PHP2Uni: Building Unikernels using Scripting Language Transpilation

Thomas Pasquier
Harvard University
Cambridge, USA
Email: tfjmp@seas.harvard.edu

David Eyers
University of Otago
Dunedin, New Zealand
Email: dme@cs.otago.ac.nz

Jean Bacon
University of Cambridge
Cambridge, United Kingdom
Email: jean.bacon@cl.cam.ac.uk

***Abstract*—Unikernels are a rapidly emerging technology in the world of cloud computing. Unikernels build on research on library operating systems to deliver smaller, faster and more secure virtual machines, specifically optimised for a single application service. These features are especially useful in cost or resource constrained environments. However, as with any new technology, early adopters need to master many technical details, and understand many aspects of the mechanisms used to build and deploy unikernels. Both of these factors may slow adoption rates. In this paper, we present our initial experiments into the use of an approach for building unikernels that is accessible to those whose technical expertise is focused on web development. We present *PHP2Uni*: a tool chain that takes a website built from PHP files—PHP remains the most widely used web language—and builds a resource-efficient unikernel image from them, while requiring little knowledge of the underlying operating system software complexity.**

I. Introduction

Unikernels are generating significant interest as a potential breakthrough in virtualisation technology, due to their improved security, small footprint, fast boot time and whole-system optimisation (see § II). These benefits align very well with service-oriented architectures [1], microservice architectures [2], the needs of the Internet of Things (IoT) [3], and edge/fog computing [4].

However, most current unikernels require operating system (OS) software experience and advanced programming skills in languages not typically used by web developers. For the fastest potential adoption in application domains such as the IoT, the skill sets and resources offered by today's web application developers need to be harnessed. Indeed, in domains such as Fog computing, programmability is often considered a challenge that needs to be addressed [5]. PHP was ranked as seventh most used language in 2015 and 2016 [6], [7]. When considering web development, PHP is positioned far above languages such as OCaml, Haskell, Erlang or even C++, that are the targets of the popular unikernel projects. To facilitate adoption, unikernels should be made more accessible to typical web developers, and build upon languages and libraries used in the web development community.

One approach for building a unikernel web application is to compile a PHP interpreter as a unikernel. This is the approach adopted, for example, by *rump-php*.[1]

In this paper we propose the use of transpilation techniques, made popular a few years ago by Facebook [8], to build unikernels from PHP code. Our approach—prototyped in *PHP2Uni*—is to transpile PHP code into C++ classes, and build unikernel images from them. A key insight of our approach stems from the combination of transpilation and unikernels, which we believe could facilitate a wider adoption of unikernels, and ease deployment of innovative solutions in resource-constrained environments.

Currently PHP2Uni is able to build IncludeOS [9] and rump kernel [10] unikernel images (see § III-A and § III-B) from standard PHP scripts.

In this paper, we make the following contributions: 1) we demonstrate the feasibility of transpilation in the context of unikernel building; 2) we present use cases and benefits that motivate this approach; 3) we discuss the engineering choices made during development and report on our experience; 4) we do a preliminary comparison of our approach with competing deployments in terms of computation and memory consumption.

The rest of the paper is organised as follows: § II gives a brief overview of virtualisation techniques, unikernels and transpilers. § III describes the implementation of our prototype. § IV presents early-stage evaluation results. § V discusses our plan for future work, and some related challenges that we identified. We conclude and discuss future work in § VI.

II. Background

In this section we briefly discuss various contemporary virtualisation techniques, namely traditional Virtual Machines (VM), containers and unikernels. We then introduce unikernels and transpilers, indicating the relevant literature.

A. Virtualisation techniques

Since the surge in Intel x86-based virtualisation that powers today's cloud computing data centres, full-machine virtualisation has been the dominant method used to provide the necessary isolation for server applications. However, recent advances in container technology on Linux have demonstrated viable alternatives.[2] Applications and their environment are isolated on top of a shared OS kernel and some common libraries. OS-level isolation mechanisms are used to separate applications, such as LXC [11] in the Linux world. Commercial solutions such as Docker[3] due to increases in efficiency, and effective deployment, are gaining momentum [12], [13].

However Unikernels [14] are emerging as a disruptive technology that may change the path of mainstream virtualisation yet again. Typically, VMs or containers run a single service/application, yet are built upon a full traditional software stack.

