) automatically; it consists of two separate tools: one for (semi)automatic porting of new apps and one for automated building of existing software libraries
- **verification tool**: to validate existing software components
- **performance optimization tool**: to analyze and configure software components for optimal performance

2.1 Decomposition Tool

The decomposition tool will be used to break down monolithic libraries such as libc and operating system primitives (e.g., memory allocators, network stack, ...) into a set of small modules that can be selected from a libraries pool to build unikernel(s). The tool will help developers in decomposing the software, and is targeted at the UNICORE consortium and not the software community at large.
Another objective was to identify dependencies between functions and libraries, as well as dependencies between different libraries that are known by package managers such as dpkg, yum or apt. Then, these relations are displayed in graphical form to help the human expert understand the interactions between different components. For this part, it was chosen to merge this identification procedure within the dependency analysis tool.
The decomposition tool is still in research phase therefore only some assumptions are established. Two areas of research were considered and needs to be further analysed.

2.1.1 Decomposing Software into Subsystems

Architecture of large and complex systems is structured into a series of packages. One goal of an architectural decomposition is to provide a way to better understand the source. Considering the Linux kernel, this one implements a number of important architectural attributes. At a high level, and at lower levels, the kernel is layered into a number of distinct subsystems. This model proposes a regulatory system that classifies services based on their common characteristics. Major components of the Linux kernel are illustrated in Figure 2.1.
The decomposition process consists of three main steps:

(i) Treat all source files from specified directories built into the Linux kernel as subsystems and perform incremental decomposition isolating one kernel sub-component at a time;

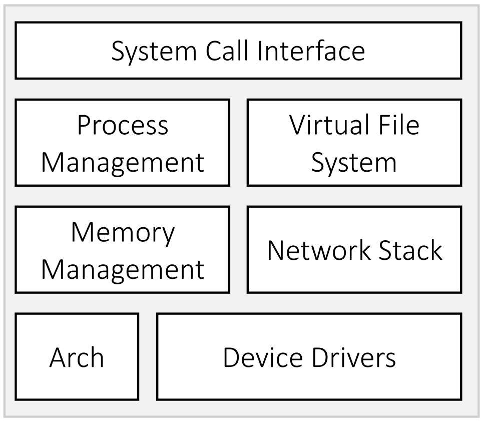

Figure 2.1: Architectural perspective of the Linux kernel

(ii) When kernel subsystems have been isolated, patterns recognition techniques can be used to extract relevant files and blocks of code. The idea is to provide as input, all the symbols that are used by a particular application and that are unknown by the UNICORE build system.

(iii) These components are then integrated with each other where unknown functions and symbols are replaced by stubs.

The first objective of the tool is thus to help experts to understand the interactions between different components and to obtain a first skeleton of a micro-library. After this semi-automatic extraction, developers will have to work on their own by implementing and verifying all stub functions to have fully functional modules.

2.1.2 Using Existing Tools

Another approach is to use open-source tools like Clang Static Analyser[1] and to integrate them to the toolchain. Such tool(s) would allow to perform an analysis of the application flow code. Once the application flow code has been identified, a graph can be generated in order to help developer(s) to extract the identified sub-systems.

2.2 Dependency Analysis Tool

The objective of the dependency analysis tool is to gather, for the target applications, which software in the operating system they actually use. Such software will include shared libraries, other applications, core kernel components, kernel modules, and so forth. The tool needs to find a sufficient, but minimal super-set of other software that must be installed for the application to work correctly.

A first functional dependency analysis tool has been developed. This one aims to break down a binary application and analyses its symbols, its system calls and its dependencies. The tool is a console-based application and relied on static and dynamic analysis. The tool has been designed to examine binary files. Indeed, source code is not always accessible. Moreover, this way of doing is independent of the programming language and is thus easier to use to gather binary information. Finally, gathering data of binary is also useful for binary rewriting techniques [2]. Nevertheless, code parsing tools can also be considered in future release(s) even if they require more time and resources to develop.

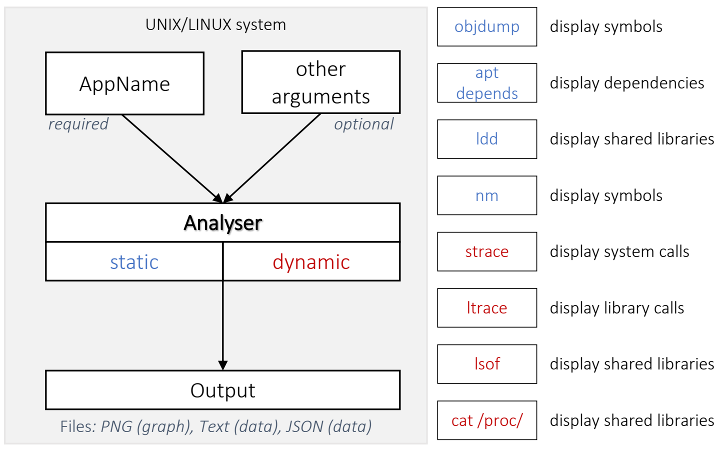

Figure 2.2: High-level overview of the UNICORE dependency analysis tool

Figure 2.2 represents a high-level overview of the tool. Firstly, the dependency analysis tool requires the application name as an input to gather binary information. This one can either be absolute or simply passed as an executable file name (e.g., SQLite). In that case, the tool will search in the path environment variable to locate the executable file and gets its absolute path (e.g., /usr/bin/sqlite).

A static analysis is performed on the binary file of the application. It allows to recover all the symbols which compose it. In order to perform such a task, the tool uses several internal commands such as `nm`, `objdump`, `apt-cache depends` and `ldd`. The output of each command is then parsed and various information is stored into adequate data structures. However, the result of this analysis can be limited since binaries can be stripped or obfuscated.

A second analysis is thus performed on the application. This one is dynamic since it requires running the program. It allows to collect various information such as system calls, library calls and shared libraries. To gather all the symbols and dependencies of a process, it is necessary to explore all possible execution paths of the application. Nevertheless, exploring all execution paths is in general infeasible. Fortunately we can approach this scenario, by using a heuristic approach which requires high application code coverage. Tests with expected input and fuzz-testing techniques have been used in order to find the most possible symbols and dependencies. As for the static analysis, several internal commands are executed: `strace`, `ltrace`, `lsof`, `cat` and `/proc/pid`. The difference here is that the program is executed. Binary information is also saved into several data structures. When both analysis are completed, a JSON file is automatically generated. In addition to the JSON file, the tool can automatically generate dependencies and shared libraries graphs. Such graphs allow to illustrate the relation between the dependencies. For example, considering again SQLite, Figure 2.3 represents all shared libraries (as well as their dependencies) acquired during the dynamic dependency analysis.

Required dependencies gathered during the static analysis procedure are shown in Figure 2.4.

