UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

MĂSURAREA PERFORMANȚELOR în UNIKRAFT cu UNIPROF

Viorel Gabriel Mocanu

Coordonator științific:

Costin Raiciu
Răzvan Deaconescu
Felipe Huici

BUCUREȘTI
2021

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

UNIKRAFT PERFORMANCE MEASUREMENT with UNIPROF

Viorel Gabriel Mocanu

Thesis advisor:

Costin Raiciu
Răzvan Deaconescu
Felipe Huici

BUCHAREST
2021

CONTENTS

I would like to thank Răzvan Deaconescu and the entire Unikraft team for all the advice and help they offered me throughout the project.

SINOPSIS

Un unikernel este o metodă de virtualizare ce înglobează toate aspectele pozitive a celorlalte soluții de virtulizare, consumul de memorie scăzut și viteza containerelor, dar și nivelul de securitate pe care îl oferă mășinile virtuale prin izolare. Unikraft este un unikernel ce creează o imagine a aplicației împreună cu toate dependințele sale, într-un executabil, printr-un proces ușor și rapid pentru utilizator, dispunând de un mecanism de configurare. Măsurarea performanței aplicațiilor devine o etapă din ce în ce mai importantă, iar Unikraft nu dispune de un astfel de utilitar pentru a putea măsura eventuale blocaje ce pot apărea în interiorul aplicației. Obiectivul tezei mele a fost să integrez un utilitar de măsurare a performanțelor pentru unikernel-uri, Uniprof, cu Unikraft și să dispun datele într-un format mai ușor de înțeles folosindu-mă de Flame Graph. După terminarea tezei se pot măsură performanțele și se pot observa potențialele blocaje pe care o aplicație le poate avea rulând în cadrul unui hypervizor Xen.

ABSTRACT

Unikernel is a virtualization method that encompasses all the positive aspects of other virtualization solutions, low memory consumption and speed of containers, but also the level of security that virtual machines offer through isolation. Unikraft is a unikernel that creates an image of the application together with all its dependencies, in an executable, through an easy and fast process for the user, having a configuration mechanism. Measuring the performance of applications is becoming an increasingly important step, and Unikraft does not have a tool to measure any bottleneck that may occur inside the application. The objective of my thesis was to integrate a performance measurement utility for unikernels, Uniprof, with Unikraft and to display the data in a pretty format using Flame Graph. At this moment, you can measure the performance and you can see the possible bottlenecks for an application running on a Xen platform.

1 INTRODUCTION

1.1 Virtual machines, Containers and Unikernels

Container technology has grown exponentially in recent years.[1] It is considered one of the best methods for packing an application so it can be run with its dependencies. Some of the reasons why developers have adopted this technology to the detriment of virtual machines are that they are lighter, require less space and are faster. But all these advantages, of course, come with a disadvantage, they do no perform so well from a security perspective, as many exploits have proven.[2]

Of course, we also have the alternative of virtual machines, they solve most of the security problems from containers having better isolation between the hypervisor and the virtual machines. But one major downside for virtual machines is that they are heavyweight, the resources provided by them are too much for a single application.

A solution to incorporate all the advantages of containers and virtual machines is using unikernels.[3] Unikernels are single address space applications packing all required software components, including those that usually run in the operating systems. Unikernels consists of several internal libraries and the actual application running on top of those.

There are also some downsides that unikernels have, such as lack of flexibility, requires some libraries operating systems to be set up, a different package for each application. Also the lack of debugging tools, these things make them difficult to use in production.

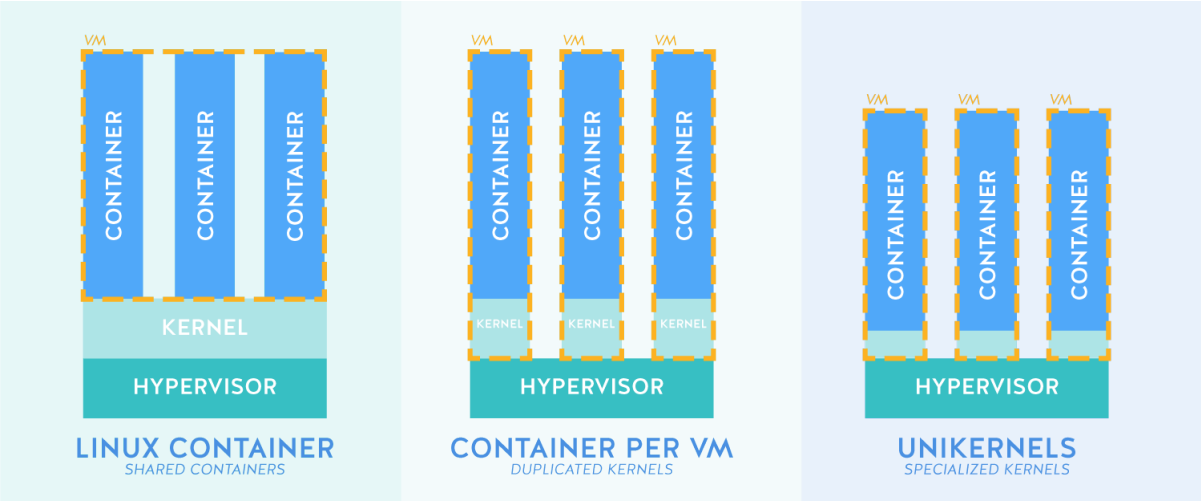

Figure 1: Types of virtualization[4]

1.2 Unikraft

There are several implementations for unikernels, but one of them is Unikraft, a Xen project.[5] Unikraft creates an image of the application together with the internal and external libraries and runs it as a virtual machine with Xen or KVM hypervisors or in Linux user space. Unikraft has a configuration menu in which for each application you can set certain libraries, in this way you can select only what your application needs, thus minimizing the attack area.

1.3 Motivation

One reason unikernels are not adopted faster and used in production environments is the lack of external tools that can support developers to create applications more easily. Although they are seen as very complicated entities, in their development it's easier to debug because the application, libraries and operating system are represented by a single binary, so you can debug all components at the same time. Regarding the identification of a problem, we can of course use gdb because it also supports Xen and KVM hypervisors. Nevertheless, unikernels do not yet have well-defined tools for measuring performance.[6]

1.4 Objectives

Currently, Unikraft does not have a tool to measure performance and to identify possible bottlenecks that may occur for an application. Therefore, one of the goals of the project is to integrate an existing tool, Uniprof[7] with Unikraft and create Flame Graphs[8] to produce insightful visualizations of stack traces.

1.5 Profiling

Profilers are software tools that are able to measure the performance of an application. For example using a profiler we can estimate how long it takes to execute a function, who called that function, how much memory is used and how many times that function is called. [9]

Depending on how accurate the results are desired or how much effort must be put into providing this data of performance, there are several methods for profiling. Mainly all these methods fall into two categories: Instrumenting and Sampling.

1.5.1 Sampling

Sampling profilers interrupts the application at regular intervals in order to be able to inspect the state of the processor and take data from it such as the instruction pointer, but also other registers. The main advantage of this method is that it does not require changing the code within the application. Depending on the frequency at which the application stops this can bring an overhead during its running, but it is certainly lower than in the case of instrumentation and in most cases is negligible because the application is stopped only for a short time necessary to read the registers.

1.5.2 Instrumentation

Instrumenting profilers adds code at the beginning and end of the function to be able to measure different performance metrics. It can measure how long it has been in function or what function has called the current function. These can be source-code profilers, this type requires a lot of work because the application had to be modified and the profiling code had to be added for each function we want to monitor. The other type is binary profilers, which use only the executable of the application and automatically introduce in the executable code the methods for profiling.