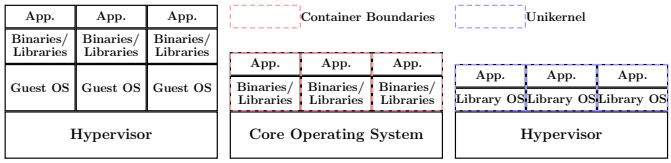

Fig. 1. Comparison of Virtual Machine, Container and Unikernel approaches.

This software stack contains a number of features useful for a general OS, but not required in their actual deployment contexts. Therefore, in a unikernel, the application and the OS merge into a single unit, that only contains the features strictly required to run the application. Unikernel particulars are further discussed in § II-B.

As shown in Fig. 1, a) the hypervisor may support separated VMs, each with its own OS; b) a single shared OS may support containers that isolate applications; or c) the hypervisor may directly support applications, each built from a library OS (Unikernel).

B. Unikernels

A unikernel runs a single application directly over the hypervisor without the need for a stand-alone operating system. Unikernels are single address space systems that bundle together an application and a selection of system components relevant for that particular application, into a single image. Unikernel applications are built using a library OS [15], [14] and only contain the minimum functionality required for the application to work. Like containers, unikernels are easily deployed, and come with a number of advantages:

1) extremely small footprint, which helps optimise resource consumption in a cloud or IoT environment [16];
2) significantly reduced attack surface, by removing all components not required to run the application [17];
3) extremely fast deployment and migration [18], for example to support edge computing;
4) whole-system optimisation targeted to the specific application [19].

Further, when compared with container-based solutions unikernels are self-contained, potentially mere megabyte-scale images that have no dependency on the underlying OS. Recent advances[4] are integrating unikernel deployment in the container-based solution Docker. UniK[5] also aims to facilitate the deployment of Unikernels with integration into Kubernetes and Cloud Foundry. These projects take useful steps toward mainstream use of unikernels.

There are a number of existing projects that focus on different aspects of unikernel technology: HaLVM[6] in Haskell, Mirage OS [19] in OCaml and IncludeOS [9] in C++ all take a clean-slate approach; ClickOS [20] emphases speed; and rump kernels [10] are built upon the rump kernel drivers, which provide compatibility with legacy POSIX software.

C. Transpilers

Transpilers (also referred to as transcompilers or source-to-source compilers) are software programs that take source code in a given language as input and generate the equivalent source code in a second language at an equivalent level of abstraction. Transpilers can also be used to handle API changes, such as Coccinelle [21], which was developed to ease maintenance of the Linux Operating System.

Another use is the translation of source code between different versions of the same language (for example, the tool provided by Microsoft to translate VB6 code when VB.net came out).

The transpiler more directly relevant to this paper is HPHPc [8], a Facebook-developed transpiler from PHP to C++. Domain Specific Languages [22] are also well-known application domains for transpilers. Transpilers have also gained popularity in the JavaScript world with projects such as Dart,[7] CoffeeScript,[8] and Flow,[9] among many others.

In the rest of this paper, we examine how well-known and understood transpilation techniques can be used to ease the development of unikernel-based services. We have chosen PHP as a proof of concept language due to its popularity (see § I) and the large number of PHP deployment solutions (see § IV-A).

III. Implementation

In this section, we discuss our implementation. We first introduce our current two target unikernel architectures:

- IncludeOS [9];
- Rump kernel [23].

The purpose of providing two targets is to 1) demonstrate that our proposed technique can easily be generalised, 2) to allow pros and cons of different unikernel environments to be evaluated. We then discuss the transpilation process that transforms websites built from PHP scripts into C++ applications, that are compiled against the two target unikernel architectures. In this prototype we built on, extended, and fixed some bugs in php2cpp.[10] Php2cpp is a simple transpiler with a small codebase. Finally, we discuss the limitations of the current prototype.

A. IncludeOS

IncludeOS is a single-tasking OS, specifically designed for a virtualised environment. Developers write code directly in C++ that builds against the library OS (`#include <os>`), with a GCC-customised tool chain that generates the corresponding OS images. The build system extracts the required functionality from the pre-compiled OS library at link time and forms a single executable binary file. The boot sector is attached and together forms the image-file. This resulting image is targeted to run on virtualised x86 hardware. The image can run in various virtualisation environments such as QEMU, KVM, VMware, or Xen.

