INSA Lyon - Département Informatique

Report

of the

Graduation Project

Evaluating unikernels for HPC applications

Evaluation des unikernels pour les applications HPC

Pierre JACQUOT

Defended July 1st, 2021

Project conducted from **February 8, 2021** to **July 2, 2021**

within the reception structure

Avalon Team, Inria (Lyon)

Referent	:	Florent Dupont De Dinechin, Maître de Conférences	INSA Lyon
Tutor	:	Christian Pérez, Chercheur	Avalon Team, Inria
Tutor	:	Pierre Olivier, Chercheur	Manchester University

Contents

1 Introduction

The last decade saw the birth of unikernels, a new field of research in the system research community [2, 6, 7, 8, 10, 11, 12, 14]. Unikernels are lightweight single application operating systems developed for the cloud, edge computing, Internet of Things, etc. They fit into small images, have low memory footprint, and boot in less than one second. They are known for accelerating the execution of program and improving throughput of network applications. This technology have proven its qualities for theses domains with numerous unikernels and publications [2, 7, 9, 10, 11, 14].

Unikernels are usually compared to containers because they are designed to function in a similar way. Indeed, unikernels are a form of lightweight virtualisation. However, unikernels have more performance benefits than containers. They also reduce the attack surface by providing only the part of the kernels that are required by the executed application.

In the context of the PRACE 6IP phase regarding the deployment of container utilities on High Performance Computing (abbreviated HPC in this report) infrastructures, the question of the suitability of unikernels for HPC has been asked.

Because unikernels increase the throughput of applications, they may improve HPC applications' performances. Reducing OS-noise could also improve the synchronisation between threads inside a single node parallel application, and between nodes in MPI application, leading to an acceleration of their execution. The goal of this study is to start a reflection about the benefits unikernels could provide, if they were deployed alongside containers in HPC infrastructure.

Section 2 introduces the paradigm of Unikernels, and covers the state of the art. Several unikernels are described and evaluated according to their ability to run parallel applications, explaining why we selected HermitCore and HermiTux for our study. Section 3 details the installation and the use of HermitCore and HermiTux with the uHyve hypervisor. Section 4 describes the experimental setup in which we conducted and the benchmarks used to evaluate unikernels. In Section 5, we evaluate HermitCore and HermiTux stability and show theses unikernels have stability issues with OpenMP applications. A performance evaluation is conducted in Section 6, showing that HermitCore and HermiTux greatly reduce the system calls overhead, but does not accelerate compute intensive applications. Section 7 finally concludes the study.

2 State of the art of unikernels

2.1 General principles

Unikernels are a new form of virtualisation that has grown during the years following the release of MirageOS [12, 13], in 2013. They are specific kernels designed for single application execution (see figure 1). By running only one application, theses kernels can be specialised and optimised in many ways [2, 6, 7, 9, 10, 11, 12, 14]. Unikernels are known for their lightweightness: they consume very few memory, and can fit in tiny images. They reduce the attack surface by removing any unnecessary part of the kernel and let the hypervisor enforce the isolation with other virtual machine. Finally, they provide fast system calls, and experience reduced OS noise.

Unikernels paradigm is commonly described as "OS as a library", or LibOS. This principle is an application of the exokernel [5] model to the cloud. In unikernel virtualisation, the hypervisor plays the role of the Exokernel, and the unikernel is the LibOS. Their goal is to provide only the functionalities required by the application to run. Each unikernel tries to reduce the size of the kernel by removing unneeded modules when they are not used by the application: Network stack, file system, thread scheduling... The following paragraphs details some aspects of this specialisation.

Unikernel speciliazation

Because there is only one application running, unikernels generally execute it in kernel-space. This single-space execution accelerates the systems calls performed by the application. System calls are known to be a slow operation because of their mode-switching procedure. Indeed, before performing the system call, the program is being executed in user-mode. When the system call instruction is executed, the user-mode execution is stopped, and then resumed in kernel-mode. Once the kernel has completed the system call, it returns its results and switch back to user-mode. In common processors, this mode-switching requires flushing the CPU pipeline, saving standardised registers (depending on the system call performed) onto the stack, changing protection domain... This process of switching back and forth from user-mode and kernel require many CPU cycles [15]. Also, since the discovery of the Meltdown vulnerability in processors older than 2018, a new feature has been added in regular kernels that is slowing down system calls even more. The KPTI feature involves a page table switch each time there is a mode-switch, implying a costly TLB flush. In unikernels, the application and the kernel are tied together. The application is executed in kernel-mode, and by doing so the mode-switching of system calls is not required anymore. This means that system calls are replaced by common function calls, saving many CPU cycles. Unikernels have their own way of transforming system calls. For example, OSv and HermitCore use a custom libC that calls directly the kernel functions instead of performing system calls [11, 7], while HermiTux rewrite the system call instruction into a function call for statically compiled binaries [14].

Finally, because the running application have all the virtual machine for itself, there is no need to provide memory isolation. Unikernels assume that this isolation is provided by the hypervisor, which is isolating the different virtual machines from each other. Unikernels are single-address space operating systems, which can accelerate memory allocation mechanisms.

Unikernel compatibility

Compatibility is an important matter with Unikernels. Most unikernels (e.g HermitCore [11], Rumprun [6]) will require the source code of an application to be recompiled in order to execute it. By recompiling the application, the unikernel's features are linked directly to the target application. The generated binary executable contains both the application and the unikernel. In this case, the compatibility between application and unikernels is handled at compile-time. If a feature is not supported by an unikernel, the compilation of the application will be impossible, and will trigger an error. In this case, the developer will need to port its application for the targeted unikernel so that it will compile with the unikernel. Another way of handling compatibility between applications

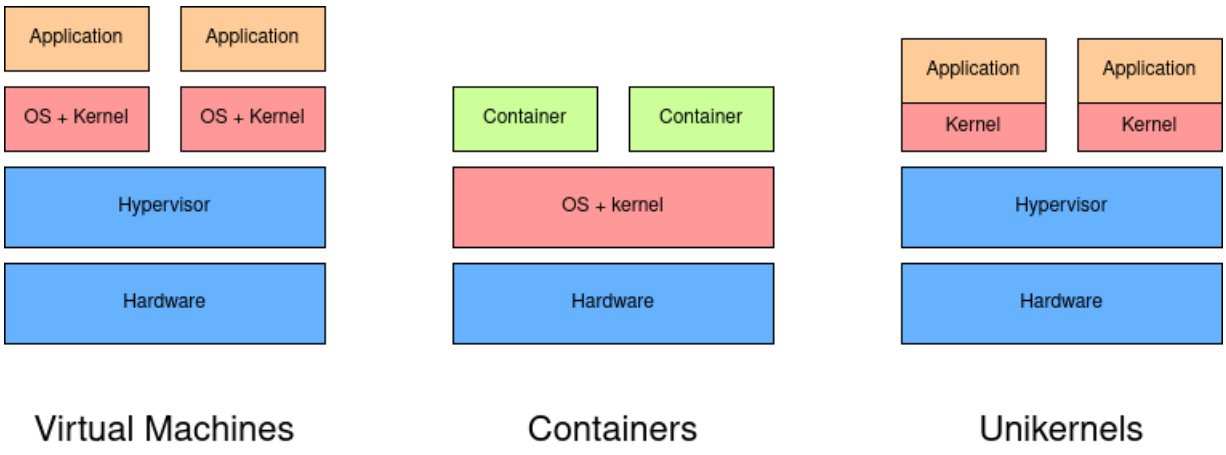

Figure 1: Unikernels compared to virtual machines and containers

and unikernels is by providing binary compatibility. Some unikernels (e.g. HermiTux [14], OSv [7]) are able to execute Linux executable binaries without recompilation. In this case the compatibility is handled by the unikernel, which need to support the features called by the binary executable. If a feature is not supported, the application will crash at execution-time. This time it is the responsability of the unikernel's developer to implement the missing feature in order to get the executable running. If the unikernel maintainer cannot implement the feature for some reason, there is still the possibility to avoid using the feature if possible.

2.2 Unikernel categories

Unikernels can be classified in two categories: Language-based unikernels, and POSIX-Like unikernels.

Language based unikernels are tied to one specific programming language. They only support applications written in this language, and require the application to be written specifically for their own API. The concept of "OS as a library" is very well adapted for theses kernels, because they really are a set of libraries that need to be included in the application source code. Although they provide a very good optimisation and specialisation of the kernel, the major downside of theses unikernels is that they require more development effort to run an application. Language based unikernels are for example MirageOS [8] an OCamL based unikernel, and IncludeOS [2] a C++ based unikernel.

Contrary to language-based unikernels and their specific API, POSIX-like unikernels try to provide the most complete implementation of the POSIX API they can. Theses unikernels offer some compatibility at sources level for Linux applications. However, because their implementation of the POSIX API is often incomplete, they sometimes require porting efforts to execute a given application. In the best cases, only the recompilation of the application sources is required, while in the general case a few porting efforts are required to make it run. Examples of POSIX-like unikernels are: HermitCore [11], HermiTux [14], Rumprun [6], OSv [7], Lupine Linux [10], and Unikraft [9].

2.3 Unikernel analysis in the context of HPC

This section presents our analysis of several unikernels with respect to their support for HPC applications.

By HPC applications, we restrict this study to applications meeting the following criteria:

- They are compatible with C, C++ and FORTRAN applications.
- They can run on multi-core architecture.
- They are compatible with OpenMP applications.

The support of languages, multi-core, and OpenMP can be verified by compiling and executing a basic OpenMP application that spawns several threads that iterate over a loop. If the compilation succeeds, the process monitor `htop` can be used to verify that the threads are really running simultaneously on distinct cores of the machine. If the application can be compiled and can complete its execution, we consider that the unikernel meets our HPC criteria.

Specific hardware support

Another criteria that has been considered is the support of existing drivers for specific HPC hardware. Some HPC application are able to take advantage of dedicated architectures, such as high network bandwidth or GPGPUs. The support of theses particular architectures is done by specific drivers, that are often proprietary drivers.

However, the field of drivers support has not been explored by many unikernels. Since they were originally designed for cloud and embedded contexts, the mechanisms for driver support have not been implemented. In the unikernels we considered for this report, only Rumprun [6] and Lupine Linux [10] may be compatible with Infiniband and GPGPU drivers. This intuition is detailled in their relative description in subsubsections 2.3.2 and 2.3.6. For other unikernels, the drivers would have to be reimplemented from scratch directly in the kernel. This is something we could not afford for this study.

In summary, we restricted our study to POSIX-like unikernels, to avoid rewritting applications for distinct APIs. By proceeding this way, we greatly reduced the amount of work needed to run the benchmarks presented in this paper with unikernels: we only had to make a few modifications to port them for the selected unikernels. The following subsections present OSv, Rumprun, Unikraft, HermitCore, Hermitux and Lupine Linux unikernels.