Both methods are quite accurate compared to sampling because we can know with greater accuracy how much time was spent in a function, their problem is an overhead due to the fact that the added code is executed many times. Due to this overhead, an area with a bottleneck can be misidentified.

1.6 Flame Graph

When we talk about performance analyzers, one of these is a profiling stack. With such a software tool, we can observe which function within our program is called the most or which function spends the most on the processor. Although it is an important metric, it is very difficult to analyze because most stack profiling programs output a dump with all functions called. This can also lead to a file with millions of rows of stack traces that is very difficult to read and interpret, it is almost impossible to figure out which function was called most often and what led to this.

Flame Graph is used to solve this problem, it can interpret the output of a stack profiler and provide an scalable vector graphics file with all function calls, much easier to interpret. An example of a Flame Graph is shown in Figure 2. [8]

In this format we can identify the functions at the top of the graph as the ones that spent the most time on the processor, also this tool can use certain color customizations to more easily see a certain group of functions. Colors only have the role of identification and are not

part of the metric.

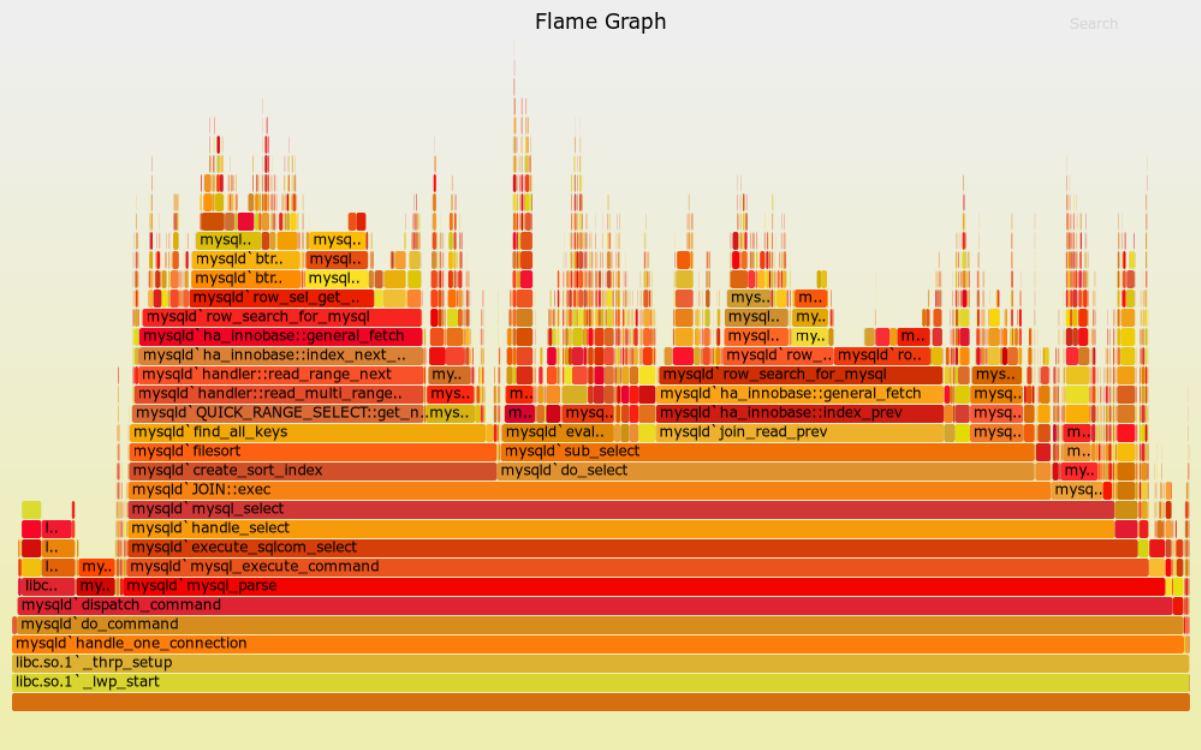

Figure 2: Flame Graph [8]

2 STATE OF THE ART

2.1 Xenctx

For Xen virtual machine environment, we already have some tools for measuring performance. One of them is Xenctx. Xenctx is a stack profiler that uses sampling and is part of the general tools for Xen guests.

Due to the fact that it was not necessarily designed for performance, this tool has some problems. It runs pretty slow, for each trace it lasts about 3ms. Even if it doesn't seem like much, for a high frequency of traces this time can make the difference.

Another problem with Xenctx is that it does not stop the virtual machine when it makes a trace, and this situation can lead to a race condition because the stack can change while reading from it. Also, to solve the symbols Xenctx uses a linear search through page tables which leads to a large overhead.

2.2 Uniprof

Because of these, a new tool, Uniprof, has appeared, specialized for unikernels. It is about 100 times faster than Xenctx because it's using a caching system and solving symbols is done offline and with the help of the nm tool.[10]

Another improvement is that Uniprof pause the virtual machine, creates a trace stack, and then start again the virtual machine, avoiding possible concurrency issues. Traces generated with Uniprof can then be aggregated to produce profiling information and visualization with Flame Graph.

Having Uniprof integrated with Unikraft would be easier to do profiling on all applications for Xen environment. This is useful because we can observe not only the stack trace, but with the Flame Graphs we can also have a visual representation to see which code-paths keep the processor busy.

Another advantage for Uniprof is that stack trace is done from the outside of the unikernel and we don't have to modify any code from our application to run it.

3 ARHITECTURAL VIEW

3.1 Xen

Xen is an open-source hypervisor of type 1 that have two types of virtualizations available: paravirtualization and full virtualization. Paravirtualization means that guest operating systems know they are executing on a virtual machine, this is done by running a modified version of the operating system. This abstraction increases a lot the performance of a virtual machine. Full virtualization offers a complete simulation of the physical system and therefore there is no need to modify the operating system as in the case of paravirtualization, this is an advantage but in terms of performance paravirtualization remains better. Unikraft supports this platform and has some ported applications to run on Xen. [11]

3.1.1 Domains

When we talk about virtual machines in Xen, we are actually referring to domains. As you can see in figure 3, it uses a privileged domain, Dom0, which has the right to access all other domains, including drivers. All other virtual machines are referred to as DomU, unprivileged domains.[12]

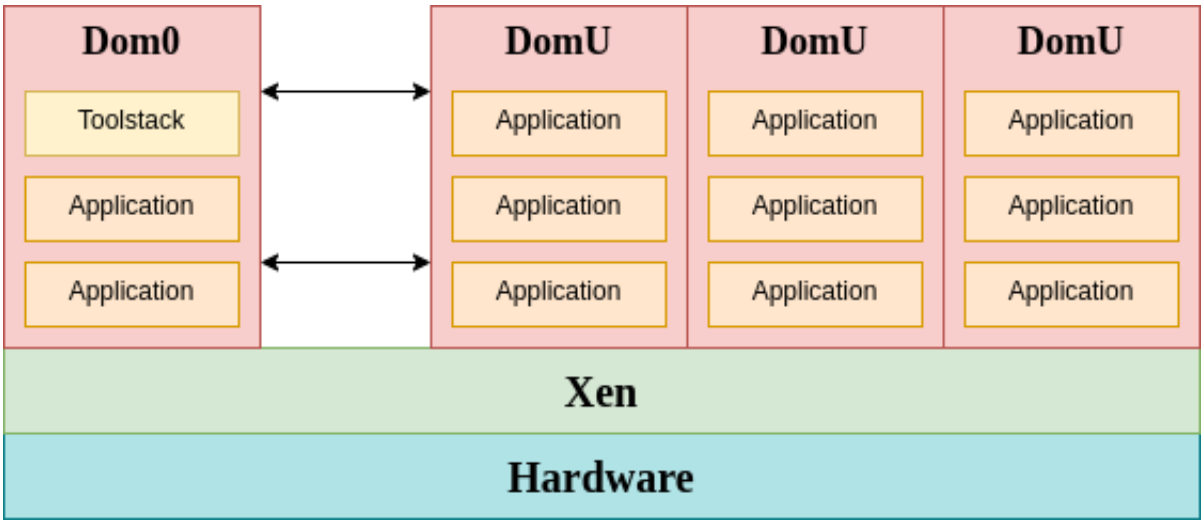

Figure 3: Xen Domains Arhitecture

3.1.2 Hypercalls

Hypercalls use the same principle as system calls, just as are used system calls to connect an application with the operating system, so hypercalls connect applications, domains and the hypervisor. This happens through Dom0 and an interface within the Xen hypervisor. [13]