IncludeOS does not have a program loader, and therefore does not use the `main` symbol as an entry point. A `Service`-class is provided and the developer must implement the `Service:start` method which is called after the OS completes its initialisation. Our prototype transpiles PHP applications into classes that are instantiated and manipulated within the `Service::start` method.

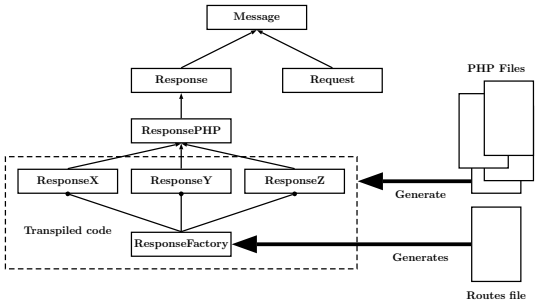

Fig. 2. PHP2Uni's architecture and key classes.

B. Rump Kernels

The rump kernel project is another, more mature, solution to build unikernel images. The rump kernel tool chain provides NetBSD drivers as portable components in order to build unikernel images from virtually any POSIX application [10], as a POSIX compliant interface is provided.

The advantage of the rump kernel project is its maturity when compared to IncludeOS. However, it is designed to run arbitrary POSIX applications, whereas more specialised unikernels such as IncludeOS will allow better performance to be obtained. We show this to be true in terms of memory footprint and computational performance in § IV.

The code obtained after transpilation is extremely similar. We discuss relevant dissimilarities in the subsections that follow.

C. From PHP files to a unikernel

PHP2Uni Service::start implementation contains the following code:

```
/* initialise service and create socket */
/* [...] */
/* handle request */
sock.onAccept([](net::TCP::Socket& conn){
    std::string str_request = conn.read(1024);
    printf("SERVICE got data: %s\n\n",
           str_request.c_str());
    // parse the request
    http::Request req(str_request);
    http::ResponseFactory rf;
    // retrieve the response
    http::Response res = rf.create(req);
    conn.write(res.get_header());
    conn.write(res.get_body());
  });
```

Listing 1. Extract from Service::start, request handling.

The HTTP request onAccept is read from the socket and parsed to generate an http::Request instance. The request instance is passed to http:ResponseFactory that instantiates the class corresponding to the method/URI pair (e.g. GET /index). These instantiated classes correspond to the transpiled PHP code.

Fig. 2 represents the class structure implemented by PHP2Uni on top of the IncludeOS library. We now describe the two steps necessary for the building of the virtual machine through transpilation:

Transpiling http::ResponseFactory: Parts of the http::ResponseFactory class are generated through the transpilation of the routes file. A routes file will contain information such as the following:

```
GET  /index  index.php
GET  /hello  lib/hello_get.php
POST /hello  lib/hello_post.php
GET  /       index.php
```

Listing 2. Example routes file.

The transpiler implements the branching condition corresponding to the method/URI pair, in order for the factory to instantiate the right class. Each unique PHP file is transpiled to the corresponding class at compile-time.

Alternatively, instead of pointing the transpiler to a routes file, the transpiler can be applied to a single PHP file that may implement an equivalent functionality through a switch construct over $_SERVER['REQUEST_URI']. In such a scenario, http::ResponseFactory is not necessary and a unique class inheriting from php::ResponsePHP is instantiated to handle all requests.

However, in order to maintain similar behaviour to a standard PHP server, we would like to be able to build a web application using several separate script files. The route file allows this behaviour to be supported, while not needing to modify the PHP scripts themselves. This allows the same application to run as PHP scripts, a PHP2Uni IncludeOS image or a PHP2Uni rump kernel image (we show deployment of the same scripts over several solutions in § IV). We believe this to be a useful division between development-time use of PHP scripts, with rich logging and diagnostic support, and production use of PHP2Uni unikernels.

Transpiling pages: php::ResponsePHP implements built-in PHP functions (e.g. mktime, hexdec etc.). The transpiler generates from PHP files, the classes that inherit from php::ResponsePHP. Each of these transpiled pages corresponds to an entry in the routes file. The transpilation process from PHP files to a php::ResponseX class in our current prototype is as follows:

- pass through the source files recursively to handle file inclusion, i.e. include / require / require_once;
- scan the source files for class declarations and transpile them as inner classes of the php::ResponseX class;
- scan the source files for function declarations and transpile them as methods of the php::ResponseX class;
- finally, parse the core PHP script (i.e. the code that is not part of a class or a function).