2.3.1 OSv

OSv is an unikernel that was originally developed by Cloudius Systems, interfacing with the application at the C Library level [7]. It uses a custom C library based on the musl C library. This custom libC is designed to avoid making system calls by directly calling OSv's features. It aims at efficiency by implementing lock-free algorithms, per-CPUs waiting queues for threads, tickless thread scheduling and a light-weight network stack. OSv provides compatibility with the Java Virtual Machine, which makes it compatible

with Java applications. The project is now open-source, and some volunteers still commit changes to the OSv repository. OSv support C, C++ and FORTRAN applications however, it does not support OpenMP applications. So we chose to put it aside.

2.3.2 Rumprun

Rumprun is an unikernel developed by the FreeBSD foundation, but not maintained anymore. It is based on the NetBSD Rump anykernel. It is a set of drivers and system call handlers [6]. Rumprun can run on any platform able to execute C99 code, with only a hundred kilobytes of RAM/ROM. This unikernel can run on bare metal, or above an hypervisor. Because Rump kernels were originally designed for developing drivers, we believe that this kernel would be a good candidate for experimenting with HPC drivers. However, despite its low level architecture compatibility, Rumprun is not compatible with SMP execution. To do so, it requires a "multi-kernel" approach: Spawning a unikernel on each core, and making them communicate through IP-protocol. Also, Rumprun only support C/C++ applications. Theses points do not meet our criteria, so we eliminate Rumprun from our list.

2.3.3 Unikraft

Unikraft [8] is an unikernel developed by NEC laboratories. It is designed to be fully modular and customisable [9], and very efficient due to performance-minded and well designed APIs. It tries to improve application performances in two ways. The first method is the gain offered by the unikernel paradigm. It reduces the overhead of systems calls, the memory footprint of the kernel and can accelerate memory allocation by choosing the right allocator for an application. The second way of performance improvement can be achieve by adapting the application to take advantage of Unikraft's lower APIs, where performance is critical. Unikraft is supported by a strong developers community, so it is a great candidate for our study. However, it does not support multi-core yet, as well as FORTRAN applications. So it is not considered in our study, but it will surely be worth looking at in a few years.

2.3.4 HermitCore

HermitCore is an unikernel developed at RWTH Aachen University (Germany), and designed for extreme-scale computing. It is designed with performance in mind, and aims at reducing overhead caused by the operating system [11]. It uses the Newlib C library, a library designed for embedded systems, requiring only a few system calls from the OS. It uses a dynamic timer to avoid most of the interruptions of the running application and its threads. It supports multi-core applications and OpenMP C/C++ and FORTRAN applications, so we use it in our study of unikernels. It is important to note that even if HermitCore is compatible with OpenMP application (thanks to an Intel OpenMP runtime shipped with the unikernel), it is not compatible with every OpenMP runtime. In section 3.1.2 we show that controling the compiler and the OpenMP runtime used with HermitCore unikernel is not an easy task. In this paper, we consider the "original" HermitCore unikernel, written in C language. It is not actively maintained anymore.

2.3.5 Hermitux

Hermitux is an unikernel based on HermitCore, developed at Virginia Tech (USA) and the university of Manchester (UK). It has been designed to be binary compatible with linux executables [14]. It does not need to recompile Linux application before executing them. By doing so, it reduces the efforts required to port applications. Hermitux reduce the system calls overhead thanks to an optimised system call handler. Even if the unikernel is not maintained as often as it was before, it keeps getting support from its main developer. Hermitux supports OpenMP applications and supports multi-core. Because of its binary compatibility with Linux binaries, it is compatible with C/C++ and FORTRAN applications. This binary compatibility is more permissive for controlling compilation and linking of OpenMP runtime, as we show in section 3.2.2. Hence, we use it in our study of unikernels.

2.3.6 Lupine Linux

Lupine Linux [10] is a configuration of the Linux kernel developed by the University of Illinois and IBM research. It specialises the Linux kernel for the execution of a single application, and removes what is unneeded in the original kernel for a given application. Lupine Linux is not actively maintained anymore. Because it relies on the Linux kernel, we expect a very good compatibility for applications and drivers. Unfortunately because the configuration of the kernel can be time-consuming, we did not had enough time to complete our analysis of Lupine Linux and studied if we were able to run multi-core OpenMP applications with it.

2.4 Conclusion

What shows the analysis of unikernels is that the support of multi-core and OpenMP is not a very common feature in unikernels. Unikernels have been designed to be suited for cloud ecosystem rather than the HPC context. Because of their lightweight and quick boot time, unikernels tends to be spawned multiple times on different cores rather than have an unique instance using several cores. We conclude by presenting a table summarising the evaluation of unikernels with respect to our criteria in Figure 2.

unikernel	C/C++/FORTRAN support	multi-core support	OpenMP support
OSv	yes	yes	no
Rumprun	no	no	yes
Unikraft	no	no	no
HermitCore	yes	yes	yes
Hermitux	yes	yes	yes
Lupine Linux	yes	?	?

Figure 2: Hermitux and HermitCore are the only unikernels that meet our criteria.

3 Installation and usage of HermitCore and Hermitux unikernels

This section illustrates the use of HermitCore and Hermitux on a Linux Debian 10 system. For each unikernel, we describe its installation process, how to compile an application, and finally how to execute it. Except for installing packages that requires `root` privileges, every command could be performed as regular user on the system.

3.1 HermitCore

3.1.1 Installation

Before being installed, HermitCore requires several packages used in classic development project. They can be installed with the commands provided in Listing 1.

Listing 1: Required packages installation for HermitCore

```
sudo apt-get update
sudo apt-get install git build-essential cmake nasm apt-transport-https \
     wget libgmp-dev bsdmainutils libseccomp-dev python libelf-dev
```

The installation of HermitCore is very simple. There are debian packages available for this unikernel, which avoid the trouble of compiling it. HermitCore can be installed with the commands in listing 2. Theses commands install HermitCore in the `/opt/hermit` directory. In the following parts of this section, we assume that HermitCore is located in this directory.

Listing 2: Installation of HermitCore

```
for dep in binutils-hermit_2.30.51-1_amd64.deb gcc-hermit_6.3.0-1_amd64.deb \
        libhermit_0.2.10_all.deb  newlib-hermit_2.4.0-1_amd64.deb; do \
    wget https://github.com/ssrg-vt/hermitux/releases/download/v1.0/$dep && \
    sudo dpkg -i $dep && \
    rm $dep;
done
```

3.1.2 Applications compilation

Compiling applications for HermitCore is not trivial. Because the unikernel and the executable are merged into a single binary, it is mandatory to link HermitCore's kernel with the target application during the linking operation. To do so, HermitCore provide its own toolchain in the `/opt/hermit/bin` directory. It contains compilers for C/C++ applications (`x86_64-hermit-gcc` and `x86_64-hermit-g\++`, but also binary utilities such as `x86_64-hermit-objdump` (an equivalent of `objdump`, `x86_64-hermit-c++filt` (an equivalent of `c++fil1t`), and `x86_64-hermit-elfedit` (an equivalent of `elfedit`).

Once the toolchain has been discovered, the build process for a given application is rather simple. A simple C application can be compiled with the command provided in Listing 3. OpenMP applications can be compiled with the command provided in Listing 4.

Listing 3: Compilation command used for compiling a simple hello world application. This application simply print "Hello world" on the standard output of the program.

```
/opt/hermit/bin/x86_64-hermit-gcc -o hello hello.c
```

Listing 4: Compilation command used for compiling an OpenMP basic application. This application simply spawns threads, that will print a few lines on the program standard output.

```
/opt/hermit/bin/x86_64-hermit-gcc -o omp-test -fopenmp omp-test.c
```

The command on Listing 4 lets HermitCore's toolchain fetch its own OpenMP runtime. Indeed to compile and execute OpenMP applications, HermitCore is shipped with a pre-compiled Intel OpenMP runtime [11] so that their compilation is simple. However, for evaluation purpose, one may want to control the OpenMP runtime used for executing the application.

For evaluation purposes, we tried to link other OpenMP runtimes with HermitCore. We wanted first to link the LLVM OpenMP runtime (version 11) to our target applications. Our goal was to execute each applications with the same OpenMP runtime regardless of what unikernel is used to execute it. We also tried to link the GCC OpenMP runtime to target application, to compare differents runtimes performances for a given unikernel. Sadly, we did not manage to successfully link theses two runtimes to HermitCore applications. Either the compilation was aborted due to linking errors, either the compiled application crashed at execution.

Another point one might wish to control during the compilation process is the compiler. Indeed, being able to choose which compiler to use means controlling the generation of the binary executable. However, because the linking part of HermitCore application is not trivial, due to the linking of the application code with HermitCore's kernel, using a different toolchain for linking is difficult.

A workaround to this difficulty could be to compile the application sources into object files with a compiler that is not from the HermitCore's toolchain, and let the HermitCore toolchain handle the linking operation. This can be done in three steps:

- Compile the application sources into object files with the compiler of your choice.
- Use `x86_64-hermit-elfedit` tool to convert the objects files into HermitCore's binary format.
- Link the converted object files and HermitCore's kernel with HermitCore's toolchain.

The commands corresponding to this 3-steps build are shown in Listing 5. By converting the compiled object files (here compiled with the clang compiler) to the standalone format, they can be assembled into a HermitCore binary by the HermitCore toolchain.

Listing 5: Conversion to HermitCore format

```
clang -c omp-test.c
/opt/hermit/bin/x86_64-hermit-elfedit --output-osabi Standalone omp-test.o
/opt/hermit/bin/x86_64-hermit-hermit-gcc -o omp-test -fopenmp omp-test.o
```

This trick for compilation has been used to try to compile a new OpenMP runtime for HermitCore. What we tried was to compile the LLVM OpenMP runtime version

11 with the clang compiler. We gathered the resulting object files of the compilation and converted them to HermitCore's binary format. Then, we tried to link directly the object files of the OpenMP runtime and our application with the HermitCore toolchain. Unfortunately, this method triggers linking errors, and does not generate any executable.

3.1.3 Executing HermitCore applications with uHyve

Once the application is compiled, the only thing left is to launch the unikernel. HermitCore supports QEMU and KVM hypervisors. However because theses hypervisors tend to have a boot sequence that can be quite time consuming, and an important memory footprint, a light-weight hypervisor named uHyve has been developed in conjunction with HermitCore. UHyve allows the unikernel to boot in less than a second, and has a light memory footprint compared to the classic hypervisors cited above.

Located in HermitCore utilities, the `proxy` loads HermitCore and its application, and starts the virtualisation. This tools use several environments variables to size the virtual machine. Figure 3 lists the environment variables we used for configuring the creation of the virtual machine. Listing 6 shows an example of command syntax that can be used for executing an OpenMP application on 8 cores.