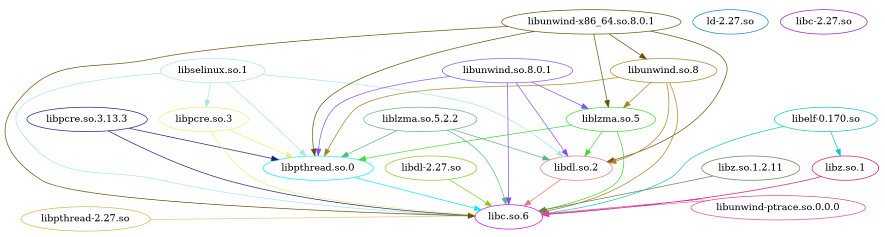

Figure 2.3: Shared libraries required by SQLite (dynamic analysis)

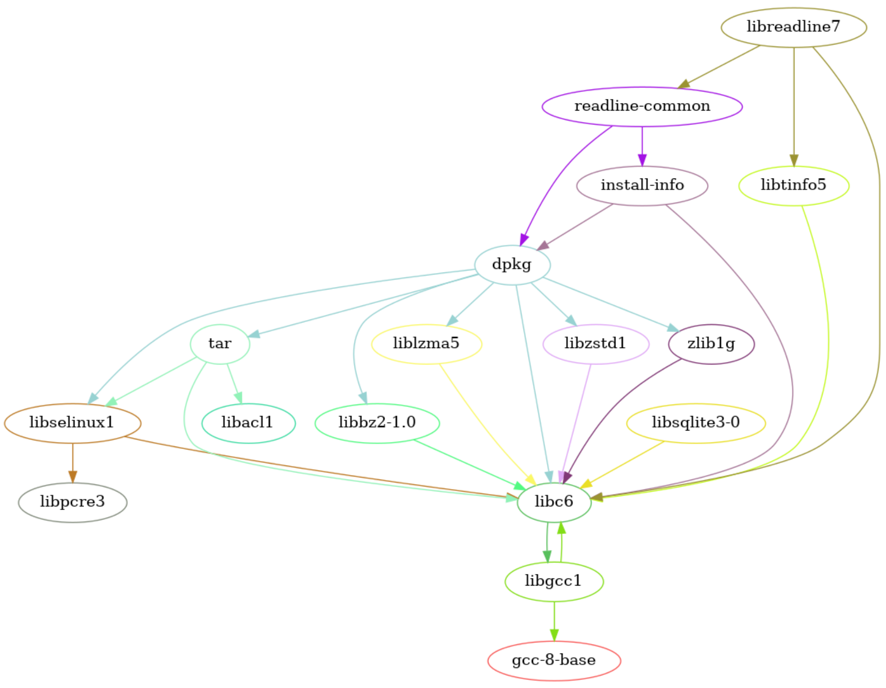

Figure 2.4: Dependencies required by SQLite (static analysis)

	#syscalls	#libcalls	#libs	LoC diff
Ruby	42	82	8	48.34%
Python3	143	3155	7	58.09%
Redis client	42	390	3	70.66%
Redis server	79	2411	11	81.06%
Nginx	91	756	18	83.66%
Lighttpd	74	544	8	94.50%
SQLite	59	141	7	100%
Memcached	82	173	3	100%

Table 2.1: Compilation results for the auto-porting tool

2.3 Tests and Fuzz-Tests

For the dynamic analysis, different tests were used to explore execution paths of a particular application. In order to do it, the dependency tool provides two different ways: either passing a test file as argument which contains internal commands to test the current application (e.g., SQL queries for a database, ...) or either to perform manually tests by specifying a duration to test.

For each application, it is thus necessary to manually write the tests or test the program. In addition, tests can use the default configuration of an application. As a result, it is possible that not all execution paths are tested.

2.4 Automatic Build Tool

With respect to automating the build, there are not two tools in existence, each with their own purpose. One tool is used to assist developers in porting a new application on the of the μlibs. It is a semi-automatic process that does most of the work the developer has to do to integrate the application build system with UNICORE; it is then left for the developer to make adjustments. The second tool (named `kraft`) is used to build, configure, deploy and assist in the development of applications and UNICORE μlibs.

2.4.1 Semiautomatic Porting

To speed up the porting of new application we develop a specialized tool. It inspects a software project's source code and attempts to automatically generate the Makefile and its related files.

The goal is to automatically port as much as possible and then leave the rest for the developer, i.e. this is a semi-automated process. In the ideal case everything would be done automatically and no other effort is required from the developer. In the optimal case, a large percentage of the porting is automatic and only a fraction is left in the hands of the developer. It relies on the dependency analysis tool in Section 2.2.

To test it, we run our tool against a number of applications (see Table 2.1). As shown, for two projects (memcached and SQLite) the tool was able to port them fully automatically. For the remaining ones, while it was not able to generate a working image, we quantify to what extent it succeeded by comparing the difference in lines of code between its auto-generated files and those we created by hand (which we know to work); we report these results in the last column of the table. The takeaway here is that while fully automated porting is quite difficult to achieve (e.g., due to having to handle different build systems and unmet dependencies), having *most* of the porting work done automatically significantly reduces the burden on the developer.

2.4.2 kraft: Automatic Build and Deploy

Once applications are ported and part of accessible repositories, there is a need to build them for deployment. This requires configuration of the build system, updating Kconfig files, pulling required dependencies. To automate this process, a specialized tool, **kraft** is available. It provides the `kraft` command to set up the environment for building, development and deployment of UNICORE unikernel images. This section given an outline of kraft and its automatic build use case, while the development and deployment parts are discussed in Chapter 4.

To begin using UNICORE unikernels you can use the command-line utility `kraft` which is a companion tool used for defining, configuring, building, and running unikernel applications. With `kraft`, you can create a build environment for your unikernel and manage dependencies for its build. Whilst `kraft` itself is an abstraction layer to the UNICORE build system, it proves as a useful mechanic and starting ground for developing unikernel applications.

Once `kraft` it installed you can begin by initializing a new unikernel repository using `kraft init`. As an example, you can build a Python 3 unikernel application by running the following:

```
kraft list
mkdir ~/my-first-unikernel && cd ~/my-first-unikernel
kraft up -a helloworld -m x86_64 -p kvm
```

If this is the first time you are running `kraft`, you will be prompted to run an update which will download UNICORE core and additional library pool sources. These sources are saved to directory set at the environmental variable `UK_WORKDIR` which defaults to `~/.unikraft`.

With a newly initialized unikernel application, the `./my-first-unikernel` directory will be populated with a `deps.json` file which contains references to the relevant library dependencies which are required to build a unikernel with support for Python 3. This file is used by `kraft` to configure and build against certain library versions. In addition to this file, a new `.config` file will also be placed into the directory. This file is used by UNICORE's build system to switch on or off features depending on your application's use case.

The unikernel application must now be configured against the UNICORE build system so that you and it can resolve any additional requirements:

```
kraft configure ./my-first-unikernel
```

This step can be made more interactive by launching into the unikernel's Kconfig configuration system. Launch an ncurses window in your terminal with `kraft configure --menuconfig`.