D. Limitations

At the moment only a small portion of built-in PHP functions and classes are supported (e.g. base64_encode, Exception etc.), but we are increasing this coverage steadily. There is no inherent limitation on this front, simply an engineering resource constraint. This paper only presents a proof of concept of our proposed approach, and the full coverage of the PHP built-in functions and classes which were developed and expanded for more than two decades is beyond the aim of this paper. We further discuss this particular issue in § V-C. However, we believe this approach shows promise, based on our evaluation results. In the next section we compare the approach adopted here with alternative deployment solutions. We show that PHP2Uni, especially the version that uses IncludeOS, has the potential to occupy a particular niche in

VM Type	VM Size
LAMP Stack Ubuntu VM[11]	~400 MiB (×200)
rump-php	~63.5 MiB (×30)
PHP2Uni-rump	~23 MiB (×10)
PHP2Uni-IncludeOS	~2 MiB

TABLE I
SIZE OF VIRTUAL MACHINES

cloud or IoT environments where computational resources are extremely constrained, or represent a financial cost that should be minimised.

IV. EVALUATION

We performed evaluations in order to identify clearly the benefits of the proposed approach. The tests were run on an i7 Ubuntu 14.04 LTS machine with 8 GiB of RAM. An in-depth report on IncludeOS performance is given in [9], as well as subsequent papers by the same authors. In this paper, we particularly focus on comparing PHP2Uni against other PHP deployment solutions.

We compared the VM size of our unikernel with that of the standard Linux stack in § IV-B. When comparing performance or memory footprint (in § IV-B and § IV-C respectively) we run either directly above the OS or via QEMU/KVM for the unikernel solutions.

A. Considered deployment solutions

For this paper, we identified and compared a number of open-source platforms available for the deployment of PHP applications:

Apache2:[12] Using Apache httpd has been the traditional way to deploy PHP applications. This is our baseline against which other solutions are compared. We used an 'out of the box' configuration without any particular optimisations.

```
1 FROM php:5.6-apache
2 COPY config/php.ini /usr/local/etc/php/
3 COPY src/ /var/www/html/
```

Listing 3. Dockerfile for PHP server.

Docker-Apache2:[13] We ran a Docker httpd/PHP stack for comparison: we used the simplest configuration possible. The Docker configuration file that was used is shown in Listing 3. Typical deployment will be the so-called *LAMP* stack (i.e. Linux, Apache, MySQL, PHP), in a VM running on top of an IaaS platform. Again, we used an 'out of the box' configuration without any particular optimisations.

rump-php:[14] This is a PHP interpreter compiled against the rump kernel. It provides a much smaller footprint image when compared with a standard LAMP VM. Deployment would generally involve a rump-Nginx server handling HTTPS requests, multiple rump-php instances, and a rump-MySQL instance (alternatively, the MySQL backend may be deployed in a more traditional fashion). We used the default configuration without any particular optimisations, taken directly from the rump package GitHub repository.

HHVM:[15] HHVM is the Hip Hop Virtual Machine [24]—an open source solution developed by Facebook. HHVM is the successor of the HPHPc [8] transpiler, which uses ahead-of-time compilation. HHVM uses just-in-time compilation [25] in order to provide improved performance when compared with a more traditional PHP deployment environment. We used the default configuration provided when building HHVM from their GitHub repository.

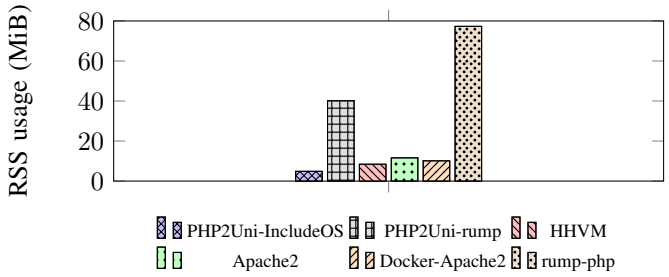

Fig. 3. Memory usage (RSS and as reported by docker stat) comparison across solutions.