- `HERMIT_ISLE`: Specify hypervisor. Its value can be `uhyve`, `qemu` or `kvm`.
- `HERMIT_CPUS`: Specify the number of cores the unikernel the unikernel have access to. This variable must be defined in conjunction with the variable `OMP_NUM_THREADS` for OpenMP multi-core applications.
- `HERMIT_MEM` : defines the amount of memory allocated to the unikernel. Memory is specified by a number and its unit (e.g. `4G` for allocating 4 gigabytes).
- `HERMIT_VERBOSE`: setting this variable to `1` will cause the unikernel to print its kernel log at the end of the execution. By default, this variable is set to `0`.

Figure 3: A non exhaustive list of the environment variables used by the `proxy` tools of HermitCore to create the unikernel.

Listing 6: HermitCore execution

```
HERMIT_ISLE=uhyve HERMIT_CPUS=8 OMP_NUM_THREADS=8 HERMIT_MEM=4G \
/opt/hermit/bin/proxy omp-test
```

3.2 HermiTux

3.2.1 Installation

HermiTtux is an unikernel based on HermitCore. It requires HermitCore to be installed on the machine before being build, as described in Figure 1 and Figure 2 for installing HermitCore. For building Hermitux, the hermit compiler `x86_64-hermit-gcc` is required. It is installed with the toolchain of HermitCore.

The installation of Hermitux is rather simple. It is only a matter of cloning and building of repository. This process compiles the kernel sources, and some other components. The commands used for building Hermitux are given by Listing 7. The files generated are all located inside the Hermitux repository: `hermitux-kernel` for Hermitux' kernel, `libiomp` for Hermitux' LLVMP OpenMP runtime, and `musl` for Hermitux' wrapper for the compiler.

Listing 7: HermiTux installation

```
export CC=/opt/hermit/x86_64-hermit-gcc
git clone https://github.com/ssrg-vt/hermitux
cd hermitux
git submodule init \&\& git submodule update
make
```

Once Hermitux is compiled, an additionnal step can be to edit the variable `HERMITUX_BASE` in the files `Makefile.template` and `Makefile.template.omp` inside the tools directory. This variable is used to locate Hermitux in the filesystem. Theses Makefiles are generic makefiles used for building applications that are in the `apps` folder of the repository. Changing the value of this variable will enable to compile the demo applications of the `apps` folder.

3.2.2 Compiling applications

Hermitux is a binary compatible unikernel. This unikernel has been designed to avoid the recompilation phase of a binary executable compiled for a linux system. The merging of the target application and the Hermitux kernel is done by the Hermitux tools just before executing the virtual machine. This really simplifies the compilation process of an application. It is not the developers that are supposed to port their application for Hermitux, but Hermitux that is supposed to support the functionalities requested by the application[14]. Hermitux still provides a wrapper for the compiler used for its build: the `musl-gcc` compiler.

Hermitux is compatible with OpenMP applications. However, we observed that Hermitux can suffer from compatibility with some OpenMP runtimes. Hermitux is indeed shipped with a static library of the LLVM runtime of OpenMP. Compiling an application with this runtime will produce a binary that can be executed by Hermitux. The compilation command used to build an executable OpenMP application can be found in Listing 8. Trying to execute an application that been linked with another runtime of OpenMP will result into a crash at execution.

Listing 8: OpenMP application compilation

```
hermitux/musl/obj/musl-gcc -static -o omp-test -fopenmp \
    -Lhermitux/libiomp/build/runtime/src omp-test.c
```

We tried several methods to link a different OpenMP runtime with the application. First, we wanted to link our application with the LLVM OpenMP runtime version 11 compiled with Clang compiler. Indeed, if we know that Hermitux' OpenMP runtime is the one made by LLVM, we do not know what version it is and which compiler was used for compiling it. We cloned the LLVM project repository, and cloned the 11 version of OpenMP with Clang. We tried three methods for linking our self compiled runtime:

- Dynamically link the application with the runtime's library file (shared object).
- Create a static library from the object files of the runtime, and link both the application and the static library into a static executable.
- Link the compiled object files of the runtime directly to the application, into a static binary.

Only the last method provides an executable that can be executed by HermiTux without crashing. The fact that linking a static library of our runtime does not work while the linking of the object files is surprising. We believe that in the OpenMP runtime that we compiled, something is missing to make it compatible with HermiTux kernel. The fact that the third method produces a functional executable comes from the fact that we use HermiTux's wrapper (as shown in Listing 9) for linking all the object files together. Inside the `runtime` folder are located the object files resulting from the compilation with Clang compiler of the LLVM runtime. If we managed find a way for using another version of the LLVM runtime, we did not manage to find a way for using the GCC runtime.

Listing 9: Compilation with a LLVM OpenMP runtime version 11

```
clang -c -fopenmp omp-test.c
hermitux/musl/obj/musl-gcc -static -o omp-test -fopenmp omp-test.o \
    omp-runtime/*.o
```

3.2.3 Executing applications with uHyve

Hermitux' execution can be hypervised in a very similar way to HermitCore's although there are a slight difference. First, a special environment variable must be defined to use uHyve's proxy: `HERMIT_TUX=1`. The other variables shown in Figure 3 can be used. Second, to ensure binary compatibility with Linux binaries, Hermitux' kernel is separated from the application binary. The two binaries are merged just before the start of the unikernel. From this results two cases of execution:

- Execution of dynamically linked application, which requires to provide a loader to load the shared objects used by the binary (see Listing 10).
- Execution of statically linked applications, which does not require a loader (see listing 11).

Listing 10: Execution of a dynamic executable

```
HERMIT_ISLE=uhyve HERMIT_TUX=1 \
hermitux/hermitux-kernel/prefix/bin/proxy \
/lib64/ld-linux-x86-64.so.2 \
hermitux/hermitux-kernel/prefix/x86_64-hermit/extra/tests/hermitux \
application
```

Listing 11: Execution of a static executable

```
HERMIT_ISLE=uhyve HERMIT_TUX=1 \
hermitux/hermitux-kernel/prefix/bin/proxy \
hermitux/hermitux-kernel/prefix/x86_64-hermit/extra/tests/hermitux \
application
```

4 Experimental setup

This section details the experimental setup used for the stability evaluation (Section 5) and the performance evaluation (Section 6).

4.1 Hardware description

The evaluations have been performed on machines of the Grid'5000 infrastructure. The information given here are extracted from Grid'5000 documentation [1]. We used the following nodes.

- `nova` is a machine from the Lyon site of Grid'5000. The specifications of its nodes are:

Model	Dell PowerEdge R430
CPU	Intel Xeon E5-2620 v4 (Broadwell, 2.10GHz, 2 CPUs/node, 8 cores/CPU)
Memory	64 GiB

- `gros` is a machine from the Nancy site of Grid'5000. The specifications of its nodes are:

Model	Dell PowerEdge R640
CPU	Intel Xeon Gold 5220 (Cascade Lake-SP, 2.20GHz, 1 CPU/node, 18 cores/CPU)
Memory	96 GiB

On each node, we take care of disabling Hyperthreading before making any experiments.

4.2 Unikernels used

We use HermitCore and HermiTux[1] from unikernels for our studies. Experiments are also performed on a Debian 10 distribution (linux kernel version `4.19.0-14-amd64`), to have a reference.

Due to unikernels limitation regarding OpenMP and compilers, we have been force to compile specific executables for linux and each unikernels:

- Linux executables are compiled with Clang compiler version 11, and use the LLVM OpenMP runtime version 11 (compiled with Clang v11).
- HermitCore executables are compiled with HermitCore's toolchain, and HermitCore's OpenMP Intel runtime. The version of this runtime is unknown to us.
- HermiTux executables are compiled with Clang compiler version 11, and linked with the LLVM OpenMP runtime version 11 (compiled with Clang v11) thanks to HermiTux' GCC wrapper.

[1]Commit identifier of HermiTux version used: `d92b5bd45a34f595c56a5737db4815f5f3a8e790` from `https://github.com/ssrg-vt/hermitux`

It is important to note that we do not control the compiler used to compile HermitCore applications and which OpenMP runtime is used with HermitCore unikernel. As explained in the section about Technical details about unikernels, we did not manage to get the LLVM OpenMP runtime working with HermitCore. The results regarding HermitCore presented in the following section must be interpreted with caution.

4.3 Benchmarks used

4.3.1 Bots benchmarks

The Bots benchmarks [4] are a a set of benchmarks developed by the Barcelona Supercomputing Center. They are used for evaluating various OpenMP tasking implementations for given problems. Here is a list of the benchmarks we used, extracted from the documentation of the Bots benchmarks:

- Alignment: Aligns sequences of proteins.
- FFT: Computes a Fast Fourier Transformation
- Floorplan: Computes the optimal placement of cells in a floorplan.
- Health: Simulates a country health system.
- NQueens: Finds solutions of the N Queens problem.
- Sort: Uses a mixture of sorting algorithms to sort a vector.
- SparseLU: Computes the LU factorization of a sparse matrix.
- Strassen: Computes a matrix multiply with Strassen's method.

In the table figure 4, we check the boxes corresponding to a problem solving and its implementations we have at our disposal. The differents implementations alter the tasks generation of the benchmarks:

- An `-tied` implementation means that the tasks generation is limited. If this suffix is not present, it means that the tasks generation is unlimited.
- An `-if_clause` implementations mean that new tasks are generated when a condition is fulfilled. This condition is verified by an OpenMP directive.
- A `-manual` implementation means that new tasks are generated when a condition is fulfilled, but this time, the condition is not checked by the OpenMP directive. It is done by a "manual" `if` statement in the sources.
- `for-` implementations generate tasks with a `omp for` directive. In this case, there can be multiple tasks generators.
- `single-` implementations means that there will be only a single tasks generator.

implementations	alignment	fft	fib	floorplan	health	nqueens	sort	sparselu	strassen
omp-tasks		x	x	x	x	x	x		x
omp-tasks -tied		x	x	x	x	x	x		x
omp-tasks -if_clause			x	x	x	x			x
omp-tasks -if_clause-tied			x	x	x	x			x
omp-tasks -manual			x	x	x	x			x
omp-tasks -manual-tied			x	x	x	x			x
for-omp -tasks	x							x	
for-omp -tasks-tied	x							x	
single-omp -tasks	x							x	
single-omp -tasks-tied	x							x	

Figure 4: Problems solved by the Bots benchmarks, and their respective implementations.