The configuration step used in `kraft` will perform necessary checks pertaining to compatibility and availability of source code and will populate your application directory with new files and folders, including:

- `kraft.yaml` – This file holds information about which version of the UNICORE APIs, additional libraries, which architectures and platforms to target and which network bridges and volumes to mount during runtime.
- `Makefile.uk` – A Kconfig target file you can use to create compile-time toggles for your application.
- `build/` – All build artifacts are placed in this directory including intermediate object files and unikernel images.
- `.config` – The selection of options for architecture, platform, libraries and your application (specified in `Makefile.uk`) to use with UNICORE.

When your unikernel has been configured to your needs, you can build the unikernel to all relevant architectures and platforms using:

```
kraft build ./my-first-unikernel
```

This step will begin the build process. All artifacts created during this step will be located under `./my-first-unikernel/build`.

2.5 Verification Tool

The verification tool is used for application equivalence and correctness. Features are to be added, with fuzzing currently being used for correctness. To use fuzzing for flaw discovery, the unikernel app image will be run in two modes: either as standalone as a Linux user space process with the fuzzer injecting input to its exposed functions or will run as a minimal fuzzer in case of a Xen / KVM build. Fuzzing will target both newly built apps using the μlibs and the μlibs themselves; in case of μlibs, the application will be minimal containing only required information for fuzzing.
As a fuzzing engine for creating and updating inputs to the program, Syzkaller is used. Syzkaller was chosen because of two main aspects:

(i) Syzkaller is one of the few fuzzers that uses as target an entire API and not just one function.

(ii) Syzkaller targets kernels. So it is designed for data structures that are usually found in kernels which makes it easier to adapt for UNICORE unikernels.

The biggest change in Syzkaller's architecture required for unikernels is the function calling: the way Syzkaller invokes the unikernel's equivalent system call. `dlsym()` and `dlopen()` were used to load the unikernel image and extract required unikernel API symbols.
Steps required to integrate UNICORE unikernel images in Syzkaller were:

(i) `Extracting the system call API for UNICORE images`
A list of all the system calls that UNICORE APIs possess has to be extracted first.

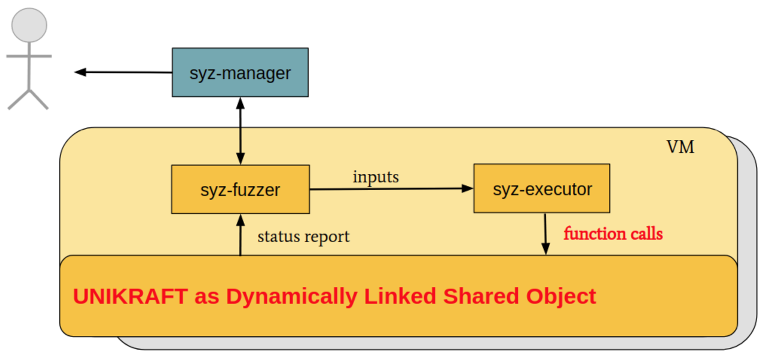

Figure 2.5: Syzkaller new architecture

(ii) `Defining the system calls using syzlang`

Define the system calls and data structures present in Syzkaller in the `.txt` files. Those files are present in `sys/$TARGET_OS` folder and will be used firstly to generate the constants used to call the system calls.

(iii) `Generate the constants`

Out of the previously defined files and the source files of the target operating system, constants are extracted.

(iv) `Generating the GO definition files`

Using the definitions in the `.txt` files, they are converted into GO files which Syzkaller can read.

(v) `Add UNICORE unikernels as a target`

For this step, multiple files need to be changed in order for Syzkaller to recognise the unikernel as a target.

(vi) `Change the calling system in Syzkaller`

Because we can not call the functions in the same way as for the Linux kernel, we have to change it to use `dlopen()` and `dlsym()` instead.

(vii) `Create a list of function targets`

Because we changed the way Syzkaller is making the function calls, an additional C++ source file needs to be added in order to make the calls with `dlsym()`. This step will be detailed in the next chapter.

In the end, the new architecture of the project is shown in Figure 2.5

The three main components in Syzkaller are kept as they are with changes to accommodate unikernel characteristics. The function calls are still invoked in a virtual machine, but the operating system in the virtual machine will not be the target OS, but Linux.

Syzkaller now runs with unikernel support. In the listing below is shown the output from running the fuzzer. It continuously starts new virtual machines and feeds input to it as shown in Listing 2.1.

```
2020/06/21 20:25:27 comparison tracing      : failed to get target: unknown target: linux/amd64 (
    supported: [unikraft/amd64])
2020/06/21 20:25:27 extra coverage          : failed to get target: unknown target: linux/amd64 (
    supported: [unikraft/amd64])
2020/06/21 20:25:27 setuid sandbox          : enabled
2020/06/21 20:25:27 namespace sandbox       : /proc/self/ns/user does not exist
2020/06/21 20:25:27 Android sandbox         : enabled
2020/06/21 20:25:27 fault injection         : CONFIG_FAULT_INJECTION is not enabled
2020/06/21 20:25:27 leak checking           : CONFIG_DEBUG_KMEMLEAK is not enabled
2020/06/21 20:25:27 net packet injection    : /dev/net/tun does not exist
2020/06/21 20:25:27 net device setup        : enabled
2020/06/21 20:25:27 concurrency sanitizer   : /sys/kernel/debug/kcsan does not exist
2020/06/21 20:25:27 devlink PCI setup       : PCI device 0000:00:10.0 is not available
2020/06/21 20:25:27 USB emulation           : /dev/raw-gadget does not exist
2020/06/21 20:25:36 VMs 1, executed 5, corpus cover 0, corpus signal 0, max signal 0, crashes 0,
    repro 0
2020/06/21 20:25:46 VMs 1, executed 245, corpus cover 0, corpus signal 4, max signal 9, crashes 0,
     repro 0
2020/06/21 20:25:56 VMs 1, executed 495, corpus cover 0, corpus signal 14, max signal 23, crashes
    0, repro 0
2020/06/21 20:26:06 VMs 1, executed 726, corpus cover 0, corpus signal 23, max signal 26, crashes
    0, repro 0
2020/06/21 20:26:16 VMs 1, executed 967, corpus cover 0, corpus signal 27, max signal 29, crashes
    0, repro 0
```

Listing 2.1: Syzkaller output for unikernel target

2.6 Performance Optimization Tools

The performance optimization tools are planned to use machine learning techniques to optimize a unikernel's build configuration options using data collected by the performance analysis. As preliminary steps support for PGO (Profile Guided Optimization) was added and preliminary performance measurements validating the benefits of specialized UNICORE unikernels were done.

2.6.1 Profile Guided Optimization

`Profile Guided Optimization (PGO)` [3] is an optimization solution through which the compiler could use the information collected while running the application. This mechanism obtains dynamic data about the application. The compiler will be able to create better scenarios that will approximate the application flow using the data received from this mechanism.

Initially, without PGO, for UNICORE builds the compiler will perform all the optimizations associated with the selected optimization flags based on static heuristics, not taking into account the behavior of the application. PGO support instructs the compiler to make optimizations using information while running the application. It also adds the benefit of getting coverage profiles of the libraries, opening the possibility of using them in fuzzing.