PHP2Uni-IncludeOS and PHP2Uni-rump: This is the solution described in this paper, where we transpile standard PHP code to C++ and compile the resulting code into an extremely small unikernel image. Again, we did not try to provide particularly optimised configurations. Deployment as part of a multi-tier architecture is identical to rump-php, simply replacing rump-php instances, with PHP2Uni instances. Details of the configuration can be found on GitHub, please see the Availability section.

The authors acknowledge that the tested solutions could be fine-tuned to potentially perform better. However, the observed differences across solutions is significant enough that we believe the results to be relevant, regardless.

B. Memory footprint

Table I shows the difference in memory requirements for VMs necessary to run a PHP application. The Ubuntu VM runs the whole LAMP (Linux, Apache, DB, PHP) stack rather than simply httpd + PHP. In terms of size, the effectiveness of the unikernel approach is clear: note that a PHP2Uni-IncludeOS image is 200 times smaller than a traditional LAMP image.

Fig. 3 shows the memory consumed by the four solutions discussed in § IV-A. The figure reports the Resident Set Size (RSS), which is the memory actually allocated to a process, as opposed to the Virtual Memory Size (VMS), which includes swapped data and shared libraries. For the Docker instance the value is that reported by docker stat. We see that PHP2Uni-IncludeOS has memory consumption in the same order of magnitude as the comparable solution running directly above the OS. On the other hand, rump-based solutions consume much more memory (in line with the VM size reported in Table I). However, compared to solutions running directly over the OS, they benefit from much stronger isolation. IncludeOS has been designed with memory constraints in mind, which explains its very small footprint.

C. Performance

Fig. 4 shows a performance comparison for the four solutions presented in § IV-A. The two microbenchmark applications tested are index, representing a typical PHP front-end page with a mix of static and dynamic content as well as user submitted parameters; and primes, which is a computation-heavy PHP script, that computes all the prime

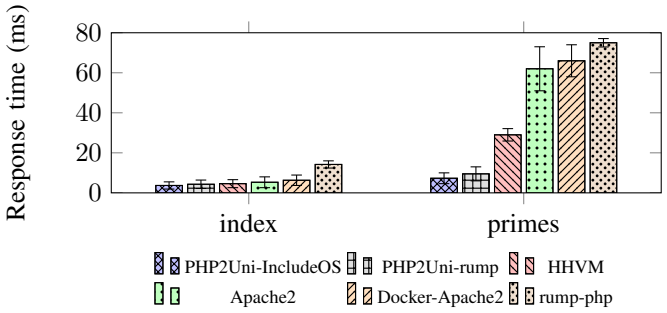

Fig. 4. Performance comparison across solutions.

numbers between 3 and 5,000. index aims to test PHP code that produces a high throughput in terms of response speed, whereas primes is an example of a workload that requires server-side computation to be performed.

We note that many PHP scripts that have been deployed today will likely be throughput-limited by the services that they depend on. For example, many scripts will interact with a back-end database system. Waiting on external services does not differ between the runtime models that we are examining. In such scenario, the performance is dependent on the local computation as evaluated in index and prime and the latency introduced by the external service.

PHP2Uni has similar performance to the other approaches when dealing with web pages that requires minimum computation (i.e. index). However, there is a clear performance gain when relatively computation-intensive jobs need to be handled (i.e. primes). This result is not surprising as one would expect a C++ compiled program to have better performance than an interpreted PHP script for this kind of workload.

D. Discussion on deployment strategy

We see the solutions presented in § IV-A as a spectrum with various pros and cons. Our approach is not overall better or worse, but may fit particular needs better than the alternatives.

We compare a spectrum of deployment approaches, from a full LAMP stack VM image to a PHP2Uni IncludeOS image. These approaches are compared across five dimensions: the memory footprint, the performance, the isolation strength in a shared environment, the size of the software stack, and the behaviour of the code once in deployment compared to the development environment. Details are discussed below:

Memory footprint: As discussed in § IV-B, memory usage is one of the most significant factors when comparing the various approaches. This factor may prove of extreme importance in two scenarios: financial constraints and resource constraints. In a 'pay for use' service model, a smaller image signifies lower cost, therefore compromises on other aspects may be acceptable. In constrained environments (e.g. fog-cloud [26], mobile-cloud [27] etc.) similar compromises may be acceptable in order to minimise resource consumption.