To compile the Bots benchmarks for HermitCore unikernel, a few modifications are performed on the sources. Because the structure `utsname` is not defined in HermitCore, the program can't compile with `x86_64-hermit-gcc`. This structure is returned by the `uname()` system call to describe the kernel name, release and version. Every use of this structure in the benchmarks is removed. It is not a critical functionality of the benchmarks so this deletion do not have any negative impact on the benchmarks. We also remove a call to the `basename()` function that was not supported by HermitCore. Again, it is not a critical feature of the benchmarks. Finally, a symbolic link has been created from `/opt/hermit/lib/gcc/x86_64-hermit/6.3.0/include/memory.h` pointing to `/opt/hermit/x86_64-hermit/include/hermit/memory.h`. Theses modifications make the Bots benchmarks compatible with HermitCore.

4.3.2 Rodinias benchmarks

The Rodinias benchmarks [3] are designed to evaluate different accelerators for compute intensive applications: OpenMP, OpenCL, and CUDA. They are composed of already existing benchmarks, that have their unique behaviour, and come from many domains: Medical Imaging (`Leukocyte`, `Heartwall`...), Bioinformatics (`MUMmerGPU`), Fluid Dynamics (`CFD Solver`), Linear Algebra (for example `LUDecomposition`)... Among the many benchmarks composing the Rodinias benchmarks, only a subset is interesting for our study. In order to evaluate unikernels performance for HPC applications, we are interested in applications with long and intensive computation phases. We used the Rodinias benchmarks only with the OpenMP accelerator, to study unikernels behaviour on multicore CPUs.

When we wrote this report, we did not finished the experiences with all of the benchmarks we selected due to compatibility problems, and a lack of time. We observed a bug in HermiTux that caused some execution times to be negative. A few debugging showed that the bug occured when using the `gettimeofday()` system call. This bug has been fixed. Sadly, two benchmarks still cause troubles. The `bfs` benchmarks does not spawns threads correctly. The problem is surely coming from the benchmarks, because the benchmarks spawn always the maximum threads it can for an execution on debian 10, and only spawn one threads when executed with unikernels. There might be a bug located inside the benchmark source code. The `Kmeans` benchmark source code contains a function definition that conflicts with another located inside HermitCore's kernel, causing errors at compile time. Because the Rodinias benchmarks do not share the common operations (such as time measuring, initialising etc.), making modification in theses benchmarks is longer than in the Bots benchmarks. Due to a lack of time, we had to prioritise other topics, to presents results in this report. Finally, the benchmarks we managed to get working for both HermitCore and HermiTux are the following:

- `lud` (LU Decomposition) is a benchmark coming from the field of Linear Algebra. It is a benchmarks decomposing a matrix as a product of matrices.
- `LavaMD` (LavaMD2) is a benchmark coming from the filed of Molecular Dynamics. It calculates position of particles in a 3D space considering the forces that apply between the particles.

5 Stability Evaluation

5.1 Overview

This section present the stability evaluation of HermitCore and HermiTux unikernels on the Bots benchmarks. During our first attempts at executing OpenMP application with unikernels, we noticed frequent and random crashes, as well as applications never completing. In this section we try to quantify this phenomenon.

5.2 Metrics

The following metrics are considered.

For a given program, we consider its number of crashes experienced for a given number of executions. By "crash", we mean that the execution of the program did not complete, and was aborted due to an error (segfault, pagefault...).

For a given program, we also consider its number of deadlocks experienced for a given number of executions. By "deadlock", we means that for some reasons, the program got stuck during its execution, and could not complete its execution before a certain time limit. This time limit is an overestimation of the time that should take an execution of the program, determined empirically before the experience execution. We assume that if the program did not completed before this time limit, it has been stuck into a deadlock.

5.3 Parameters

For this study, we consider three parameters.

The first one is the unikernel used for the stability experience. We consider HermitCore, Hermitux and linux (a Debian 10 distribution) for this experience. The goal of this parameter is to see if some unikernels are more subject to stability issues than other.

The second parameter is the number of cores allocated by the execution. By varying this parameter, we want to observe whether the stability of the selected unikernels is influenced by the number of core allocated to their execution. Executions of parallel programs with high cores number and threads number usually increase the probability of encountering a non-deterministic bug. We run the benchs on 1, 2, 4, 8 and 16 cores, with one thread per core and no hyperthreading. With this parameter we want to see if HermiTux and HermitCore are subject to non-deterministic bugs.

The last parameter is the program. By changing the program used for the stability evaluation, we want to observe if some programs experience more crashes or deadlocks during their executions. Observing this may help to find the reasons why some executions cannot complete successfully. We use the Bots benchmarks for this experience. In theses benchmarks, a single problem (e.g. a Fibonacci computation) is solved by different implementations (e.g. an omp-tasks implementation and omp-tasks-tied implementation).

5.4 Experimental methodology

We start by executing the programs used in this experiments several times, in order to empirically determine an overestimated time limit for their execution. Our goal here is to determine a time limit that the execution time of a program will never cross when the

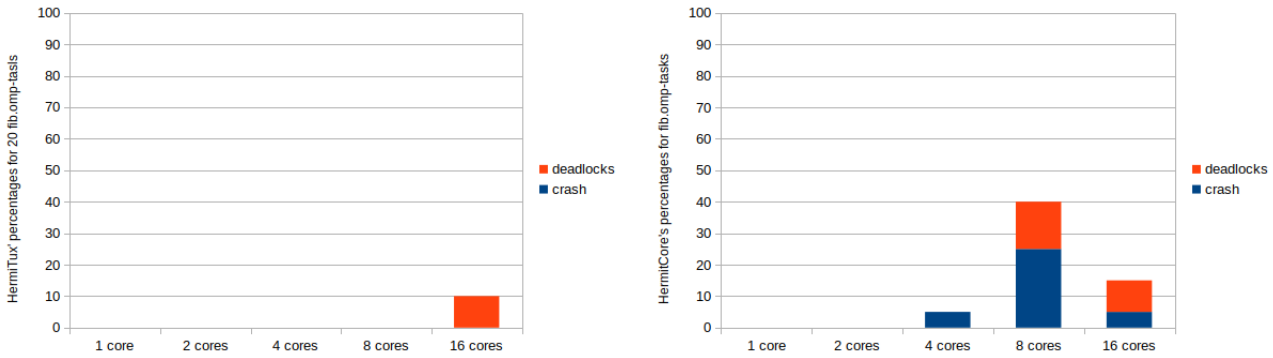

Figure 5: Stability evaluation for fib.omp-tasks executable.

execution complete successfully. The timeout is computed by adding 20 seconds to the mean of the execution times.

Once we determined time limit for each programs, we execute them 20 times each, with linux, and with each unikernel. The program is executed with a `timeout` command, that will kill the process after a given time. The code returned by the execution of this command give us the type of execution we observed:

- If this code is 0, then the execution completed successfully. It is the code returned by the benchs when they complete successfully.
- If this code is 124 or 137, then the execution was stopped by the `timeout` command. This means that the program did not completed in time, and that we observed a deadlock.
- For any other code we assume that an error occurred during the execution of the application. We categorise it as a crash.

5.5 Stability improvement of HermiTux

Preliminary experiments shown a significant instability of HermiTux. After discussing with the HermiTux' main developer, he has been able to fix two major bugs that were located inside HermiTux' kernel. A first issue was a memory allocation related bug, that occurred for a particular use case of the `mmap` system call. The second issue was a thread management bug that caused deadlocks.

As these fixes have been committed to HermiTux, this section evaluates the latest version of HermiTux, i.e., the version with these two bug fixes.

5.6 Experimental results

This section analysis the number of crashes and deadlocks experienced over 20 executions, for the bots benchmarks shown in the table.

It is important to note that the stability analysis has also been performed on a Debian 10 system (e.g. by executing the benchmarks without using a unikernel). Every benchmarks cited below works perfectly with Debian 10, and thus we do not report it in the figures.

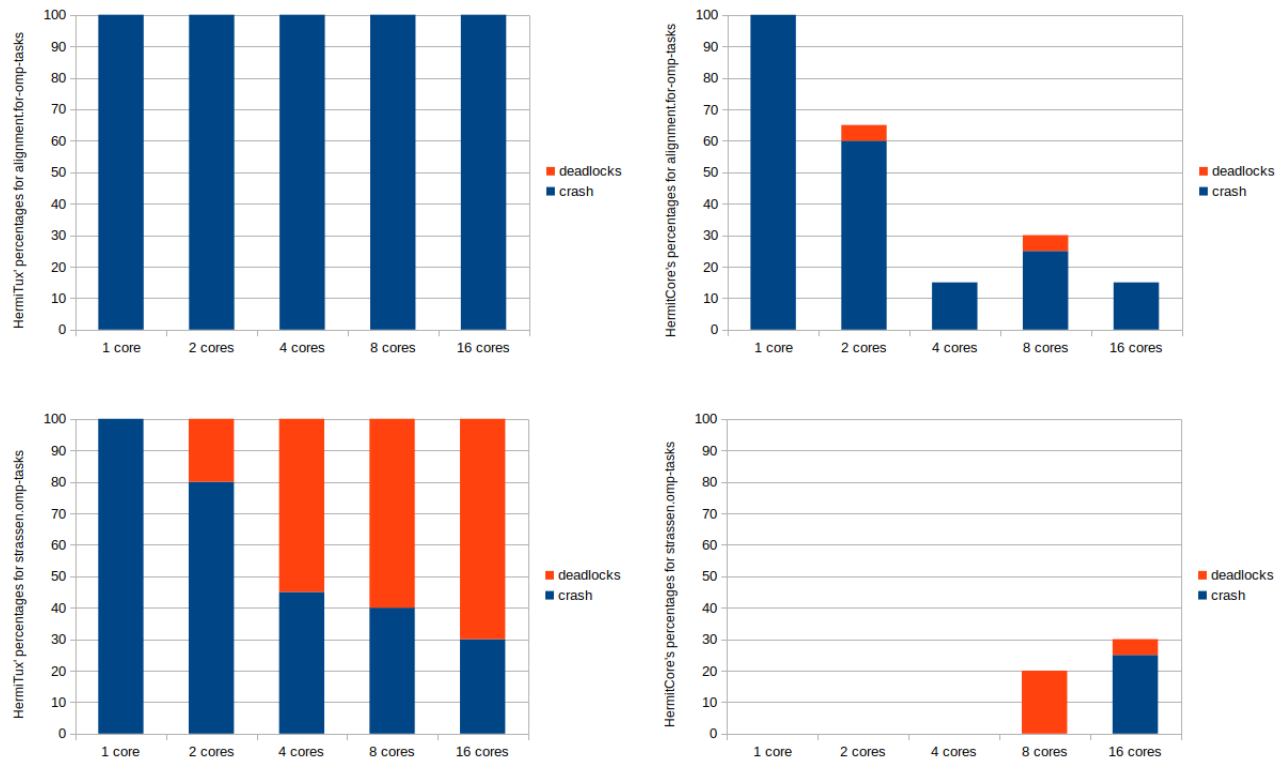

Figure 6: Stability evaluation for alignment.for-omp-tasks and strassen.omp-tasks benchmarks

5.6.1 Stability in function of the number of cores allocated

Figure 5 compare the stability of HermiTux and HermitCore in function of the number of cores for our subset of the Bots benchmarks. Histograms are plotted to show the percentages of crashes or deadlocks for 20 executions in function of the number of cores used. This figure shows that there is one class of benchmarks for which HermiTux is more stable than HermitCore. HermiTux experiences very few crashes and deadlocks at execution for this benchmark. For HermitCore, crashes and deadlocks often start to occur at 4 cores excutions. On 20 executions, we observe important number of crashes and deadlocks for HermitCore, which can lead to 50% of the executions not completing successfully for some benchmarks. Among the benchmarks we used, 29 shows executions where both unikernels are able to execute successfully the application, and where HermiTux experience less crashes and deadlock than HermitCore.