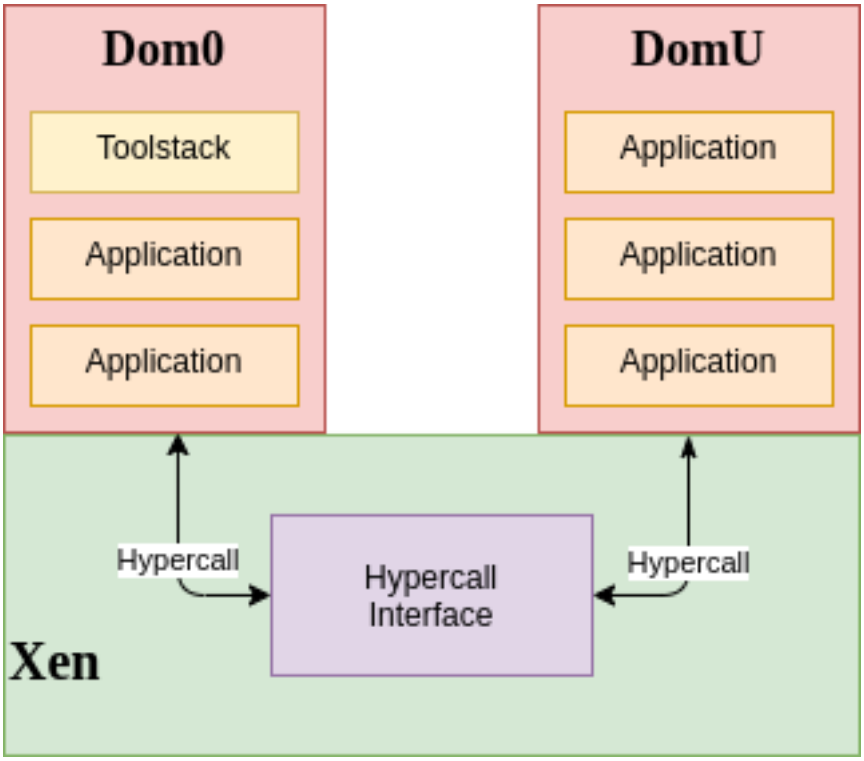

Figure 4: Xen Hypercalls

3.2 Uniprof

As you can see in figure 5 Uniprof has a simple and modular structure. We have general functions declared and defined in include directory, but also functions for binary search. In the xen-interface directory we have only logic for communicating with a Xen domain, both for x86 and for ARM. In the uniprof.c file all the profiling logic happens, where the frequency at which the stack is made is calculated, the virtual machine is stopped, the stack trace is copied, the virtual machine is restarted. Another symbolize.cc program is used to solve the symbols.

Uniprof runs in the privileged domain, Dom0, it needs rights to interact with the other domains. It is not necessary to modify the application for which we want to observe the stack traces, the first step is to start the application we want to analyze in a virtual machine. The next step is to start Uniprof from Dom0 with the mandatory parameters (the id of the domain for which

[0]https://wiki.xenproject.org/wiki/Hypercall

```
Uniprof
├── include
│   ├── binsearch.h
│   └── config.h
├── xen-interface
│   ├── xen-interface-arm.c
│   ├── xen-interface-common.c
│   └── xen-interface-x86.c
├── configure
├── uniprof.c
└── symbolize.cc
```

Figure 5: Uniprof - Project Structure

we want to do the analysis, the frequency at which we want to stop the virtual machine to be able to copy a stack trace) and finally we get the file with a lot of stack traces. Of course, the connection between those domains is performed using hypercalls through Xen supervisor and this architecture is represented in figure 6.

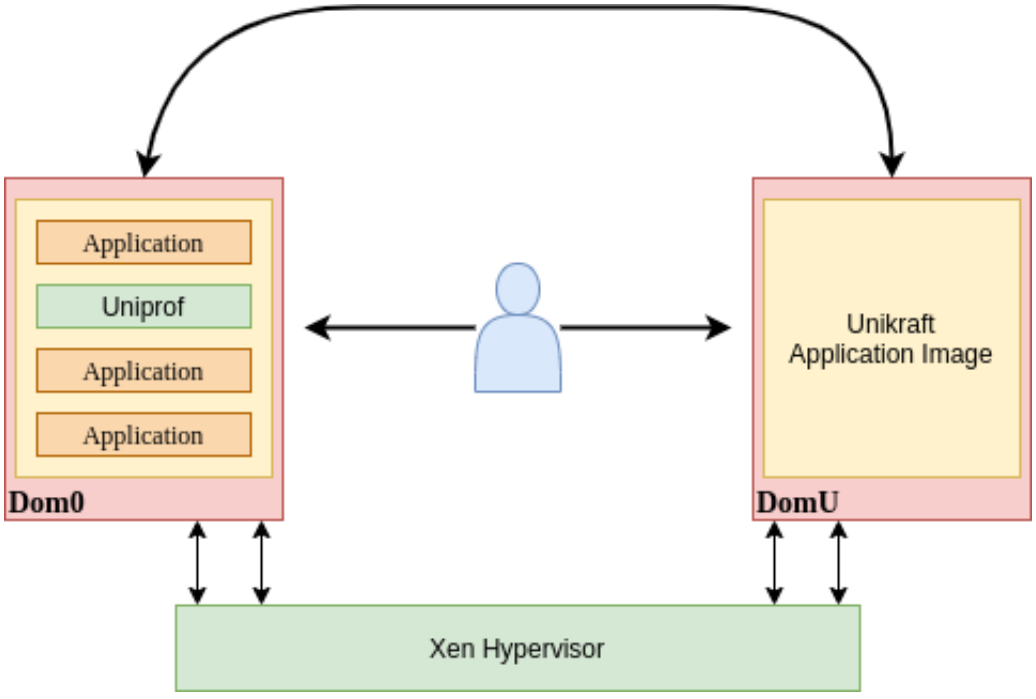

Figure 6: Uniprof Dom0-DomU

4 IMPLEMENTATION DETAILS

4.1 Profiling

Stack profiling is done by collecting a number of stack traces. By inspecting each stack trace we can observe currently run functions, and the functions that lead to the current function being called. This is done by using the instruction pointer(RIP/EIP) and frame pointer(RBP/EBP) registers. This approach is shown in Figure 7.

The instruction pointer gives an address to the current instruction, while the frame pointer has an address that points to the bottom of the current function stack frame. As each function is called, registers are pushed onto the stack before the function starts its execution. The return address is also pushed onto the stack in a fixed location relative to frame pointer, therefore by traversing the frame pointers and reading the return addresses, a stack trace is created. With the help of a symbol table, these raw addresses can be translated into functions.