Performance: As discussed in § IV-C, we can see that transpilation techniques when compared to interpretation of PHP script provide improved performance. Again, in exchange for other trade-offs, this could prove useful in financially or computationally constrained environments.

Isolation: Here we make a distinction between traditional virtualisation techniques and container-based solutions. Container-based solutions aim at reducing resource consumption by sharing common parts of the OS and software stack at the expense of stronger isolation.[16] However, unikernels as an emerging approach, provide both smaller memory consumption and strong isolation. The recent acquisition by Docker of Unikernel Systems and development of solutions such as UniK for the Kubernetes world (see § II-B), indicate a growing interest in such technology as a possible alternative to containers. This is especially true since rump-based solutions can run virtually any POSIX application, thus occupying the same market niche as container-based solutions and proposing an interesting alternative.

Size of software stack: VM-based solutions rely on a general-purpose OS and the software stack associated with the environment, in order to run applications. Container-based solutions, while reducing memory footprint by sharing the OS and part of the software stack, still rely on that same general-purpose software stack. Rump-based solutions rely on a library OS that builds upon components compatible with POSIX applications. RumpPHP remains dependent on the original code of the PHP interpreter. Finally, the two PHP2Uni solutions are each based on a small unikernel stack, and no legacy interpreter. IncludeOS goes even further, with a minimalist set of features sufficient to provide services, rather than supporting the POSIX legacy. The definite advantage of a smaller software stack is the drastic reduction of the number of 'moving parts' and, in consequence, of things that can go wrong. As discussed in § II-B this is a potential net gain in terms of security (although it needs the underlying unikernel technology to have matured enough), but also in terms of performance and optimisation.

Deployment behaviour: This is one of the main trade-offs of unikernel-based solutions. In the case of development of PHP solutions, the natural development cycle involves running a webserver directly on the developer's local computer, with a quick develop/test cycle. The unikernel and standard legacy OS environments may present different behaviours that require further integration tests. This is further aggravated by the inclusion of transpilation. As pointed out in the Facebook HHVM paper [24], transpilation of large programs is slow, and developers tend to rely on standard interpretation solutions during development. When moving to the deployment environment, issues often arise due to differences in behaviour of the underlying platform. As a consequence, it requires a more complex integration phase, reducing the overall development productivity. This led Facebook to move from transpilation [8] to just-in-time compilation [24] as the gain in computational performance did not justify increased development cost. However, by adding a unikernel solution to the transpilation approach, in addition to an increased computational performance, we also reduce memory usage significantly, and provide stronger isolation guarantees. We perceive this as having a major impact in an as yet relatively small, but increasing, niche, namely extremely constrained environments.

There are a number of potential scenarios where having

a VM with an extremely small footprint, coupled with transpilation of some common language, could prove extremely beneficial. These include cloudlet [28], fog-computing [26] and other cloud decentralisation approaches, where cloud services are migrated closer to the end-user could benefit from the approach. These target environments may prove to be more resource-limited than typical cloud platforms. In scenarios where applications migrate from the cloud to devices on the edge, a standard PHP script could be running in the cloud, to generate equivalent applications for edge deployment by transpilation, in order to save resources. A less mainstream scenario for emergency situations is typified by cloudrone [29], where drones flying above a disaster area provide network and embedded cloud services (e.g. Facebook's Safety Check).

The addition of transpilation techniques is seen as a means to open up such techniques to existing legacy web applications and frameworks, but also in a language (and potentially a wider range of languages in the future) familiar to a wide range of web-developers. In particular, a direct use of IncludeOS would represent a major departure from the skill sets of the vast majority of web-developers, which may in turn increase development costs.

V. Future Work & Challenges

We believe that our approach presents an interesting alternative to developing services in resource or cost-constrained environments. In this section we discuss interesting challenges, and future directions that we identify as worth exploring.

A. Unikernel maturity

IncludeOS is itself a work in progress and the API and underlying implementation is neither stable nor feature complete. We aim to keep PHP2Uni up-to-date with IncludeOS evolution and to take advantage of features/improvements as they appear. The rump kernel toolchain appears to be more mature in comparison with the relatively younger IncludeOS and was easier to use within our workload. However, the extremely small size of IncludeOS images make this solution the ideal one for deployment in resource constrained environments.