PGO compiler support works in three stages:

- instrumentation stage: in this stage, an executable is produced that contains annotations with the number of runs for each basic block in the program. For example, if the block is a branch, then the compiler will figure out which branch is more likely.
- training stage: the executable obtained in the previous stage is run within this stage. After running, a file is generated that contains data about the execution of the program.
- optimization stage: within this stage, the compiler uses the information obtained at the previous stage in order to make better estimates and optimizations.

To implement this mechanism, the `-fprofile-generate` flag is used for the instrumentation stage and the `-fprofile-use` flag is used for the optimization stage. These flags are used for both compilation and linking. In the linking step, the linker adds the `libgcov` library.

2.6.1.1 Helloworld Application Performance Analysis

For starters, we create a simple application for sorting a set of integers (15000 and 10000) using the `Bubble Sort` algorithm to test the correctness of the `PGO` mechanism and evaluate the percentage of performance it brings.
We updated the build environment for PGO. Then, we executed the instrumentation step to generate the profiling file associated with the sorting algorithm. After obtaining this file, we did the optimization stage in which we obtained an optimized executable based on the profiling file.
There is an improvement in the running time of the sorting application compiled with the `-O2` flags and those specific to `PGO`. Consequently, the mechanism brings a performance increase of `6,32%`.

2.6.1.2 Redis Application

We used the Redis application as a unikernel image to further evaluate performance benefits of PGO. We compared the `Redis` app compiled with the `-O2` flag vs the `Redis` app compiled with the `-O2` and `PGO` flags.
We considered three scenarios:

- tests containing only read operations (GET)
- tests containing only write operations (SET)
- tests containing 50% read operations and 50% write operations

The version of the application with the `-O2` flag for GET and the `PGO` mechanism is more efficient on average by `2.59%` compared to the version compiled only with the `-O2` flag.
The version of the application compiled with the `-O2` flag for SET and `PGO` flags is more efficient on average by `2.07%` compared to the version compiled only with the `-O2` flag.

For the 50% / 50% write version we get an improvement in the performance of the `Redis` application if it is compiled with the `-O2` and `PGO` flags. Consequently, the `PGO` mechanism increases the performance of the `Redis` application by `2.06%` for write operations and by `3.3%` for read operations if a scenario is applied with 50% write operations and 50% read operations.

2.6.1.3 SQLite Application

We also used the `SQLite` application as a UNICORE unikernel image. We compared the `SQLite` app compiled with the `-O2` flag vs the `SQLite` app compiled with the `-O2` and `PGO` flags.
We considered three scenarios:

- tests containing only insert operations
- tests containing only update operations
- tests containing 50% insert operations and 50% update operations

The `PGO` mechanism brings a significant improvement in the performance of the `SQLite` application for insert operations. Consequently, an increase of performance by `12.39%` can be observed in the case of running a scenario that contains only insert operations.
The version of the `SQLite` application compiled with the `-O2` and `PGO` flags is `2.29%` more efficient if a scenario containing only update operations is applied for only update operations.
There is an improvement in the performance of the `SQLite` application if it is compiled with the `-O2` and `PGO` flags. Hence, the `PGO` mechanism increases the performance of the `SQLite` application by `6.87%` if a scenario is applied with 50% insert operations and 50% update operations.

2.6.2 Specializing APIs for Performance

In this section we show an initial exploration of the specialization opportunities UNICORE provides, with the main goal of achieving better performance. Considering these results we plan to use machine learning techniques to optimize a unikernel's build configuration options using data collected by the performance analysis.
As an example, we analyze different memory allocators. No single memory allocator is perfect for all purposes [4], making them a great target for specialization. To support this, the ukalloc API allows for multiple allocators to be present in a single image, and enables different micro-libraries to use different allocators.
To test whether this approach could lead to performance gains, we used four memory allocators that comply with its API:

(i) the buddy memory allocator from MiniOS [5];

(ii) TLSF, a general purpose dynamic memory allocator specifically designed to meet real-time requirements;

(iii) mimalloc, a state-of-the-art, general-purpose allocator by Microsoft;

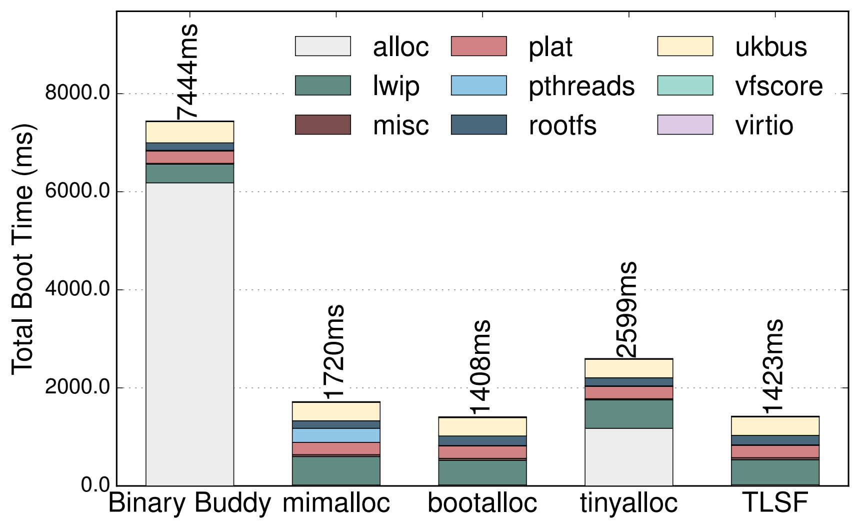

Figure 2.6: Boot time for nginx with different allocators

(iv) tinyalloc, a small and simple allocator; and bootalloc, our simple, region allocator to speed up boot times.

We them to examine performance of different workloads.

We built nginx with all the above allocators and measured the guest boot time (Figure 2.6), as well as the sustained performance of the server (Figure 2.7). The difference in boot times for the different allocators is quite large: from 1.5ms (tlsf) to 7.5ms (buddy), hinting that a just-in-time instantiation use-case of nginx should steer clear of the buddy allocator. At runtime, however, the buddy allocator performs similarly to tlsf and mimalloc, with tinyalloc at half of the performance.

Boot performance is similar for SQLite, with the buddy allocator being the worst and tinyalloc and tlsf among the best (results not shown for brevity). At runtime, though, the pecking order depends on how many queries are run (see Figure 2.8): tinyalloc is the fastest for less than 1000 queries by 5-26%, being a strong candidate for lambda-style deployments of SQLite. It becomes suboptimal with more requests, as its memory compaction algorithms are slower; using mimalloc, instead, provides a 20% performance boost when many queries are serviced.

Rhea name is the UNICORE build, the current version name.

Results for Redis, shown in Figure 2.9, further confirm that no allocator is optimal for all workloads, and that the right choice of allocator for the workload and use-case can boost performance by 2x.