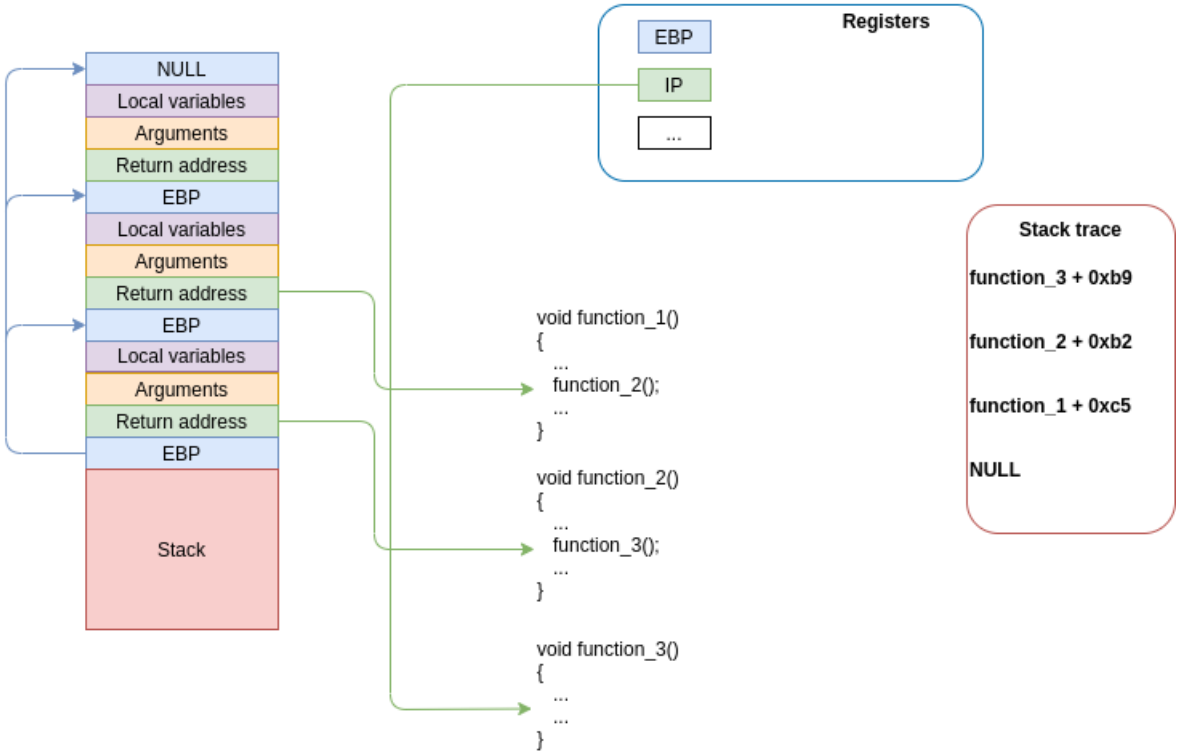

Figure 7: Stack walk

4.1.1 Registers

In order to make a stack trace, we need access to registers, to find out where we are at the moment in the code, using the instruction pointer. We also need a frame pointer to find out the size of the stack frame and the return address for walk down the stack. To find these values Uniprof uses a hypercall, getvcpucontext() which communicates with the hypervisor to find out the status of the registers as we can see in Listing 4.1. [14]

```
1
2  void walk_stack_fp(int domid, int vcpu, int wordsize, FILE *file, void *
       symbol_table) {
3    int ret;
4    guest_word_t fp, retaddr;
5    void *hfp, *hrp;
6    vcpu_guest_context_transparent_t vc;
7
8    DBG("tracing vcpu %d\n", vcpu);
9    if ((ret = get_vcpu_context(domid, vcpu, &vc)) < 0) {
10     printf("Failed to get context for VCPU %d, skipping trace. (ret=%d)\n",
         vcpu, ret);
11     return;
12   }
13
14   // our first "return" address is the instruction pointer
15   retaddr = instruction_pointer(&vc);
16   fp = frame_pointer(&vc);
17   DBG("vcpu %d, initial (register-based) fp = %#"PRIx64", retaddr = %#"
       PRIx64"\n", vcpu, fp, retaddr);
18
19     while(fp != 0) {
20         .....
21     }
22
23     .....
24 }
```

Listing 4.1: Get registers values using hypercall

4.1.2 Access stack memory

To read all information from stack memory, as return addresses and frame pointers, we need to map the memory from the guest that runs the unikernel into the guest that runs the profiler, in our case Dom0.

For this, we must do an address resolution to transform the virtual address to the guest address. Even with paravirtualization, we need to manually walk page tables for this transformation.This happens in this code sequence, Listing 4.2, where we map each part of the memory.

```
void *guest_to_host(int domid, int vcpu, guest_word_t gaddr) {
  static mapped_page_t *map_head = NULL;
  mapped_page_t *map_iter;
  mapped_page_t *new_item;
  guest_word_t base = gaddr & PAGE_MASK;
  guest_word_t offset = gaddr & ~PAGE_MASK;

  map_iter = map_head;
  while (map_iter != NULL) {
    if (base == map_iter->base)
      return map_iter->buf + offset;
    // preserve last item in map_iter by jumping out
    if (map_iter->next == NULL)
      break;
    map_iter = map_iter->next;
  }

  // no matching page found, we need to map a new one.
  // At this pointer, map_iter conveniently points to the last item.
  new_item = malloc(sizeof(mapped_page_t));
  if (new_item == NULL) {
    fprintf(stderr, "failed to allocate memory for page struct.\n");
    return NULL;
  }
  new_item->base = base;
  xen_map_domu_page(domid, vcpu, base, &new_item->mfn, &new_item->buf);
  VERBOSE("mapping new page %#"PRIx64"->%p\n", new_item->base, new_item->
    buf);
  if (new_item->buf == NULL) {
    fprintf(stderr, "failed to allocate memory mapping page.\n");
    goto out_free;
  }

    .....

}
```

Listing 4.2: Address resolution

Of course, translation of addresses is done through the hypervisor using hypercalls. As you can see in Listing 4.3 we can use two different libraries to do this translation.

These libraries come with the same things, except that libxencall was introduced a bit more recently, in Xen version 4.8 and comes with an improvement in performance compared to libxc. In this way the latency of a stack trace can be reduced up to 3 times.[10]

```
void xen_map_domu_page(int domid, int vcpu, uint64_t addr, unsigned long *
    mfn, void **buf) {
  int err __maybe_unused = 0;
  DBG("mapping page for virt addr %"PRIx64"\n", addr);
#if defined(HYPERCALL_XENCALL)
  *mfn = xen_translate_foreign_address(domid, vcpu, addr);
  if (*mfn) {
    // This works since size is 1, so the array has size 1, so it's just a
    pointer to an int
    *buf = xenforeignmemory_map(fmemh, domid, PROT_READ, 1, (xen_pfn_t *)
    mfn, &err);
    if (err) {
      xenforeignmemory_unmap(fmemh, *buf, 1);
      *buf = 0;
    }
  }
  else {
    *buf = 0;
  }
#elif defined(HYPERCALL_LIBXC)
  *mfn = xc_translate_foreign_address(xc_handle, domid, vcpu, addr);
  DBG("addr = %"PRIx64", mfn = %lx\n", addr, *mfn);
  *buf = xc_map_foreign_range(xc_handle, domid, XC_PAGE_SIZE, PROT_READ, *
    mfn);
#endif
  DBG("virt addr %"PRIx64" has mfn %lx and was mapped to %p\n", addr, *mfn,
    *buf);
}
```

Listing 4.3: Translate address

4.1.3 Symbol table

After all, we will only have a lot of addresses. In order to be able to interpret them more easily, we will have to solve them to be able to figure out which function was called.