B. Transpilation versus just-in-time compilation

HHVM [24] uses just-in-time compilation techniques [25], in order to address shortcomings of the HPHPc approach. The problems addressed by HHVM were: 1) PHP being a dynamically typed language, types are sometimes unknown when used with transpilation, which leads to inefficiency. PHP2Uni relies on type hints. Type hints consist of comments in the PHP code that describe the type of a given variable. The transpiler uses these hints to decide which type to assign to the corresponding C++ code. In the case when a hint is not present, the transpiler determines the variable based on the type of the value being assigned. This is especially useful when declaring classes or functions, where it may be difficult to resolve the type of a given parameter. Hints can generally be extracted from comments surrounding the code when following convention. We could, for example, easily imagine extracting this information from phpDocumentor (a tool to automatically generate documentation from source code) comments.[17] HHVM addresses this issue through just-in-time compilation, as it has access to runtime values and can therefore optimise the code. 2) As transpilation takes time for very large programs, interpretation via HPHPi was used during the development life cycle. But the behaviour of the program compiled by HPHPc in deployment differed from the PHP code running over the (HPHPi) interpreter. HHVM addresses this issue by being used as both the development and the deployment environment. HHVM has the distinct advantage of providing the familiar and rapid PHP development environment, while providing a very significant performance improvement. PHP2Uni does not target large applications but rather, smaller and simpler micro-service applications that need to run under cost or resource constraints.

In such scenarios, we believe that the clear gain in terms of memory, compared to traditional VM images, and in terms of isolation compared to containers, outweigh the disadvantages. However, work on the development environment may focus on guaranteeing the preservation of a familiar workflow to developers.

C. Extending existing languages or a DSL?

The advantage of using existing scripting languages is that the same scripts can be used to instantiate the application to run in different environments. Also, developers do not need to learn a new set of skills. However, as discussed, transpilation may lead to slightly different behaviour from the interpreted language. We may have to implement legacy features developed over several decades. In addition, we may want to extend the language to take advantage of features specific to the unikernel environment. Therefore, a viable alternative may be to consider the development of a DSL language specifically designed to write micro-services than can be instantiated as unikernels, or to extend our work to support a language such as Dart.

D. Deployment tools chain

Finally, all the previous points will be explored through the development and implementation of a deployment solution that is able to migrate micro-services across several deployment environments, choosing the appropriate method among those listed in § IV. In particular, we will look at automatic migration from traditional cloud platforms to edge devices. We aim to look at extending/modifying tools such as UniK[18] or Dokku,[19] trying to provide a streamlined and convenient experience when deploying applications.

VI. Conclusion

This paper presents our early efforts in using transpilation to improve the accessibility of unikernel technology, and explores how such deployment solutions may fit in the current deployment spectrum. We have demonstrated that such an approach has potential in cost- and/or resource-constrained environments. More importantly, widely available web developer skill sets are built on, rather than requiring specialised training in new system software. This will hopefully lead to more rapid acceptance and use of unikernel technology. We have released the early prototype as open-source software, and plan to develop it further as we gather feedback.

ACKNOWLEDGMENTS

This work was supported by UK Engineering and Physical Sciences Research Council grant EP/K011510 CloudSafetyNet: End-to-End Application Security in the Cloud. We acknowledge the support of Microsoft through the Microsoft Cloud Computing Research Centre.

REFERENCES

[1] R. Perrey and M. Lycett, "Service-oriented architecture," in *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*. IEEE, 2003, pp. 116–119.

[2] J. Thones, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] J. Singh, T. Pasquier, J. Bacon, H. Ko, and D. Eyers, "Twenty security considerations for cloud-supported Internet of Things," *IEEE Internet of Things Journal*, vol. 3, no. 3, pp. 269 – 284, 2016.

[4] I. Stojmenovic and S. Wen, "The Fog computing paradigm: Scenarios and security issues," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. IEEE, 2014, pp. 1–8.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," 2016.

[6] S. Cass, "The 2015 top ten programming languages," *IEE Spectrum*, 2015. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

[7] ——, "The 2016 top ten programming languages," *IEE Spectrum*, 2016. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[8] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans *et al.*, "The HipHop compiler for PHP," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 575–586.

[9] A. Bratterud, A.-A. Walla, H. Haugerud, P. Engelstad, and K. Begnum, "IncludeOS: A minimal, resource efficient unikernel for cloud services," in *International Conference on Cloud Computing Technology and Science (CloudCom'15)*. IEEE, 2015.