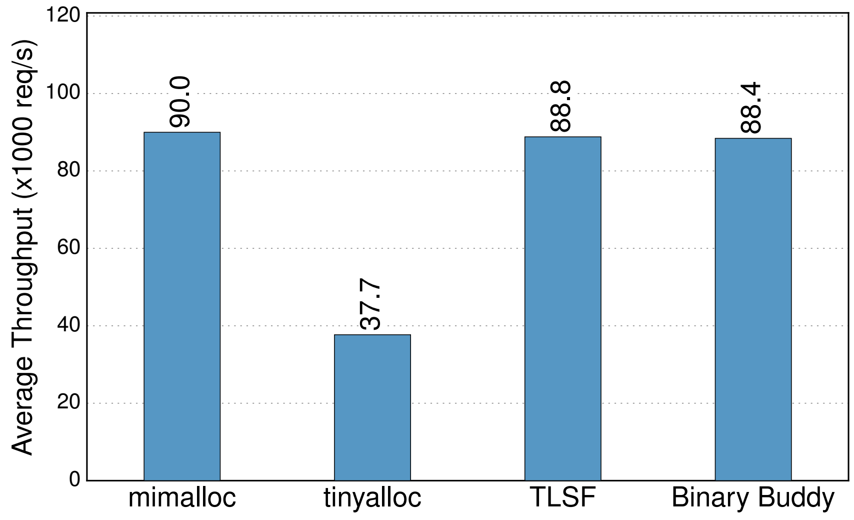

Figure 2.7: nginx throughput with different allocators

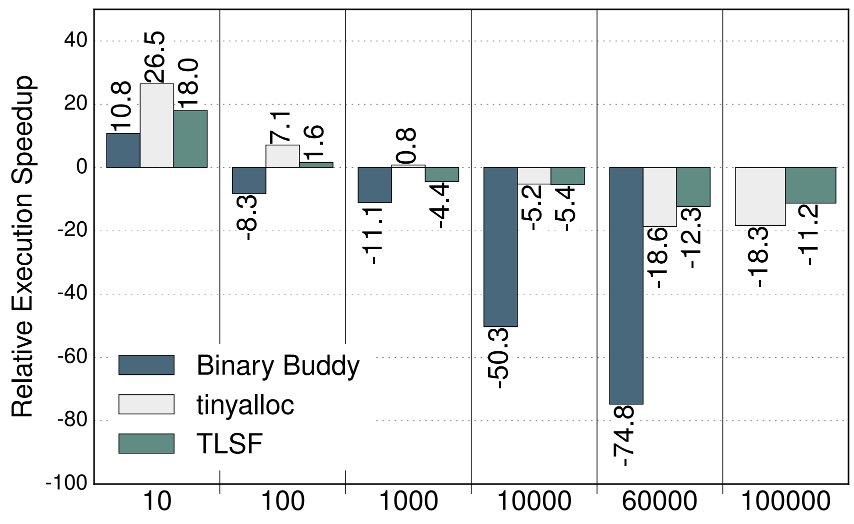

Figure 2.8: Execution speedup in SQLite, relative to mimalloc

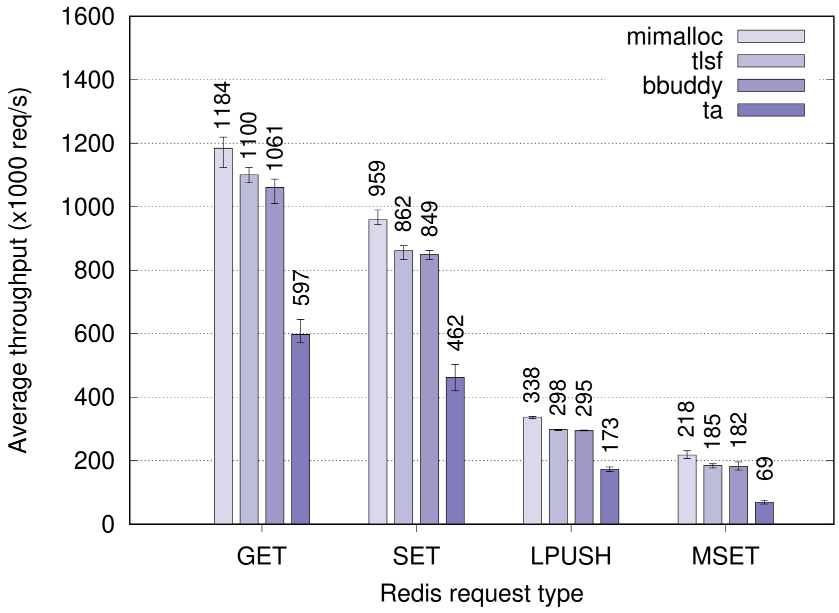

Figure 2.9: Average throughput in requests per second in Redis

3 Feature Support

Additional features are provided to UNICORE APIs and toolchain to ensure performance, portability, scalability and security. This section presents support for multi-target and for symbolic verification, now updated for a more diverse verification and validation support.

3.1 Multi-Target Support

A key feature and target of the project is to be platform and CPU architecture agnostic: that is, once a particular application has been ported to the system, deploying to a wide range of platforms and architectures should be a transparent process. To achieve this, in UNICORE we introduce a platform abstraction API that allows Unikraft to seamlessly build images targeting different platform types: virtual machines (VMs) (Xen, KVM/QEMU, Solo5 and Amazon Firecracker), containers (OCI and Docker) and bare metal (x86_64, ARM64).

The implementation of the API can be done as part of the Unikraft repository, a so-called *internal platform*, or as an external one, in which case all code resides in a completely external repo. There are no other major difference between these two; as way of example, the KVM platform is implemented as an internal platform, whereas the Solo5 one is an external platform (and repo). Whether internal or external, platforms must provide a Linker.uk file containing platform-dependent rules for linking the final image.

It's worth spending af few worth on the bare metal target. When using this target, in addition to selecting a CPU architecture, the build must contain code, in terms of an external platform, to support a specific hardware device. For instance, as of this writing, Unikraft supports the Raspberry Pi 3 B+ and the Xilinx Ultra96-v2 boards (though note that neither of these is yet upstream/public).

In all, the platform API has allowed the project to support a wide range of seemingly disparate platform targets, including different CPU architectures, devices, and virtualization platforms. The current support, while certainly not fully mature, allows us, in contrast to other unikernel projects (e.g., Rump, Hermitux) to *transparently* deploy an application, and its accompanying custome-build OS, on a wide range of different targets.

3.1.1 Host Environments

Unikernels are most often run in a fully virtualized environment, but lightweight virtualization (aka containers) environments are also possible. It is important to ensure security and robustness of the host environments particularly in the light of multiple hardware vulnerabilities discovered recently.

In order to improve host operating system support for mitigation of such hardware vulnerabilities, Unicore project develops enhancements to the Linux kernel that allow better isolation between different tenants in a multi-tenant environment, either the tenants are virtual machines or Linux containers.

The goal is to enable address space isolation and use of restricted address spaces in the Linux kernel for different privileged contexts. For example, processing of most of *VMExits* does not require full kernel mappings and therefore can be performed with a reduced page tables.