For this, we will need a symbol table that we can get quite easily from the ELF executable using the nm utility.[15] However, a necessary condition would be that the executable is not stripped, but Unikraft generates two executables, and one of them being for debugging is not stripped, so we will use it to find the symbol table and to be able to map the addresses found to a certain function.

As you can see in figure 8 this is an example of the output produced by nm for a Unikraft application.

```
00000000001ec6a0 d cclasses
00000000001be550 r _C_ctype_locale
00000000000ab22d t ceil
00000000000ac9d9 t ceilf
00000000000b38dc t ceill
00000000000b9499 t cfmakeraw
000000000008759a t chacha_encrypt_bytes
00000000000874d5 t chacha_ivsetup
00000000000870fd t chacha_keysetup
00000000001ec670 d chars.0
0000000000020678 t chdir
000000000001b733 t check_dir_empty
0000000000022d66 t chmod
0000000000023070 t chown
00000000000231c4 t chroot
000000000006e687 t _cleanup
000000000004b7f3 t cleanup_glue
000000000006e659 t _cleanup_r
000000000006ebb8 t clearerr
```

Figure 8: nm output

This output from nm must be parsed and together with the output generated by Uniprof, we can find out the connection between the addresses of the functions and their names. You can see this in Listing 4.4, where both files are open and parsed so that we finally have a trace stack with the names of the functions.

```
1
2     ....
3
4     while (std::getline(symbolfile, line)) {
5     convertor.str(line);
6     convertor >> std::hex >> addr >> type >> fname;
7     convertor.clear();
8     symbollist[addr] = fname;
9   }
10  symbolfile.close();
11
12  while (std::getline(tracefile, line)) {
13    if (line.empty())
14      std::cout << std::endl;
15    else if (line == "1")
16      std::cout << "1" << std::endl;
17    // comments; by convention, the header lines start with a comment sign
18    else if (line[0] == '#')
19      std::cout << line << std::endl;
20    else {
21      convertor.str(line);
22      convertor >> std::hex >> addr;
23      convertor.clear();
24      iter = symbollist.upper_bound(addr);
```

```
25      if (iter != symbollist.begin())
26        iter--;
27      if (addr == iter->first)
28        std::cout << iter->second.c_str() << std::endl;
29      else
30        std::cout << iter->second.c_str() << "+0x" << std::hex << addr -
    iter->first << std::endl;
31     }
32   }
33
34    ....
```

Listing 4.4: Symbol resolution

4.2 Automate profiling

Trying to generate Flame Graphs, I noticed that the process is quite difficult. It is divided into 6 steps, but they come with several configurations to do:

1. Start Unikraft application
2. Start Uniprof to get stack traces
3. Run nm to find symbol table
4. Use symbolize to map addresses
5. Run stack collapse to format stack trace for Flame Graph
6. Generate Flame Graph representation

You can see them better in the diagram 9.

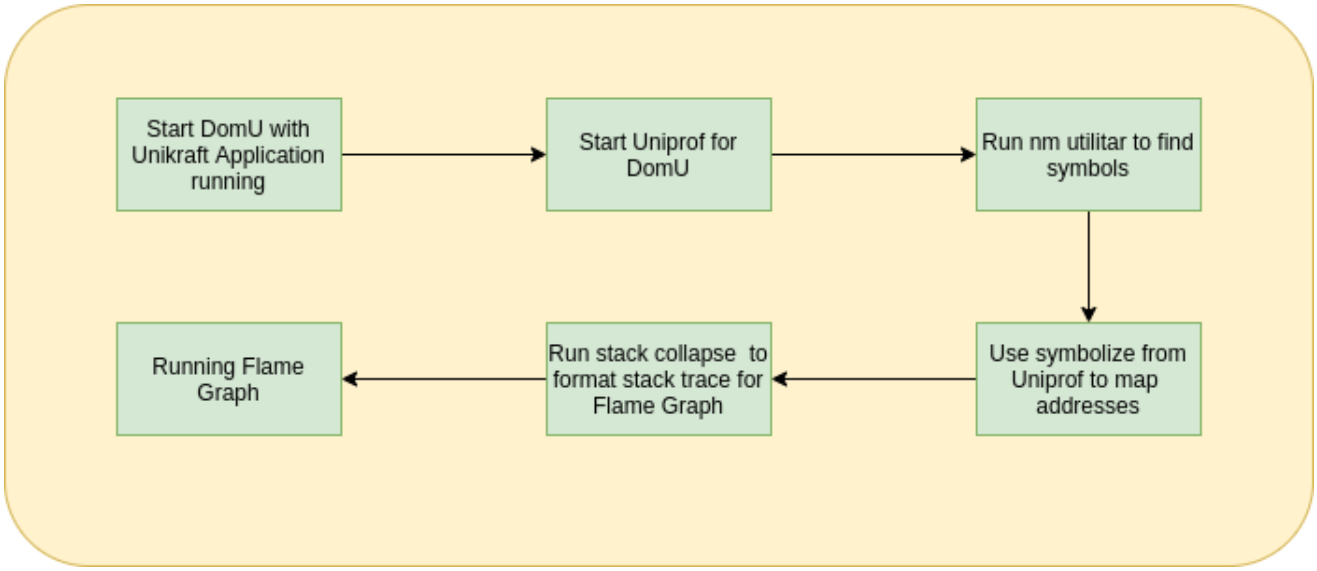

Figure 9: Stack Trace Automation

So I thought of creating a script that would automate the whole process. It takes various parameters and executes all the 6 steps mentioned above, in the end generating 5 files:

1. A scalable vector graphics file with the Flame Graph
2. Stack collapsed used to generate the Flame Graph
3. Stack trace with all addreses found using Uniprof
4. Resolved stack trace with all functions resolved
5. Symbol table

This is the list of all accepted parameters:

- -f | –frequency(Optional)
- -t | –time(Optional)
- -n | –domain(Mandatory)
- -c | –conf(Mandatory)
- -x | –exec(Mandatory)

Frequency is the number of stack traces per second, this is not a required parameter because by default it is set to 1. Time means how long Uniprof has to run and is expressed in seconds, as in the case of frequency this parameter is not mandatory because it is set to 1 if missing.

Domain is the name of the domain from which we will collect the stack traces. Conf parameter is the path to the configuration file that Xen needs to start the virtual machine in which the unikernel image will run. Exec is the path to the executable file of the application we want to run. All these parameters are mandatory because we need to know which application we want to start and which domain we want to inspect.

You can see the script in the Appendix Listing 7.1, this is a version only with the main parts except for any checks of the parameters.

4.3 Challenges

4.3.1 Configuration file

Working on this project, I also encountered some challenges, the first would be adapting to the Xen environment. In order to start the virtual machines, the Xen hypervisor needs certain configuration files that I had to get used to.[16]

Uniprof has a configuration file to make sure you have the Xen environment installed and its utilities that come in a different package. In the first attempts to run this I encountered a problem related to the missing of xenctrl.h file, where certain virtual machine control hypercalls are defined, from /usr/include. I solved this by moving this file from Xen tools to the appropriate path.