[10] A. Kantee and J. Cormack, "Rump Kernels No OS? No Problem!" *USENIX ;login: magazine*, 2014.

[11] M. Helsley, "LXC: Linux container tools," *IBM devloperWorks Technical Library*, 2009.

[12] D. Strauss, "Containers–not virtual machines–are the future cloud," *The Linux Journal*, vol. 228, pp. 118–123, 2013.

[13] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in *International Conference on Cloud Engineering (IC2E)*. IEEE, 2014, pp. 610–614.

[14] A. Madhavapeddy and D. J. Scott, "Unikernels: the rise of the virtual library operating system," *Communication of the ACM*, vol. 57, no. 1, pp. 61–69, 2014.

[15] D. R. Engler, M. F. Kaashoek *et al.*, "Exokernel: An operating system architecture for application-level resource management," in *Symposium on Operating Systems Principles (SOSP'95)*. ACM, 1995, pp. 251–266.

[16] A. Bratterud and H. Haugerud, "Maximizing hypervisor scalability using minimal virtual machines," in *International Conference on Cloud Computing Technology and Science (CloudCom'13)*, vol. 1. IEEE, 2013, pp. 218–223.

[17] K. Stengel, F. Schmaus, and R. Kapitza, "EsseOS: Haskell-based tailored services for the cloud," in *International Workshop on Adaptive and Reflective Middleware*. ACM, 2013, p. 4.

[18] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of Unikernels," in *Symposium on Networked System Design and Implementation (NSDI'15)*. USENIX, 2015.

[19] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft, "Turning down the LAMP: software specialisation for the cloud," in *Conference on Hot topics in Cloud Computing (HotCloud'10)*, vol. 10. USENIX, 2010, pp. 11–11.

[20] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Symposium on Networked Systems Design and Implementation (NSDI'14)*. USENIX, 2014, pp. 459–473.

[21] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 247–260.

[22] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography." *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[23] A. Kantee, "Flexible operating system internals: The design and implementation of the anykernel and rump kernels," Ph.D. dissertation, Aalto University, 2012.

[24] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The Hiphop Virtual Machine," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 777–790.

[25] J. Aycock, "A brief history of just-in-time," *Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[26] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16.

[27] N. Fernando, S. W. Loke, and J. W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Comp. Syst.*, vol. 29, pp. 84–106, 2013.

[28] A. Bahtovski and M. Gusev, "Cloudlet challenges," *Procedia Engineering*, vol. 69, pp. 704–711, 2014.

[29] A. Sathiaseelan, A. Lertsinsrubtavee, P. Baskaran, J. Crowcroft *et al.*, "Cloudrone: Micro clouds in the sky," *arXiv preprint arXiv:1604.08243*, 2016.

[30] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing Magazine*, no. 3, pp. 81–84, 2014.

[31] B. Des Ligneris, "Virtualization of Linux based computers: the Linux-VServer project," in *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*. IEEE, 2005, pp. 340–346.

AVAILABILITY

https://github.com/tfjmp/php2uni.

NOTES

[1] https://github.com/rumpkernel/rumprun-packages

[2] We fully acknowledge the prescience of Solaris Zones [30], the Linux VServer kernels [31] and other technologies in terms of OS-level isolation, but their impact has not come anywhere near that of the recent Linux container developments.

[3] https://www.docker.com/

[4] https://github.com/Unikernel-Systems/DockerConEU2015-demo

[5] https://github.com/emc-advanced-dev/unik

[6] https://galois.com/project/halvm/

[7] https://www.dartlang.org/

[8] http://coffeescript.org/

[9] http://flowtype.org/

[10] http://www.mibsoftware.com/php2cpp/

[11] https://bitnami.com/stack/lamp/virtual-machine

[12] https://httpd.apache.org/

[13] https://hub.docker.com/_/php/

[14] https://github.com/rumpkernel/rumprun-packages/tree/master/php5

[15] https://github.com/facebook/hhvm

[16] Although we acknowledge improvement on that front, we do not believe that a level of isolation similar to a hypervisor can be achieved.

[17] https://phpdoc.org/

[18] https://github.com/emc-advanced-dev/unik

[19] https://github.com/dokku/dokku—an extremely simple PaaS platform built with some of Heroku's open source blocks.