The following uses cases are considered as a good candidates for using address space isolation and restricted address spaces:

VMExit processing - a virtual machine running a unikernel require services of the host operating system. To receive such service, the VM does *VMExit* and the control is transferred to the hypervisor. In Linux/KVM environment, the hypervisor is the operating system kernel and the processing of the *VMExit* happens with entire kernel address space mapped. For most types of *VMExit* this is no necessary and only restricted part of the kernel mappings is required.

Secret memory areas - intended for applications to store secret information, e.g. private keys, or even the entire memory of the VM guest. Such areas can be completely unmapped from the kernel page tables and thus visible only to the process that owns the secret memory regions

Address space for Linux namespaces - Linux namespaces provide logical isolation of various operating system resources, such as mount points, system time and networking stack. The namespaces constitute the major building block of the lightweight virtualization also known as containers. For such use case, addition of a dedicated address space to network namespace would allow better privacy and security for the applications and unikernels running inside the container.

We focus on the implementation of the secret memory regions and the infrastructure required to support restricted address spaces for *VMExit* execution and for the Linux namespaces.

3.2 Validation and Verification Support

Given the small size of the unikernel source code an final image, symbolic execution is a way to automatically validate large parts of the unikernel μlibs and applications run on top of them.

We did preliminary investigation using symbolic execution engines (KLEE) and other verification tools incorporating symbolic execution (CBMC). Given the verification time required for symbolic execution engines, we plan on integrating multiple verification/validation techniques: formal verification, runtime security checks, implementation in safe programming languages. These techniques will be applied selectively by the use case beneficiary as a balance between performance and security coverage. Most of the work will start in the next period.

We rely on using symbolic execution, formal verification, secure language implementation and selective hardening (i.e. runtime security checks). The options are to be provided to the user and be advised to make a conscious choice of what to target the application for, aiming for a sweet spot of security and performance.

One of the main limitations of symbolic execution is the time taken to ensure validation of a given component. Formal verification is difficult to be done and requires extensive time, effort and knowledge from the specification writers. We plan to use symbolic execution for components where path explosion is absent or, more likely, marginally relevant. In other cases, symbolic execution may still be relevant for partial verification.

Method	Upside	Downside
symbolic execution	static analysis, mature, potential for full coverage	path-explosion problem, incomplete validation may offer insufficient guarantees
formal verification	strong theoretical foundation, secure by specification	large effort from developers writing specifications
runtime security checks	easy to implement, little developer effort (usually compiler flags)	no security guarantees, runtime overhead
secure programming language	reduced overhead (programming language support), easy to implement	no security guarantees, possible issue when linking together multiple programming languages

Table 3.1: Upsides and downsides of validation of verification methods

We call secure programming languages (Rust, Go, D) those languages that provide compile-time checks and language enforcement that prevent or reduce the risks of vulnerabilities. Implementing a given component in a secure programming language incurs little runtime overhead. The downside is they don't offer clear guarantees and the difficulty in binding together components written in different languages.

A summary of upsides and downsides of methods used for validation and verification is highlighted in Table 3.2.

Considering the different approaches, the goal is to have them all present and, consistent with the general approach of creating specialized images from existing UNICORE components, let users decide what is best for their use case. Their decision will be based on the weight given to either security of performance metric of the given build, with UNICORE being able to provide an approximate security metric and a clear performance one (such as image size or runtime overhead). The approach is highlighted in Figure 3.1.

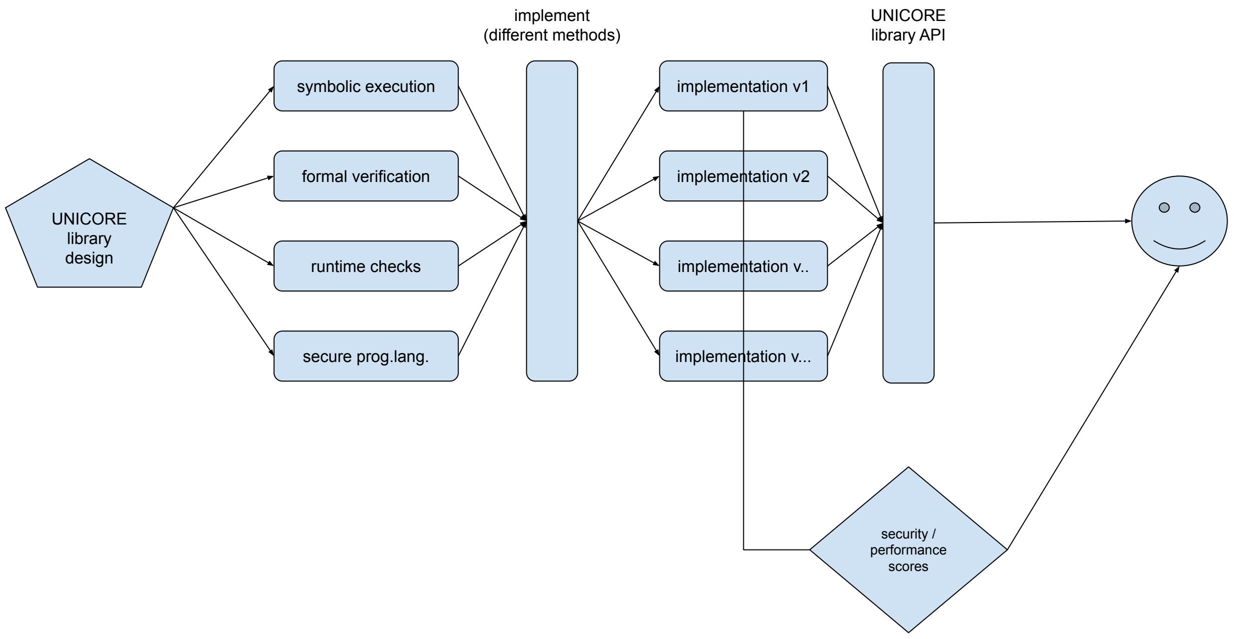

Figure 3.1: Using different methods to create specialized UNICORE images balancing security and performance

4 Development and Deployment Environment

Development and deployment relies on specific tools that configure, build, prepare the environment and run UNICORE unikernel images. The introduction of a platform abstraction API allow UNICORE to seamlessly build images targeting different platform types: linuxu, KVM, Xen and different CPU architectures (x86, x86_64, ARM). The development and deployment environment facilitates the creation of build images for these platforms.

4.0.1 Development and Deployment Using kraft

Development and deployment of UNICORE core unikernel and external libraries (i.e. existing ported source code) is facilitated by using kraft 2.4.2.

`kraft` is used for automated building of applications in a singular images. At the same time, `kraft` itself can be configured to meet the needs of your development workflow. If you are working directly the UNICORE API source code or a library then you can change kraft's behavior so that it recognizes changes which you make.

`kraft` uses environmental variables to determine the location of the core source code and all library pools. This is set using the following:

During phases of development which require modifying the core source code or an auxiliary library for the target application, kraft's runtime can be altered to facilitate varying developer requirements.