4.3.2 Dynamic link loader

For Uniprof to run, LD_LIBRARY_PATH and PATH local variables had to be updated with some paths from the Xen tools package. This was not documented and I had to find out all these ways manually. To make it easier to port to other systems, I created a script to update all the paths faster. You can see this in Listing 4.5.

```
1 #!/bin/bash
2
3 export PATH="/root/armtoolchain/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-
    linux-gnu/bin:/root/armtoolchain/qemu-4.0.0/bin:/root/armtoolchain/gcc-
    linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/ibexec/gcc/aarch64-linux-
    gnu/7.5.0:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/go/bin
    "
4
5 export LD_LIBRARY_PATH="/root/gmocanu/xen-4.8/xen/tools/libs/ctrl:
    $LD_LIBRARY_PATH"
6 export LD_LIBRARY_PATH="/root/gmocanu/xen-4.8/xen/tools/libs/toollog:
    $LD_LIBRARY_PATH"
7 export LD_LIBRARY_PATH="/root/gmocanu/xen-4.8/xen/tools/libs/util:
    $LD_LIBRARY_PATH"
8 export LD_LIBRARY_PATH="/root/gmocanu/xen-4.8/xen/dist/install/usr/local/
    lib:$LD_LIBRARY_PATH"
```

Listing 4.5: Dynamic link loader path

4.3.3 Xen version

Another problem I came across was that although I was working on a machine that had a Xen hypervisor version 4.8 installed, when I installed Xen tools, it was version 4.15. This caused some errors when I tried to run Uniprof with any Unikraft application image, it did not extract any stack traces because no hypercall was executed successfully. For this I had to use the same version of Xen tools.

If you are using Xen version 4.8 you will have to downgrade to GNU Compiler Collections to 6.0 because with any newer version of GNU Compiler Collections the compilation of Xen tools will fail due to some constraints that have been introduced. I also tested with the latest version of Xen 4.15 and everything seems to work because the API for hypercalls has not changed much.

5 RESULTS AND EVALUATION

5.1 Custom application

In order to test that I get proper results I made a simple application in which I call several functions a different number of times and see if this is reflected in Flame Graph. You can see that application in Listing 7.2.

There you can observe that I call function_1 100000 times, function_2 300000 times and function_3 600000 times. To generate the Flame Graph I used the script mentioned above. 10

```
root@c415:~/gmocanu/uniprof# bash generateFlameGraph.sh  -f 100 -t 240 -n hello-world -c
~/gmocanu/apps/app-helloworld/hello-world.cfg -x ~/gmocanu/apps/app-helloworld/build/app-
helloworld_x
en-x86_64.dbg
Frequency - 100
Time - 240
Domain - hello-world
Path to config - /root/gmocanu/apps/app-helloworld/hello-world.cfg
Path to exec - /root/gmocanu/apps/app-helloworld/build/app-helloworld_xen-x86_64.dbg
---------------------

Starting VM...

Finding domain id...
Domain id = 631

Running uniprof...
nohup: ignoring input and redirecting stderr to stdout

Running nm to find symbols...

Running symbolize to map addresses...
Running stack collapse to format stack trace for Flame Graph...

Running Flame Graph...
```

Figure 10: Generate Flame Graph

After the execution of the script was finished, I obtained 5 files.

A file with the termination .syms, here we have the output generated by the nm utility and we will find each function and its corresponding address. You can see the example in Figure 11.

```
000000000000d392 t xs_debug_msg
000000000000d43b t xs_read_integer
000000000000d4c2 t xs_scanf
000000000000d5f4 t xs_printf
000000000000d710 t xs_get_self_id
000000000000d79f t function_3
000000000000d7d0 t function_2
000000000000d801 t function_1
000000000000d833 t main
000000000000d8fb t uk_printd
000000000000d957 t uk_alloc_addmem
000000000000d9de t ukplat_memregion_find_next
000000000000da50 t main_thread_func
000000000000dc42 t ukplat_entry_argp
000000000000dcc9 t ukplat_entry
000000000000dfb0 t uk_version
000000000000dfc6 t print_banner
000000000000e099 t _print_timestamp
000000000000e171 t vprint
```

Figure 11: Nm result

```
0x240a
0x74cb
0x7692
0x73d5
0x10b42
0x10c5a
0x10cfb
0xd82e
0xd877
0xdc20
1

0x240a
0x74cb
0x7692
0x73d5
0x10b42
0x10c5a
0x10cfb
0xd82e
0xd877
0xdc20
1
```

Figure 12: Raw Stack Trace

```
hypercall_page+0x40a
notify_remote_via_evtchn+0x22
hv_console_output+0x178
ukplat_coutk+0x4b
vfprintf+0x74
vprintf+0x2a
printf+0x9f
function_1+0x2d
main+0x44
main_thread_func+0x1d0
1

hypercall_page+0x40a
notify_remote_via_evtchn+0x22
hv_console_output+0x178
ukplat_coutk+0x4b
vfprintf+0x74
vprintf+0x2a
printf+0x9f
function_1+0x2d
main+0x44
main_thread_func+0x1d0
1
```

Figure 13: Resolved Stack Trace

Two files with similar names stacktrace and stacktrace.syms, these two files represent the output generated by Uniprof. In the first file we only have the stack traces,with a lot of addresses, and in the second file these addresses are solved and we can view them much better. You can observe the difference between them in figure 12 and 13.

In order to generate Flame Graphs, a certain format of the files in which the stack traces are found is needed, unfortunately the output generated by Uniprof does not respect these conditions. Therefore before calling flamegraph.pl (Perl file that converts stack traces into a Flame Graph) we will need to call stackcollapse.pl, a Perl script that transforms the output generated by Uniprof into a file with a structure which we can use to generate a Flame Graph. You can see this file in figure 14.

```
main;function_1;printf;vprintf;vfprintf;ukplat_coutk;hv_console_output;notify_remote_via_evtchn;hypercall_page 2811
main;function_2;printf;vprintf;vfprintf;ukplat_coutk;hv_console_output 1
main;function_2;printf;vprintf;vfprintf;ukplat_coutk;hv_console_output;notify_remote_via_evtchn;hypercall_page 6403
main;function_3;printf;vprintf;vfprintf;ukplat_coutk;hv_console_output 7
main;function_3;printf;vprintf;vfprintf;ukplat_coutk;hv_console_output;hypervisor_callback2;do_hypervisor_callback;do
main;function_3;printf;vprintf;vfprintf;ukplat_coutk;hv_console_output;notify_remote_via_evtchn;hypercall_page 14777
```

Figure 14: Stack Collapse

This is also a much easier to read file than the initial one, on the left we have the call chain starting from the first to the last function, and on the right we have the number of times stack trace has been identified.

And the best way to visualize any bottleneck through stack traces is represented by the Flame Graph that you can see in figure 15.

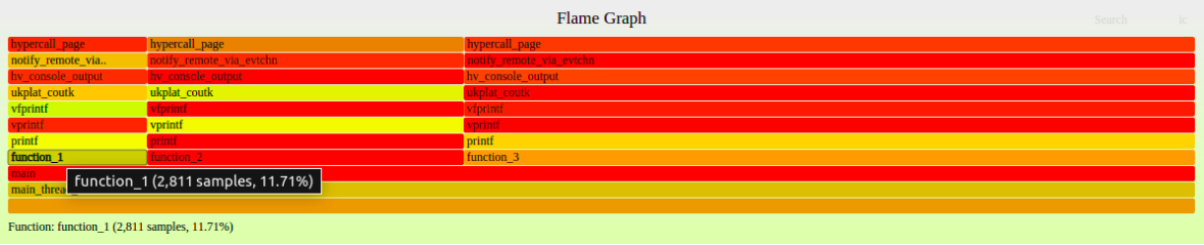

Figure 15: Flame Graph Custom Application

In order to be able to read this Flame Graph correctly we should look from the bottom up to be able to see the stack traces. Colors are not important, they are just a way to differentiate functions. If vertically we can see the call chain of the functions, horizontally we can see how many times they were called, so if a function has a larger width it was called more times.

Flame Graph also has in the upper right an option for searching through regex for certain functions, and if we open a certain function we can see how many times it was present in a stack trace. To check the output we can verify the percentage for each function tested, function_1, function_2 and function_3.