In Figure 6 we observe that HermiTux can have serious compatibility issues with some executables. The two benchmarks showed in this figure never complete successfully when executed via HermiTux. They complete successfully with HermitCore, but with a high proportion of failures and deadlocks. An interesting point to notice for HermiTux is that the alignment benchmark always crashes, and does not get stuck into deadlocks, while the strassen benchmark experience both crashes and deadlocks. Among the 42 executables we used for this study, 11 never completed successfully when they were executed by HermiTux. A similar phenomenon can be observed for HermitCore in Figure 7 which shows a very good stability for HermiTux: only one deadlock occurred, for the 16 cores

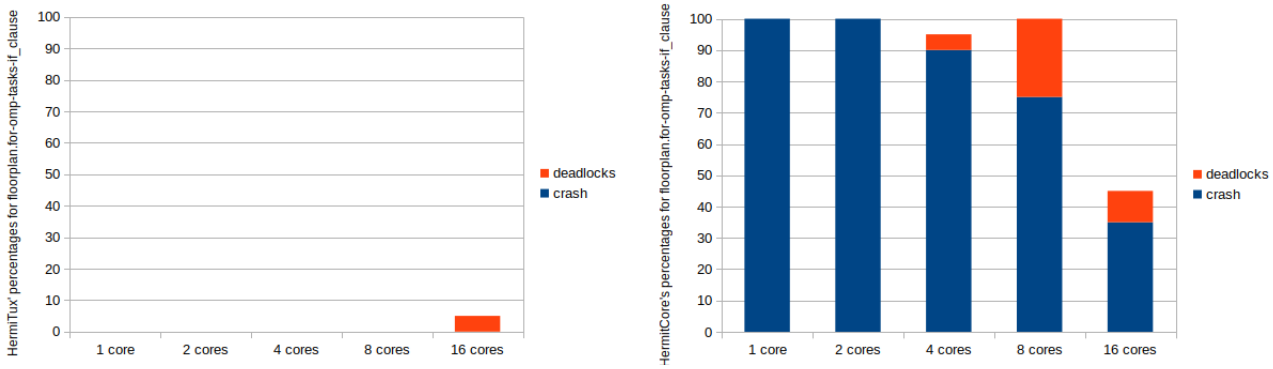

Figure 7: Stability evaluation for several implementations of floorplan.omp-tasks-if_clause.

executions. However with HermitCore, the executable cannot complete a single time. The majority of the HermitCore executions are terminated because of a crash. The rest do not complete because of deadlock, and must be killed by the timeout command. Among the 42 benchmarks used for this study, 2 benchmarks fall into this category.

The influence of the number of cores on the stability of HermiTux and HermitCore is unclear. Although the numbers measures by our experiences tend to show that deadlocks and crashes appears more often when the number of cores allocated is equal or above 4, there is not clear tendency that can be extracted from the data. HermiTux stability is quite good for a majority of the benchmarks (for 31 benchmarks among the 42 we used for the studies) as we can see in the figures 5, 7. For this benchmarks, HermitCore experience more stability issues than HermiTux. For theses benchs, the number of crashes and deadlocks are higher with the highest number of cores (4, 8 and 16) for both HermitCore and HermiTux. But there are also pathologic cases where HermiTux is unable to complete a single execution, regardless the number of core. An interesting fact is that there are also pathologic cases where HermitCore is unable to complete a single execution regardless of the number of core, but HermiTux manage to completes successfully. Figures 6 and 7 shows theses two types of pathological cases. With strassen.omp-tasks benchmark, we see that the number of deadlocks tends clearly to increase with the number of core.

5.6.2 Stability by the benchmark version used

The bots benchmarks are used to evaluate the OpenMP tasks paradigm. To do so, the same problem is solved by different implementations of that paradigm. The following figures shows the stability of HermitCore and HermiTux for all the implementations resolving a given problem. For example, the alignment problem is solved by 4 implementations (for-omp-tasks, for-omp-tasks-tied, single-omp-tasks and single-omp-tasks-tied). Each diagram present the number of crashes or deadlock measured for 20 executions, with 8 cores allocated to the computation.

Figure 8 shows that there is a class of benchmarks for which the number of crashes and deadlocks measured are in the same order. HermiTux shows a very good stability for theses benchmarks, regardless of the implementation. HermitCore is less stable, with more crashes and deadlocks.

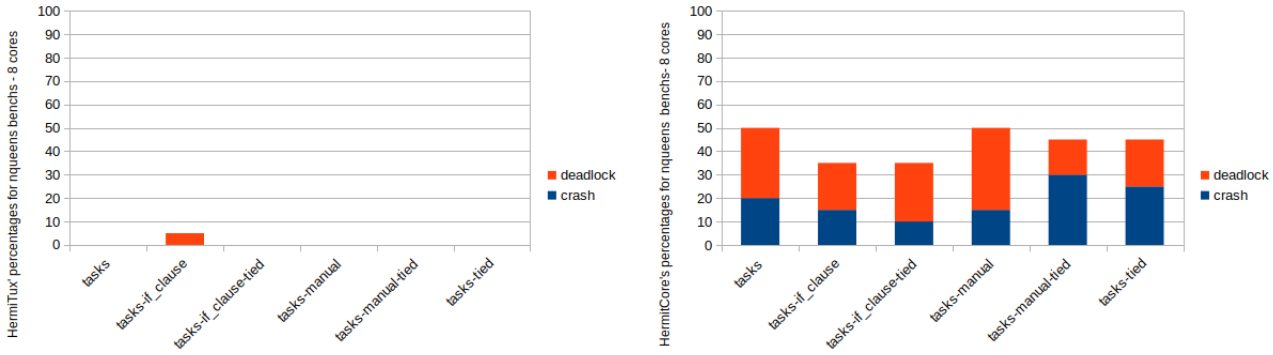

Figure 8: Stability evaluation on 8 cores for several implementations of nqueens problems.

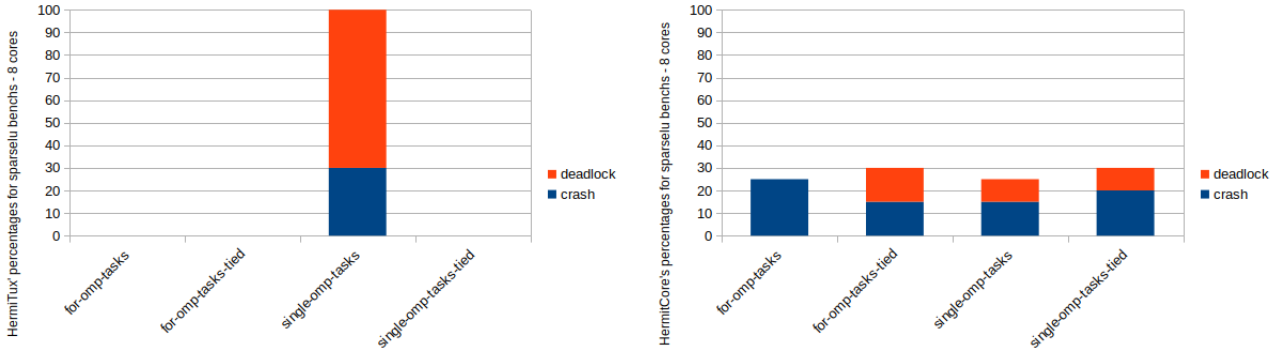

Figure 9: Stability evaluation for several implementations of sparselu problem.

One interesting benchmark is the one presented in Figure 9 . We observe here that there is one implementation of sparselu benchmarks that cannot complete successfully with HermiTux. There is only for the single-omp-tasks implementation that HermiTux has a higher number of crashes than HermitCore.

One last problem to consider, is the alignment benchmark shown in figure 10 . In this figure, we see that an execution with HermiTux always crash. For this benchmark, HermitCore has a better compatibility than HermiTux.

To determine if the stability issues experienced by HermiTux and HermitCore were caused by particular OpenMP paradigm, we observed the stability issues experienced for differents implementation of a same problem. With the figures 8 and 10, we think that regardless of the paradigm used for the implementation, the tendencies are the same for a given problem. It seems that the crashes are not due to an unsupported feature of OpenMP, but rather to the problem itself. There is one exception to this conclusion: The figure 9 shows that HermiTux cannot complete a single execution of the single.omp-tasks implementation of the sparselu problem, but can complete without experiencing errors for other implementations. It is unclear if this high rate of failure is only related to the single-omp-tasks paradigm. Indeed, the figure 10 shows that not only the single-omp-tasks paradigm can fail with HermiTux, but also the for-omp-tasks, for-omp-tasks-tied and single-omp-tasks-tied. We believe that the stability greatly depends

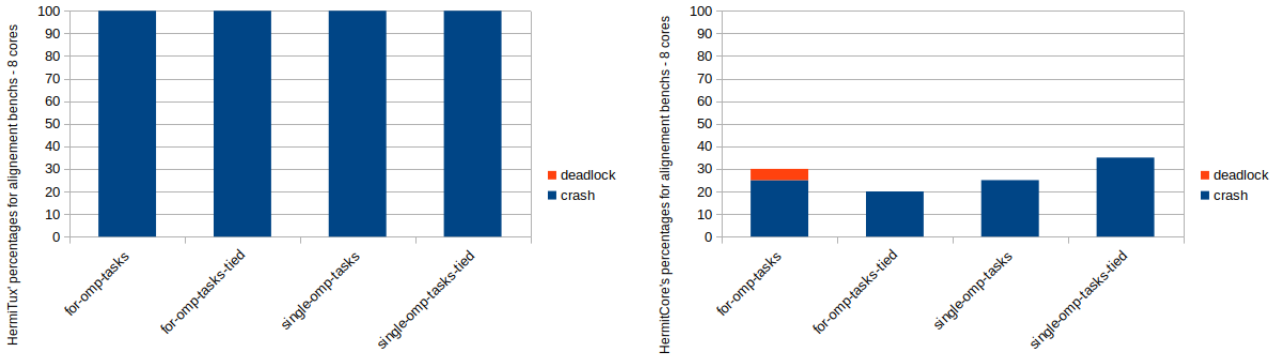

Figure 10: Stability evaluation for several implementations of alignment problem.

on the functionalities provided by the system. If a feature required by one or several implementations is not completely (or not at all) provided by the unikernel, the user is likely to experience trouble at execution.