In the following example, both the core source code and an additional library, `mylib`, have are utilized for an application. However, their source has been modified and point to external locations. This is useful if you are doing local development or wish to work with private repositories:

```
specification: '0.4'

unikraft: file:///home/developer/repos/unikraft/unikraft@3a8150d

libraries:
  mylib:
    version: devel/new-feature
    source: git://git.example.com/lib-mylib
```

The kraft tool works with these remote and local Git repositories in order to handle version control. However,

Environmental variable	Default value	Usage
UK_WORKDIR	~/.unikraft	The root directory for all sources.
UK_ROOT	$UK_WORKDIR/unikraft	The core source code.
UK_LIBS	$UK_WORKDIR/libs	Library pool sources.
UK_APPS	$UK_WORKDIR/apps	Applications and templates.
KRAFTCONF	~/.kraftrc	The location of kraft's preferences file.

Table 4.1: kraft Environment Variables

wen using the kraft tool itself in during the configure and build steps, it is handy to stop it it from automatically running git checkout on these repositories. This is particularly useful when the source tree of the core or any other library has a dirty working tree.

To ignore warnings and proceed with a command, use the global flag `-C`:

```
kraft -Cv configure
```

To prevent kraft from checking out repositories entirely, use the global flag `-X`:

```
kraft -Xv build
```

Running and debugging unikernels can be accomplished largely with the use of `gdb`. This will build an un-stripped binary with debugging features enabled. This can be toggled with the `-d|--dbg` flag on kraft run. To start `gdb` itself, include the `-g|--gdb PORT` flag during the same run stage. Additionally, it is often useful to start the guest in a paused stage, accomplished with the `-P|--paused` flag:

```
kraft run -p kvm --gdb 4123 --dbg
```

4.1 Deployment using Binary Compatibility

Binary compatibility sits on the far end of the spectrum of performance versus porting-effort, removing the need to do any porting work: as long as the underlying OS supports the syscalls a particular application expects, the application should run out of the box (assuming that the application does not use dynamic libraries, or that, if it does, such libraries exist in the OS image). This approach is also beneficial when source code is not available, or when the dependencies list is simply too large or complex to justify porting.

To support this, we created a library called *syscall shim*: each library that implements a system call handler registers it, via a macro, with this library. The *syscall shim* automatically generates a generic system call handler that forwards system call requests to the corresponding handler function, and stubs missing system calls. The generic handler is executed as soon as the application does a binary system call request (e.g., `syscall` on x86_64). Similar to HermiTux, we save the caller registers' state on the stack, take the arguments from the saved state and hand it over to the appropriate handler function. When this handler returns, we modify the saved register state by writing the return code to the return registers. After we restored the register state, we return to the caller. Finally, we create an ELF loader library that is able to load and execute a single, statically-linked application PIE.

4.2 Evaluation

We evaluate the development and deployment environment in two ways: by measuring porting effort for new libraries using ever-improving toolchain and by measuring performance of binary compatibility apps who require little porting effort, but incur runtime overhead.

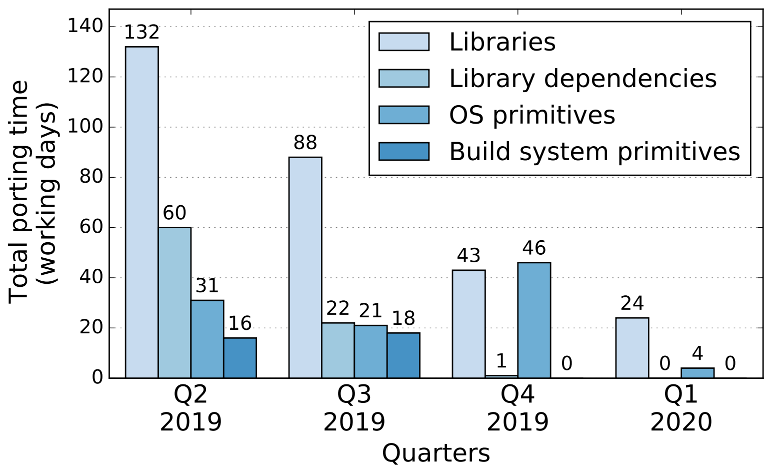

Figure 4.1: Developer survey showing that library porting times decreased over time.

4.2.1 Porting Evaluation

Because UNICORE micro-libraries provide a common code base for building specialized stacks, such manual porting is significantly less time consuming than past unikernels projects that would take in the order of months to be put together.

In fact, anecdotal accounts throughout the lifetime of APIs development to the fact that, as the common code base has matured, porting additional functionality has gotten increasingly easier. Admittedly, quantifying the actual man hours spent porting a library is a difficult exercise, e.g. because commit timestamps may hide the fact that, during the course of porting a library, significant time was spent porting one of its dependencies. Nevertheless, we have conducted a survey with all developers in the project's open source community (around 70) who have ported a library, and present the results here. In particular, we asked developers to roughly calculate the time it took to port an actual library or application, the time it took to port library dependencies (e.g., memcached requires libevent), and the time it took to implement missing OS primitives (e.g., the `poll()` function) or add functionality to the build system. We use git commit history to track when a port was started.

To show the evolution of the porting effort as the project matured, we plot the results of the survey in a timeline starting in March 2019 and ending in May 2020 in Figure 4.1; for ease of presentation, we further group the results in quarters. The figure confirms the anecdotal evidence that, as time progressed, the amount of time developers had to spend porting dependencies or implementing missing OS primitives has significantly decreased. While the time to port the actual application or library stays more or less constant, this is *precisely* the time that the approaches previously mentioned in this section, and in particular the externally-built archive one, directly address.

It is also worth noting that kraft has a `devel` command that provides a few helper tools that can produce skeleton Makefile.uk files, and that can auto-generate a file containing stubbed versions of all undefined references in a build (by relying on manual pages). Such tools help reduce porting time.

As a final note, UNICORE's support for a wide range of languages and their environments (standard li-

	Routine call	#Cycles	nsecs
Linux/KVM	System call	604.62	232.55
	System call (no mitigations)	142.31	54.74
KVM	System call	85.0	32.69
Both	Function call	6.0	2.31

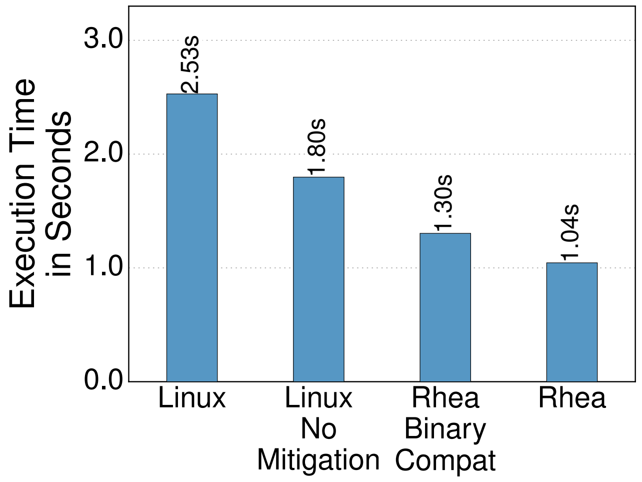

Figure 4.2: Binary compatiblity performance versus Linux for a sample application (SQLite).

braries, garbage collectors, etc.) means that a number of projects based on these (e.g., Intel's DNNL/c++, Django/Python, Ruby on Rails/Ruby, etc.) should work either out of the box or should require little additional effort to get running.