For example, function_1 appears 2811 times, function_2 6404 times, and function_3 14785 times. To calculate the percentage we will have to do the following calculations:

- function_1 - $\frac{2811}{2811+6404+14785} * 100 = \frac{2811}{24000} * 100 = 11.71\ \%$
- function_2 - $\frac{6404}{2811+6404+14785} * 100 = \frac{6404}{24000} * 100 = 26.68\ \%$
- function_3 - $\frac{14785}{2811+6404+14785} * 100 = \frac{14785}{24000} * 100 = 61.60\ \%$

All these results appear directly in Flame Graph and you can see them in figure 15 at the bottom. If we look at the program in Appendix, listing 7.2, we will see that function_1 is called 100000 times, function_2 300000 times, and function_3 600000 times, so we should have expected a 10% result for function_1, 30% for function_2 and 60% for function_3.

Due to the fact that Uniprof uses sampling as a method to achieve these results, and not the instrumentation, it is normal that results are not perfect and have some variations, but we can see that these results are not far from true with a relative error of less than 10 %.

5.2 Custom application 2

Compared to the first application I tested, this one it has a larger call chain. I wanted to check that the Flame Graph is made correctly on several levels. You can see that application in Listing 7.3.

```
root@c415:~/gmocanu/uniprof# bash generateFlameGraph.sh  -f 5 -t 500 -n hello-world -c
~/gmocanu/apps/app-test-2/app-test-2.cfg -x ~/gmocanu/apps/app-test-2/build/app-hellowo
rld_xen-x86_64.d
bg


Frequency - 5
Time - 500
Domain - hello-world
Path to config - /root/gmocanu/apps/app-test-2/app-test-2.cfg
Path to exec - /root/gmocanu/apps/app-test-2/build/app-helloworld_xen-x86_64.dbg
---------------------

Starting VM...

Finding domain id...
Domain id = 645

Running uniprof...
nohup: ignoring input and redirecting stderr to stdout

Running nm to find symbols...

Running symbolize to map addresses...

Running stack collapse to format stack trace for Flame Graph...

Running Flame Graph...
```

Figure 16: Generate Flame Graph

I ran Uniprof for about 500 seconds because the application runs longer than the first one and at a lower frequency to see how it behaves in this way. Finally I obtained the Flame Graph that you can see in figure 17.

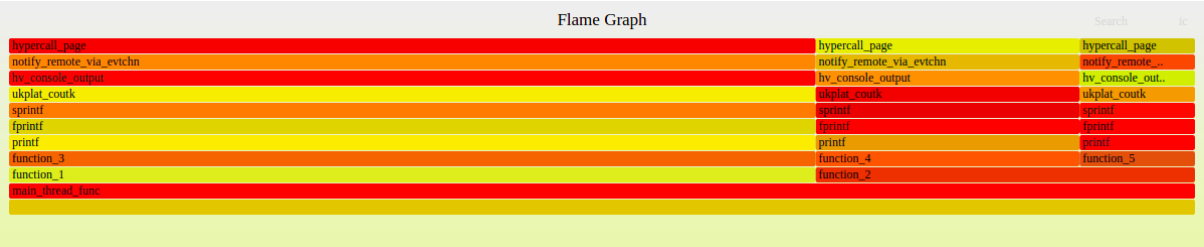

Figure 17: Flame Graph Custom Application

First time we can look at the initial level and we will notice that function_1 appears 993 times while function_2 appears 467 times. If we look at the code we notice that function_1 was called 200000 while function_2 was called 100000 times, which would lead to a ratio of 66.6% for the first function and 33.3% for the second function.

In the Flame Graph obtained through Uniprof we obtained the following percentages:

- function_1 - $\frac{993}{993+467} * 100 = \frac{993}{1460} * 100 = 68.01$ %
- function_2 - $\frac{467}{993+467} * 100 = \frac{467}{1460} * 100 = 31.98$ %

If we look at the second level we will see that function_1 calls function_3 and function_2 calls function_4 and function_5. We don't have to check anything for function_3 because it is called every time function_1 is invoked, but we can see if the call chain for function_2 works properly.

In the source code function_4 was called 200000 while function_5 was called 100000 times, which again would lead to a ratio of 66.6% for function_4 and 33.3.% for function_5.

In my results I obtained:

- function_1 - $\frac{325}{325+142} * 100 = \frac{325}{467} * 100 = 69.59$ %
- function_2 - $\frac{142}{325+142} * 100 = \frac{142}{467} * 100 = 30.41$ %

The results are very similar to what I called in the code.

5.3 Redis

I tried to do some tests on other more complex applications that have support in Unikraft, one of them being Redis. To start the image I needed a file for Xen configuration. You can see it in figure 18.

```
name            = 'redis'
vcpus           = '1'
memory          = '4'
kernel          = '/root/gmocanu/apps/app-redis/build/app-redis_xen-x86_64.dbg'
memory          = 512
vif             = ['ip="10.0.0.1"']
p9              = ["tag=rootfs,security_model=none,path=/root/gmocanu/apps/app-nginx"]
```

Figure 18: Redis Xen Configuration

Beside specifying memory and image name, I had to mention the path to the executable file that starts the application. As you can see we used the debug executable because it is not stripped and we can solve its symbols. Other parameters required for the Redis application were the virtual interface for communication with the virtual machine where it will run, of course, it was necessary to manually create a bridge, and p9 parameter to be able to have a virtual file system where we can map a test file for Redis.

In the virtual file system part, I discovered a bug in the Unikraft application that does not allow mounting a 9pfs file system on the Xen platform. Because of this, I was unable to map a test file to Redis, but still, I was able to run Uniprof on a Redis instance.

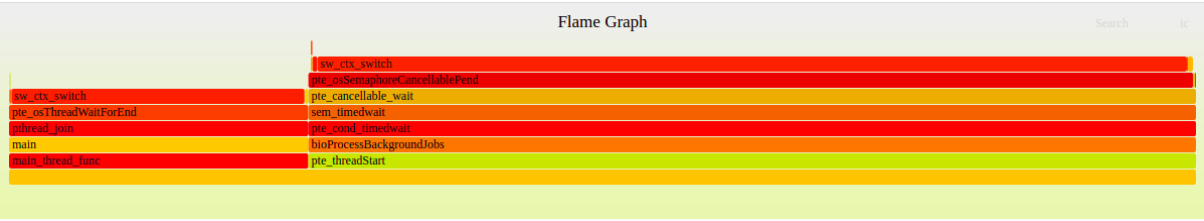

Figure 19: Redis Flame Graph

You can see the Flame Graph from Redis application generated in figure 19. It can be observed some functions for threads, waits and semaphores are running. This confirms that Redis is not running anything but is just waiting for future functions to be executed.

6 CONCLUSION AND FUTURE WORK

6.1 Objective

The objective of this project was to integrate Unikraft with Uniprof in order to identify certain bottlenecks that may occur in applications running in Unikraft. The representation through Flame Graph of the stack traces obtained from Uniprof was another priority because in this way the stack traces are easier to understand and everyone can identify problems easily.

6.2 Current status

Currently, I have managed to integrate Uniprof with Unikraft and automate the entire process of creating a Flame Graph. I did some tests on custom applications with simpler or more complicated call chains to test that Uniprof performs these metrics well. In order to check, I compared the percentage of calls of functions obtained from Uniprof with the one that should have occurred if I had used an instrumentation method.

My environment was running Xen version 4.8 because this was the environment on which Uniprof was developed, but I tried to run it on the latest version of Xen, 4.15 and everything went the same way because all the hypercalls that Uniprof uses have not been changed between these two versions.

I put everything in a repository on Github, and now if someone needs these metrics they should just clone code and replace the application they want to analyze.