5.7 Conclusion

Our experiences show that HermitCore often experienced crashes and deadlock for the Bots benchmarks while HermiTux has a better stability. Hunting bugs in HermiTux' kernel has greatly improved its stability. We believe that performing the same engineering work on HermitCore's kernel could also improve its stability results. However, this require consequent engineering efforts: race conditions are one of the hardest type of bugs to fix. We can say that unikernels are not ready for running OpenMP workload in production yet. However, with development effort and some engineering, it should be possible to significantly improve their stability for parallel applications.

6 Performance evaluation

6.1 Overview

In this section, we present the performance evaluation of HermitCore and HermiTux unikernels, with a subset of the Bots benchmarks and a subset of the Rodinias benchmarks. The goal of this study is to determine if executing OpenMP application with unikernels reduce their execution time compared to a Linux execution.

6.2 Metrics

In this section we refer to "performance" as the execution time measured by the benchmarks. Increasing the performance means to reduce the execution time.

Another metric measured is the number of system calls performed by an execution of the executable on a Linux kernel. We measure the system calls performed by Linux with the `strace` command. System calls performed by unikernels are not measured for two reasons:

- In unikernels, system calls are not treated as system calls.
- Due to unikernels limitations, counting the number of system calls performed is not easy as running an executable under `strace`.

However, the code executed is the same for Linux and unikernels. So executing the benchmarks with unikernels is likely to invoke the same amount of system calls.

6.3 Parameters

We consider the following parameters for our study. The first one is the unikernel used for running the benchmark. We consider HermiCore, HermiTux and Linux (a Debian 10 distribution). The goal of this parameter is to compare performances between unikernels and Linux. The second parameter is the number of cores and threads allocated for the execution. With this parameter, we want to compare the speedup between unikernels and Linux. The benchmarks are run on 1, 2, 4, 8 and 16 cores and threads, and with hyperthreading disable.

The third parameter is the program variant used. As shown in Figure 4, the bots benchmarks are composed of several implementations solving a same problem. For example, the `nqueens` problem is solved by the following implementations of the tasks paradigm:

- `omp-tasks`
- `omp-tasks-tied`
- `omp-tasks-if_clause`
- `omp-tasks-if_clause-tied`
- `omp-tasks-manual`

- `omp-tasks-manual-tied`

With this parameter we want to compare execution time between unikernels and Linux in function of the variant used.

Finally, the last parameter is the CPU model that execute the bench. The goal of this parameter is to determine if unikernels are able to take advantage on Linux only on specific CPU models (old models, specific architecture, etc.).

6.4 Experimental methodology

The Bots benchmarks measure the time required to solve their problem. After the program has been loaded, before starting the problem-related computation, it takes a starting timestamp. Once the computation is over, it takes another timestamp and computes the elapsed time by subtracting the first timestamp to the latter. The measured elapsed time is then printed to its standard output. For a given executable, we redirect this output to a log file. Benchmarks are executed 20 times each, and we compute their mean execution time. Because unikernels have stability issues, the benchmarks execution is monitored by a script. This script ensures that each 20 values used to compute the mean are values coming from a successful execution. If a crash or a deadlock occurs during an execution, its results is discarded, and another execution is performed. Some benchmarks were put aside for this evaluation. Section 5 has shown that there are executables that unikernels cannot execute successfully at all (for example alignment.for-omp-tasks never completed successfully with HermiTux).

6.5 Results

This subsection presents several graphs. Mean execution times graphs present the execution time of a given benchmark. Speed up graphs show the acceleration of a given benchmark for a given number of core, in comparison of its execution on one core. It is computed by dividing the mean execution time for a 1 core execution by the mean execution time for a given number of core execution. System calls graphs present the number of `sched_yield` system calls performed by the benchmarks when executed on a Linux kernel.

Error bars We know that error bars are crucial for interpreting results and data distributions. However, I did not manage to print error bars on time on the following graphs. They will feature on the final report that will be published by the Inria.

6.5.1 Bots benchmarks

Figure 11 shows that unikernels and Linux have similar performances with `fft.omp-tasks` and `sparselu.for-omp-tasks`. This is the case for most of the benchmarks. Before looking at this graph, we were expecting HermitCore to be faster than HermiTux.

Indeed, even if HermiTux' system calls are faster than regular OSes system calls, they can't compete with HermitCore. While HermiTux accelerate the system calls by its simple and optimised system calls handler (it doesn't perform the world-switching which is not required, do not used the `sysret` instruction that is slower than classical return

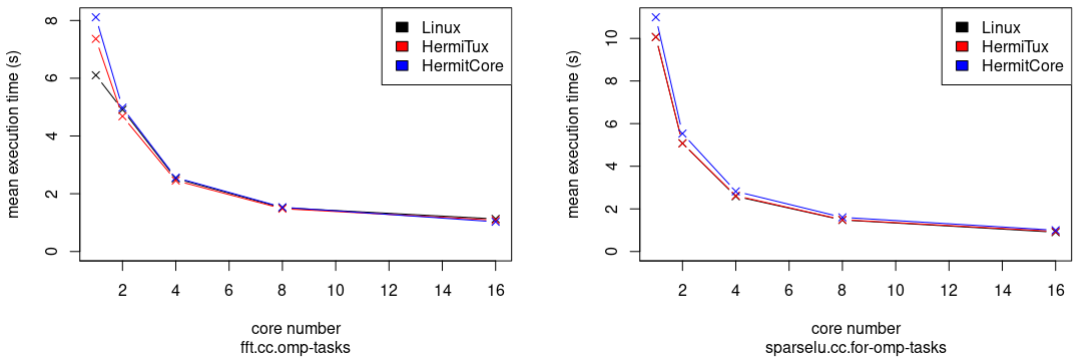

Figure 11: Mean execution times for fft.omp-tasks and sparselu.for-omp-tasks benchmarks.

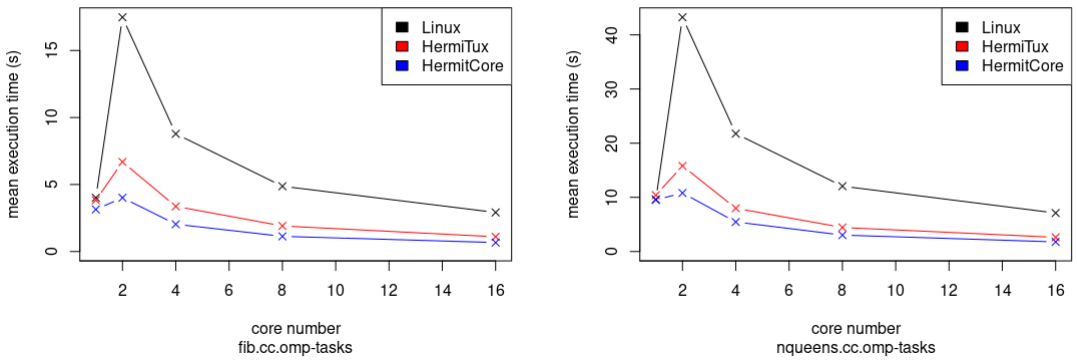

Figure 12: Mean execution times for fib.omp-tasks and nqueens.omp-tasks benchmarks.

instruction...), HermitCore's system calls are treated as common function calls. Instead of saving values into special registers, triggering an execption to handle the system call, HermitCore simply call a function from the kernel, which is really faster.

However, we observe in figure 11 that HermitCore is slower than HermiTux. This is not an isolated case, because on many benchmarks we ran HermitCore is slower than HermiTux. The reason of this overhead for HermitCore is unknown to us. Because HermitCore application are not compiled with the same compiler and OpenMP runtime as HermiTux applications and Linux executables, we cannot draw conclusions about the causes of this overhead.

Figure 12 shows benchmarks where unikernels dramatically accelerate execution time. `fib.omp-tasks` and `nqueens.omp-tasks` benchmarks are two times faster when they are executed with a unikernel than with Linux! The parallelisation paradigm used for resolving the `fib` and `nqueens` problems seems to be ineffective with Linux. We observe for theses benchmarks that Hermitcore is faster than HermiTux. When Unikernels have results similar to the Linux kernel, HermitCore seems to be slower than HermiTux. But

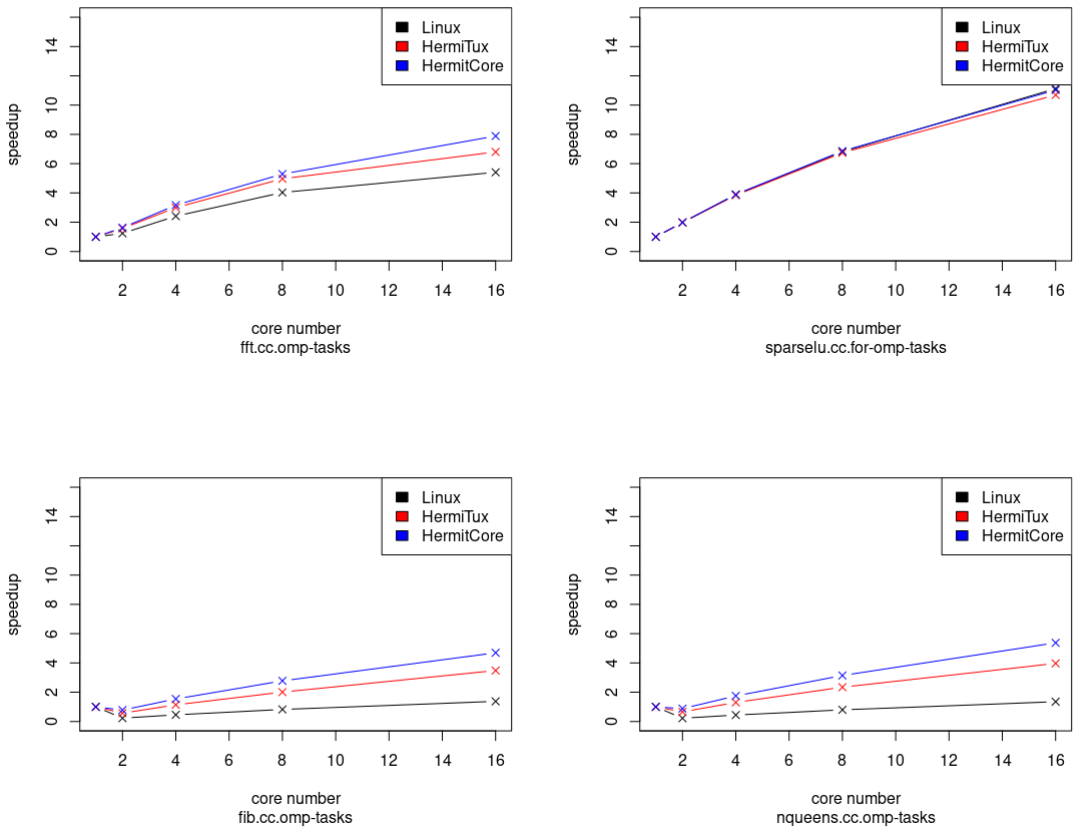

Figure 13: Speedup curves of the benchmarks shown above.

when Unikernels are computing faster than Linux, it seems that HermitCore takes the advantage on HermiTux. It is difficult to explain this difference between HermitCore and HermiTux, because we do not control the OpenMP runtime and the compiler used for HermitCore. Theses differences might be related to different optimisations performed by LLVM and Intel OpenMP runtimes for example.