4.2.2 Binary Compatibility Evaluation

What are the downsides of binary compatibility? The main performance differences arise in how system calls in the application are handled: with binary compatibility, an indirection layer is used. With porting, a syscall is essentially a function call.

The table below shows the results of microbenchmarks that measure the costs of executing a `no-op` system call with each approach, and contrasting them to in Linux.

Binary compatibility has a tenfold performance cost compared to a function call when the indirection layer is used (OSv, Rump, HermiTux all rely on binary compatibility). The syscall costs for Linux are twice and 8-fold more expensive than our indirection layer, depending on whether KPTI and other mitigations are enabled. Without mitigations, most of the cost comes from changing protection levels; with mitigations, a TLB flush is needed on every syscall.

To understand how these affect application performance, we benchmarked SQLite (over RamFS, 60K insert operations) in two versions: a ported version, where all syscalls are function calls to UNICORE libraries, as well as binary compatibility (see Figure 4.2). As shown, binary compatibility does not come for free, as SQLite over glibc on top of our shim takes twice as long as the native run. For comparison, unikernel build outpaces SQLite on a Linux VM, even in a kernel without security mitigations.

For virtual machines running a single application, the syscall costs of Linux running as a guest are likely not worth their costs, since isolation is also offered by the hypervisor. In this context, unikernels can get important performance benefits by removing the user/kernel separation and its associated costs. The indirection required for binary compatibility reduces unikernel benefits significantly.

5 Orchestration Tools Integration

In spite of the main goal of UNICORE is not to develop an orchestrator which allows to launch unikernels it is useful to provide tools that permits to adopt and deploy an application in a unikernel environment. In order to ease the adoption of unikernels and companies do not have to retrain staff we are working on the integration into existing orchestrator tools. Also, this integration will permit to run unikernels in multiple platforms like KVM, Xen or containers. The tools chosen to integrate into are OpenNebula which is a very popular virtual machine orchestrator and Kubernetes which is a container orchestrator.
The Unikraft toolchain will create a unikernel file which is the executable where the application runs so, the goal is to launch the resulting unikernel file via orchestrator. As this file will be treated as a common kernel file the approach is to define the kernel file before launch the instance in the hypervisor. Another approach will be to embed the resulting unikernel file into disk image an run it as if it were a virtual machine.

5.1 OpenNebula Orchestrator

The OpenNebula orchestrator is a virtual machine manager that runs on top libvirt. Libvirt has the advantages that it can manage Xen or KVM, so it is possible to run the unikernel on KVM or Xen indistinctly. Launching a unikernel from OpenNebula over the KVM hypervisor is as simpler as defining the kernel with the unikernel which will result in the addition of the *-kernel parameter* in the command line with the following:

```
qemu-system-x86_64 -kernel unikernel-file.exe
```

The next step regarding this orchestrator is to explore the possibility of add firecracker support as a hypervisor back end.

5.2 Kubernetes Orchestrator

Following the previous approach unikernels have to be able to launch from Kubernetes, the most popular container orchestrator. A container environment is not aware of the kernel, so we've been exploring the possibility to launch virtual machines from Kubernetes as if they were container. To do so we found two possibilities: create a new custom resource definition that permits to launch unikernels or use an already developed add-on for Kubernetes that allows to launch virtual machines from it. The second option has been chosen by using kubevirt.

5.2.1 Kuvebirt

Kubevirt is a virtual machine management add-on for Kubernetes. By now is a work in progress development but the project and its community is very active. As OpenNebula or other virtual machine orchestrators Kubevirt leverages the libvirt API to launch virtual machines over KVM which makes easier the integration of unikernels into kubevirt. Currently there is no support for define a kernel parameter in Kubevirt so, the work is focused on develop the feature in order to run unikernels as simple VM with a custom kernel (the unikernel).

Kubevirt uses yaml files to define the specifications of the virtual machines, the same as Kubernetes does with containers. The work will consist in create a new parameter in the specification file as it follows:

```yaml
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
metadata:
  labels:
    special: vmi-alpine-kernel
  name: vmi-alpine-kernel
spec:
  domain:
    firmware:
      kernelBoot:
        volumeName: kernelDisk
        # kubevirt assumes that the volumeName named above will container a vmlinuz (kernel) and
    initrd.img for boothing. If not, then we should raise an error
        cmdline: "<user cmdline arguments>"
  volumes:
  - containerDisk:
      image: unikernel.exe
    name: kernelDisk
    # The assumption is that the container will container a vmlinuz and initrd.img file which will
     then be used by kubevirt
```

Listing 5.1: Kubevirt spec file with kernel parameter

The above listing shows an example kubevirt file where appear the new spec parameter *kernelBoot* with its *volumeName* that references to the file *unikernel.exe* defined in the *containerDisk* parameter.

6 Code Release

UNICORE tools and core API are released under BSD license. As tools diversified, multiple repository have been set up. Code can be accessed at:

- `https://github.com/unikraft/kraft`: kraft and build / deploy automation
- `https://github.com/unikraft/kraft`: core libraries (Unikraft)
- `https://github.com/cs-pub-ro/syzkaller/`: Syzkaller updates integrating UNICORE unikernels support

7 Conclusion

This document presented the current state of the UNICORE toolchain and development and deployment environment. UNICORE unikernel images benefit from fast porting time, automatic building, development / deployment facilitating tools. Modularity of software components allows specialized unikernel images with performance and security benefits.

Dependency analysis and automatic build tools shorten the time taken to add new components to the UNICORE software set with binary compatibility filling the spot of proprietary or difficult to configure and build applications. UNICORE builds run on multiple platforms and multiple architecture, with a modular platform API paving the way to extend that. Verification and validation support will strengthen the security of unikernel applications, more than the inherent reduction in attack surface. The integration of modern orchestration will increase the use cases and availability of UNICORE unikernel builds.

Development is done following an open source model, with code releases being available.

References

[1] “Clang static analyzer.” [Online]. Available: https://clang-analyzer.llvm.org/

[2] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, “A Binary-Compatible Unikernel,” p. 15, 2019.

[3] N. kumar, “Profile-guided optimization (pgo) using gcc on ibm aix,” https://developer.ibm.com/technologies/systems/articles/gcc-profile-guided-optimization-to-accelerate-aix-applications/, last accessed: 25 March 2020.

[4] J. Savage and T. M. Jones, “Halo: Post-link heap-layout optimisation,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 94–106. [Online]. Available: https://doi.org/10.1145/3368826.3377914

[5] Github, “Xen Minimal OS - Memory management related functions,” https://github.com/sysml/mini-os/blob/master/mm.c.