6.3 Planned work

After all this, we planned to start a separate project for a KVM type hypervisor that would behave the same as Uniprof. Because we are talking about another hypervisor, there will definitely be other hypercalls for that. The first step will be to adapt to the KVM hypervisor and the API it exposes to interact with memory. Then we will be able to decide if we can integrate this in Uniprof or it will be a separate project.

7 APPENDIX A

7.1 Generate Flame Graph automation

```
#!/bin/sh

# Load $PATH and $LD_LIBRARY_PATH
source export.sh

# Global variables
script_name=$0
symbolize="symbolize"
uniprof='uniprof'
stackcollapse='stackcollapse.pl'
flamegraph='flamegraph.pl'

# Function for running
function run
{
    parse_arguments "$@"

  echo "Starting VM..."
  nohup xl create $path_conf > /dev/null 2>&1
  echo ""

  echo "Finding domain id..."
  dom_id=$(xl list | grep "$domain" | awk '{print $2}')
  echo "Domain id = $dom_id"
  echo ""

  echo "Running uniprof..."
  nohup "$path_to_uniprof$uniprof" -F $frequency -T $time - $dom_id > "
    $domain-stacktrace" 2> /dev/null
  echo ""

  echo "Running nm to find symbols..."
  nohup nm -n $path_exec > "$domain.syms" 2> /dev/null
  echo ""

  echo "Running symbolize to map addresses..."
  nohup "$path_to_uniprof$symbolize" "$domain.syms" "$domain-stacktrace" >
    "$domain-stacktrace.syms" 2> /dev/null
  echo ""

```

```
  echo "Running stack collapse to format stack trace for Flame Graph..."
  nohup "$path_to_flamegraph$stackcollapse" "$domain-stacktrace.syms" > "
    $domain-stackcollapse" 2> /dev/null
  echo ""

  echo "Running Flame Graph..."
  nohup "$path_to_flamegraph$flamegraph" "$domain-stackcollapse" > "$domain
    -flamegraph.svg" 2> /dev/null
  echo ""
}

# Function for parsing arguments
function parse_arguments
{

# Parsing arguments
while [ "$1" != "" ]; do
    case "$1" in
        -f | --frequency )          frequency="$2";          shift;;
        -t | --time )               time="$2";               shift;;
        -n | --domain )             domain="$2";             shift;;
        -c | --conf )               path_conf="$2";          shift;;
    -x | --exec )               path_exec="$2";          shift;;
    -h | --help )               usage;                   exit;;
        * )                         invalid $1;              exit;; #
    Invalid option
    esac
    shift
done
}

run "$@"
```

Listing 7.1: Generate Flame Graph

7.2 Custom Application

```
#include <stdio.h>
#ifdef __Unikraft__
#include <uk/config.h>
#endif /* __Unikraft__ */

int function_3(int a,int b) {
    int ret = a/b;
    printf("aux = %d\n",ret);
    return ret;
}

int function_2(int a,int b){
```

```
13      int ret = a*b;
14      printf("aux = %d\n",ret);
15      return ret;
16  }
17
18  int function_1(int a,int b)
19  {
20      int ret = a+b;
21      printf("aux = %d\n",ret);
22      return ret;
23  }
24  int main(int argc, char *argv[])
25  {
26      printf("Start\n");
27      int i = 0;
28      int aux = 0;
29      for(i = 0; i < 100000; i++){
30          aux = function_1(aux, 1);
31      }
32
33      aux = 1;
34      for(i = 0; i < 300000; i++){
35          aux = function_2(aux, 2);
36      }
37
38      aux = 1000000;
39      for(i = 0; i < 600000; i++){
40          aux = function_3(aux, 1);
41      }
42      printf("End\n");
43  }
```

Listing 7.2: Custom Application

7.3 Custom Application 2

```
1  #include <stdio.h>
2  #ifdef __Unikraft__
3  #include <uk/config.h>
4  #endif /* __Unikraft__ */
5
6  int function_3(int a)
7  {
8    int i;
9    int ret = a;
10   for(i = 0;i < 100000; i++){
11     printf("aux = %d\n", ret);
12   }
13   return ret;
```

```
}

int function_4(int a)
{
  int i;
  int ret = a;
  for(i = 0;i < 200000; i++){
    printf("aux = %d\n", ret);
  }
  return ret;
}

int function_5(int a, int b)
{
  int i;
  int ret = a * b;
  for(i = 0;i < 100000; i++){
    printf("aux = %d\n", ret);
  }
  return ret;
}

int function_2(int a,int b)
{
  int i = 0;
  int ret = 0;
  ret = function_4(b);
  ret = function_5(a,b);
  printf("aux = %d\n",ret);
  return ret;
}

int function_1(int a,int b)
{

  int ret = 0;
  ret = function_3(a);
  printf("aux = %d\n",ret);
  return ret;
}

int main(int argc, char *argv[])
{
  printf("Start\n");
  int i = 0;
  int aux = 0;
  for(i = 0; i < 200000; i++){
    aux = function_1(aux, 1);
  }

```

```
64    aux = 1;
65    for(i = 0; i < 100000; i++){
66      aux = function_2(aux, 2);
67    }
68
69    printf("End\n");
70  }
```

Listing 7.3: Custom Application 2

BIBLIOGRAPHY

[1] A. Kohgadai, "Container adoption trends of 2020." https://www.stackrox.com/post/2020/03/6-container-adoption-trends-of-2020. Last accessed: 05 June 2021.

[2] "Docker security vulnerabilities." https://www.cvedetails.com/vulnerability-list/vendor_id-13534/product_id-28125/Docker-Docker.html. Last accessed: 05 June 2021.

[3] "Unikernel." http://unikernel.org/. Last accessed: 05 June 2021.

[4] I. Eyberg, "Introduction to unikernels." https://nordicapis.com/introduction-to-unikernels/. Last accessed: 05 June 2021.

[5] S. Kuenzer, "Unikraft." https://xenproject.org/developers/teams/unikraft/. Last accessed: 05 June 2021.

[6] G. Gain, "Unikernels and toolchains." https://unicore-project.eu/unikernels-and-toolchains/. Last accessed: 21 June 2021.

[7] F. Schmidt, "Uniprof: A unikernel stack profiler." http://sysml.neclab.eu/projects/uniprof/uniprof-poster.pdf, 2017. Last accessed: 05 June 2021.

[8] B. Gregg, "Flame graphs." http://www.brendangregg.com/flamegraphs.html. Last accessed: 05 June 2021.

[9] B. Gregg, *Systems Performance: Enterprise and the Cloud.* 2020.

[10] P. Brown, "uniprof: Transparent unikernel for performance profiling and debugging." https://www.linux.com/training-tutorials/uniprof-transparent-unikernel-performance-profiling-and-debugging-2/. Last accessed: 21 June 2021.

[11] H. Fayyad, L. Perneel, and M. Timmerman, "Xen." https://www.researchgate.net/publication/258327364_Full_and_Para-Virtualization_with_Xen_A_Performance_Comparison. Last accessed: 15 June 2021.

[12] "Domain." https://wiki.xenproject.org/wiki/Domain. Last accessed: 15 June 2021.

[13] D. Barrett and G. Kipper, "Virtualization and forensics." https://www.sciencedirect.com/book/9781597495578/virtualization-and-forensics. Last accessed: 16 June 2021.

[14] "Hypercall interfaces." http://xenbits.xen.org/docs/unstable/hypercall/x86_64/include,public,domctl.h.html#Struct_xen_domctl_ext_vcpucontext. Last accessed: 16 June 2021.

[15] "nm - list symbols from object files." https://linux.die.net/man/1/nm. Last accessed: 20 June 2021.

[16] "xl domain configuration file syntax." https://xenbits.xen.org/docs/unstable/man/xl.cfg.5.html. Last accessed: 21 June 2021.