The speedup graphs in Figure 13 reveal that the speedup of theses benchmarks is not very important. We observe that the `fft.omp-tasks` and `sparselu.omp-tasks` benchmarks, Linux is able of accelerating, even if the acceleration is not as high as we could expect. The speed up of `fib.omp-tasks` and `nqueens.omp-tasks` shows that unikernels are able to accelerate the benchmarks execution while Linux struggle to accelerate more than one time.

System calls Because unikernels are known to accelerate the execution of programs by reducing the system call overhead, we measured the number of systems calls performed by an execution of the benchmarks executed above.

Figure 14 presents a graph where the number of `sched_yield` systems calls performed by an execution of the benchmarks on Linux is plotted in function of the number of cores allocated. We observe that there are benchmarks that clearly stands out by

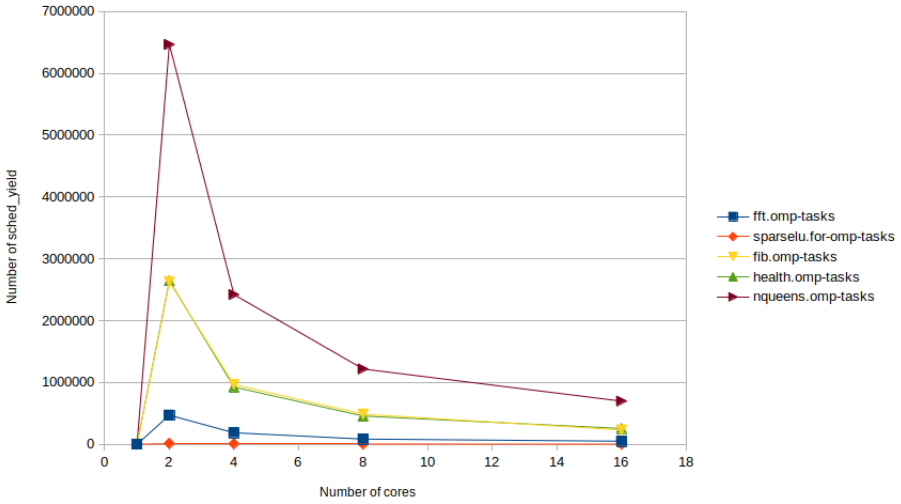

Figure 14: Systems calls performed during the execution of the benchmarks.

performing a lot of `sched_yield` system calls. Theses benchmarks are `fib.omp-tasks` and `nqueens.omp-tasks`. There is a peak of system calls for 2 cores, after which the number of system calls performed decrease with the number of cores allocated. Looking at theses curves, and the speedups of the corresponding benchmarks, we can infer that the great decrease of the speed up is due to the high number of `sched_yield` performed. This seems coherent with the fact that as their number decrease with the increasing of the number of cores, the speed up of the benchmark is increasing. Also, we note that for benchmarks where we observe similar performance between unikernels and linux, we observe smaller number of `sched_yield` which also seem coherent.

SPECTRE/Meldown impact Since the discovery of the SPECTRE and Meltdown security breaches in Intel processors, the cost of system calls have greatly increased. The `nova` node used to run the benchmarks is equipped with an Intel CPU launched before the discovery of theses breaches. To verify if the impact of system calls on the time execution was due to the correction of the CPU security issues, we ran the benchmarks on another machine of the Grid'5000 infrastructure. This machine is equipped with a more recent CPU, where the SPECTRE and Meltdown breaches have been fixed by hardware design. The system calls should have a lower overhead than on the machine we used for our previous measures. We want to see if the tendencies we observed are still valid with recent CPUs.

Figure 15 shows that the graphs have the same outline when the benchs are executed on the new CPU. We note that the gap between Linux and unikernels in their performance is smaller, but still present. For the benchmarks where Linux and unikernels have similar performances, the times are almost identical. The speed ups shows the same tendencies as described above.

System calls and LLVM runtime The question we asked to ourselves when analysing the results of this experience is "Why are some benchmarks performing so many `sched_yield` system calls ?". Our researches showed that is this the LLVM OpenMP runtime that use

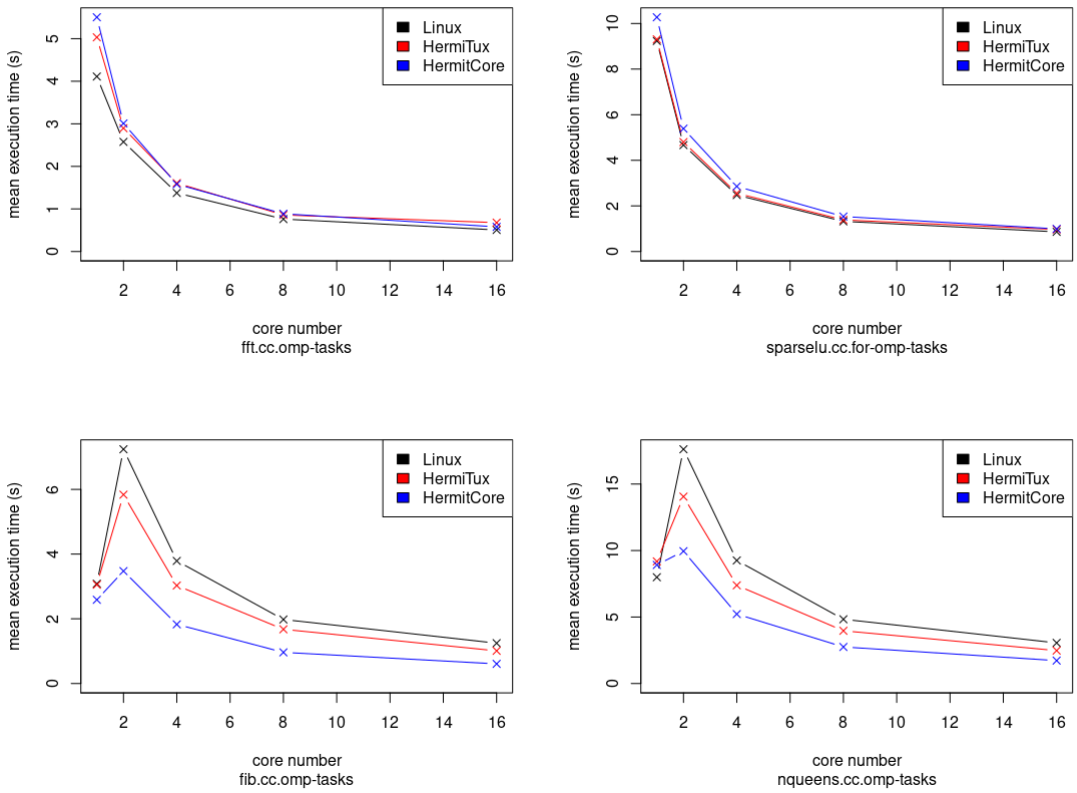

Figure 15: Mean execution times of the 4 benchmarks presented above, executed on a gros node of Grid'5000.

this system call, for its "tasks stealing paradigm". This systems call is used to give back the CPU to the kernel when there are no tasks to steal. This means that there are not enough tasks to perform for `fib.omp-tasks` and `nqueens.omp-tasks` benchmarks. We think that theses two problems are not very representative of HPC applications. The two other problems, `fft.omp-tasks` and `sparselu.for-omp-tasks` seems more representative of HPC applications, because they are more compute intensive.

6.5.2 Rodinias benchmarks

This section details the experiences we conducted with the Rodinias benchmarks `LavaMD` and `lud_omp`. Theses benchmarks consist in more compute intensive applications that are more representative of HPC applications.

Figure 16 shows the execution time for the `LavaMD` and `lud_omp` benchmarks. The execution times of unikernels and Linux are very similar. Same for the speed up curves, that are almost identical.

We lalso ook at the system calls performed by Linux when executing theses benchmarks. The number of `sched_yield` call measured shows that the two Rodinias benchmarks we used for this experience are more compute intensive. Figure 17 shows that the number of `sched_yield` system calls performed by the two benchmarks are really low. This confirms the fact that unikernels take advantage when Linux is slowed down by

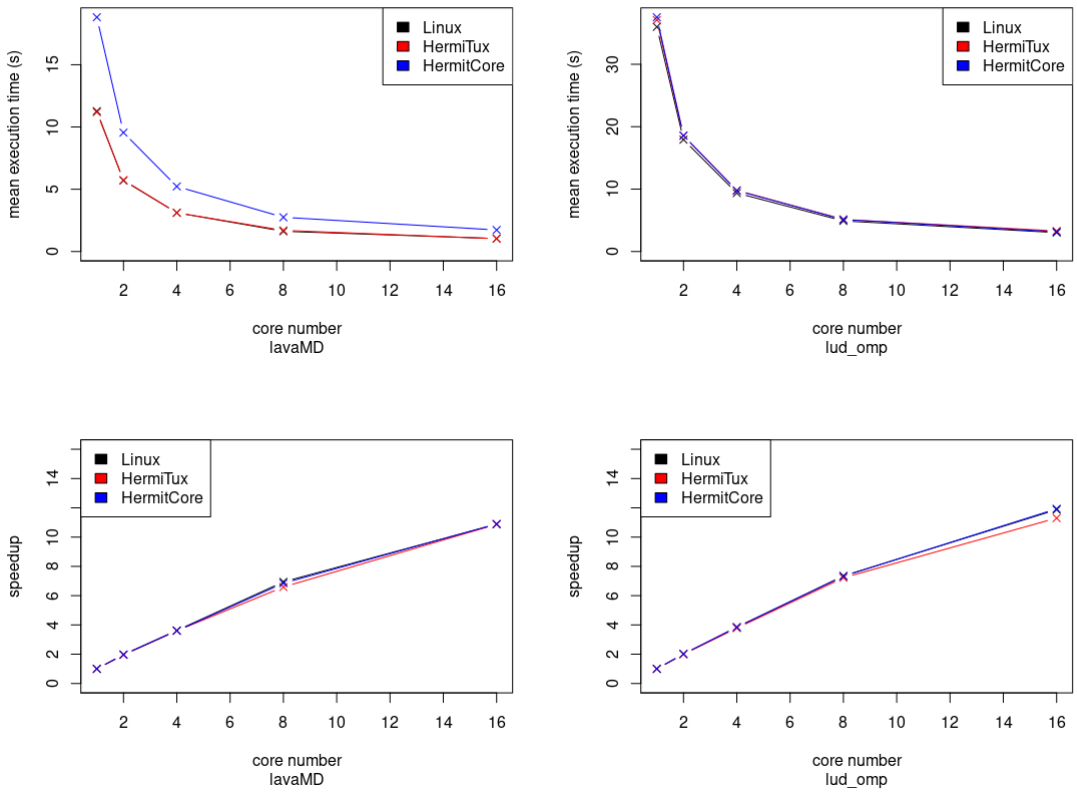

Figure 16: 2 Rodinias benchmarks, executed on a gros node of Grid'5000.

the many systems calls performed by a program. When a program is compute intensive, unikernels does not stand out by accelerating its execution.

6.6 Conclusion

Our study shows that unikernels dramatically accelerates the execution of programs when the system calls overhead is slowing down the execution. Unfortunately, this is not a characteristic of HPC applications. They tend to have long computation phases, where no system call or any blocking operation is involved, to maximise performance.

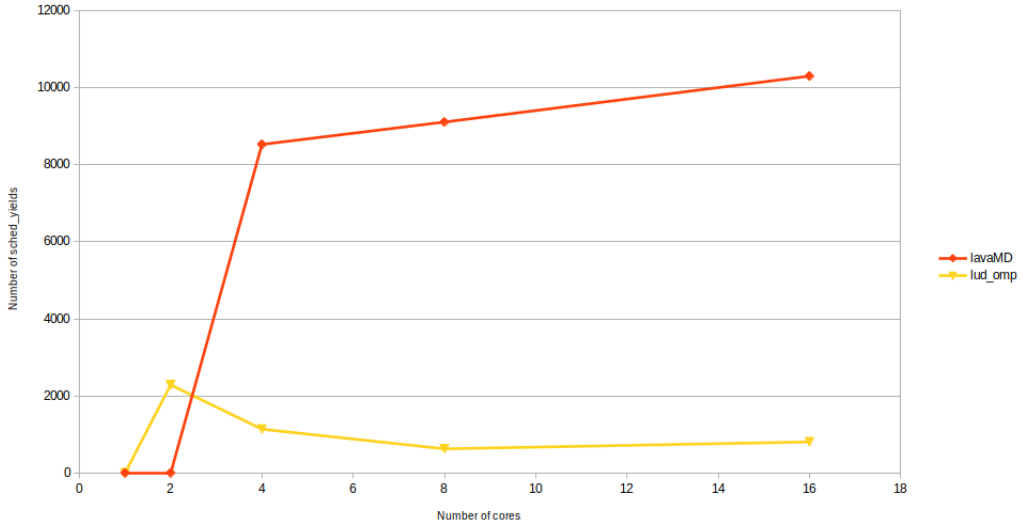

Figure 17: Systems calls performed for lavaMD and Lud benchmarks.

7 Conclusion

The goal of this paper is to evaluate unikernels suitability for HPC applications. Unikernels were originally designed for cloud computing few years ago. The objective was to determine if they were now able to improve the execution of HPC applications.

We chose two unikernels for our studies: HermitCore and HermiTux. Theses unikernels are both able to execute C/C++/FORTRAN application, and support multi-core and OpenMP.

Our first study (in section 5) shows that Unikernels still have stability issues in running OpenMP application. Our measures show that random crashes and deadlocks can occur at important rates. With this study, we have been able to give feedback to HermiTux' main developer, and improve its stability toward OpenMP applications. We believe that supporting OpenMP application without crashing and deadlock is only a matter of engineering. This engineering require a lot of debugging and development efforts.

Our second study (in section 6) confirms the fact that unikernels greatly reduce the overhead due to system calls. The use of unikernels can dramatically reduce the execution time of the benchmarks that perform a lot of system calls. However, HPC applications tend to have very long computation phases, without performing system calls or I/O operations. As soon as we start evaluating unikernels on more compute intensive benchmarks, unikernels does not accelerate the execution of the application.

Unikernels are able to execute HPC OpenMP applications. However, most of the unikernels presented in this paper are either experimental prototypes, or not in their final version. The technology still need some maturation to be ready for production. Even if a few years may be required to improve unikernels design, there are open questions left for other studies concerning unikernels such as their impact on system noise, and their support of Infiniband and GPGPU drivers.

Future work

In section 6 we showed that Unikernels performances were related to the number of system calls performed by the application they are executing. Our performance evaluation for the Bots benchmarks is incomplete because of crashes and deadlock that occurs on several Bots benchmarks (see section 5). However, it could be interesting to retrieve the number of system calls performed by theses benchmarks during an execution on a Linux kernel. With this number, we could be able to classify the behaviour that unikernels would likely have if they could execute theses benchmarks without crashing.

In section 1 we explained that Unikernels are known to lower the system noise. We plan to write a simple application to determine if Unikernels experience lower OS-noise than regular Linux kernels.

In section 2 we quickly presented the Lupine Linux unikernel and why it may be an interesting candidate for executing HPC applications. This unikernel need to be analysed to confirm that it is compatible with multi-core architecture and OpenMP application as well as C/C++ and FORTRAN programming languages.

An interesting question raised by this study is why some OpenMP applications are performing so many system calls. We think that this may vary from an OpenMP runtime to another (from LLVM to GCC for example). A future work to perform is to run the same benchmarks with different OpenMP runtimes, to see their impact on the performance. This require some engineering effort, because using a different OpenMP runtime with a unikernel is not a trivial task.

8 Personal review

This graduation project allowed me to work on a very recent and experimental technology : Unikernels. Working on such a new topic was very interesting, because it is a field of study that is still unknown for a lot of people. Thanks to the laboratory and the Avalon Team, I have been able to collaborate with Pierre Olivier, a researcher who has developed HermiTux, a unikernel which is described in this report. Thanks to this collaboration, I learned a lot about unikernels and Operating Systems.

The field of unikernels is part of the Operating Systems domain. I had to face a lot of technical difficulties that were due to the unusual context of Unikernels. Usually, compiling C/C++ applications is usually easy, it can quickly become complicated for compatibility reasons between unikernels and POSIX-API, binary format, missing headers... While experimenting with unikernels and resolving issues, I regularly used tools that I had never used before : `file`, `readelf`, `objdump`, `nm`, `c++filt`, `elfedit`... Working with a research mindset also enabled me to be more demanding on my work: I had to become aware of what libraries I was compiling programs with, why I was compiling with one specific library and not another etc.

Working in the Avalon Research Team was very stimulating. By participating at the workgroup meeting every week, I had a glimpse about several field of study that are currently explored by researchers. It was also a very good way to continue my reflection about my future career, and helped me to decide whether or not start a thesis after graduating.

Unfortunately, theses interactions were quite limited due to the remote work imposed by the sanitary conditions. The pandemic of Covid-19 had a great impact on the organisation of work, and the research team I was working in was no exception. This had the effect to increase some difficulties I encountered during my project. Even if I had the possibility to contact people by mail or to do video conferences when I was facing difficulties, I realised that nothing can replace human interactions. Communicating while social distancing is very difficult, and tiring for a long period. This situation showed me that I need more social interactions than I thought, before the start of the pandemic, and I think this is one of the most important things I learnt during my project. This situation required me to be very organised, and communicative. I noticed that I progressed better when I was setting clear objective to myself on a short period of time rather than having vague objectives. Also, it was important for me to communicate my progression to my referents. I usually lack of organisation and I am not the most communicative person, so this experience enabled me to pratice theses skills. This is something that will be surely useful in my future career.

9 References

[1] Sébastien Badia, Alexandra Carpen-Amarie, Adrien Lèbre, and Lucas Nussbaum. Enabling large-scale testing of IaaS cloud platforms on the grid'5000 testbed. In *Proceedings of the 2013 International Workshop on Testing the Cloud - TTC 2013*, page 7. ACM Press.

[2] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257. IEEE.

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE.

[4] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *2009 International Conference on Parallel Processing*, pages 124–131. IEEE.

[5] Dawson R Engler, M Frans Kaashoek, James O'Toole, and M I T Laboratory. Exokernel: An operating system architecture for application-level resource management. page 16.

[6] Antti Kantee and Justin Cormack. Rump kernels : No OS? no problem!

[7] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv— optimizing the operating system for virtual machines. page 13.

[8] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, Joel Nider, Mike Rapoport, and Costin Lupu. Unleashing the power of unikernels with unikraft. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 195–195. ACM.

[9] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394. ACM.

[10] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15. ACM.

[11] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: A unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, pages 1–8. ACM.

[12] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. page 12.

[13] Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. 57(1):61–69.

[14] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE 2019*, pages 59–73. ACM Press.

[15] Livio Soares. FlexSC: Flexible system call scheduling with exception-less system calls. page 14.

Evaluating unikernels for HPC applications

Pierre JACQUOT

Abstract

Unikernels are lightweight single-application operating systems. They are designed to run as virtual machines, but some are able to run on bare metal too. They are quite popular in the system research community due to the increase of performance they can provide. By reducing the system call overhead and the OS-noise, they might be a good alternative to containers for HPC applications. This report is an early version of the final report that will be published later by the Inria. It evaluates the suitability of unikernels for HPC applications. This is done by conducting stability and performance studies with the Bots benchmarks and the Rodinias benchmarks. They are performed on multi-core architectures, on single node.

Keywords: Unikernels ; Linux Kernel ; Operating System ; HPC ; System Calls.

Résumé

Les unikernels sont des systèmes d'exploitation mono-application. Ils sont conçus pour fonctionner de manière virtualisée, même si certains sont capable de fonctionner sans hyperviseur. Leur récente popularité au sein de la communauté système de la recherche provient de l'amélioration des performances qu'il peuvent fournir. En réduisant le surcoût liés aux appels systèmes, ainsi que le bruit système ils pourraient s'avérer être une bonne alternative aux conteneurs pour les applications HPC. Ce rapport est une version anticipée du rapport final qui sera publié par l'Inria. Il a pour but d'étudier la pertinence des unikernels pour les applications HPC. Pour ce faire, des études de stabilité et de performances sont réalisées avec les benchmarks Bots, et les benchmarks Rodinias. Ces études sont réalisées sur des machines mono-noeud, dotées de processeurs multi-coeurs.

Mots-clés : Unikernels ; Noyau Linux ; Système d'Exploitation ; Calcul Haute Performance ; Appels Systèmes.