uniprof: A Unikernel Stack Profiler

Florian Schmidt
NEC Laboratories Europe
florian.schmidt@neclab.eu

ABSTRACT

Unikernels are increasingly gaining traction in real-world deployments, especially for NFV and microservices, where their low footprint and high performance are especially beneficial. However, they still suffer from a lack of tools to support developers. uniprof is a stack profiler that supports Xen unikernels on x86 and ARM and does not requires any code changes or instrumentation. Its high speed and low overhead (0.1% at 100 samples/s) makes it usable even in production environments, allowing the collection of realistic and highly credible data.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Software maintenance tools**;

KEYWORDS

call stack profiling, introspection, Xen, unikernels

ACM Reference format:
Florian Schmidt. 2017. uniprof: A Unikernel Stack Profiler. In *Proceedings of SIGCOMM Posters and Demos '17, Los Angeles, CA, USA, August 22–24, 2017,* 3 pages.
DOI: 10.1145/3123878.3131976

1 INTRODUCTION

Unikernels [6, 7], long a topic in the research community, are starting to receive more and more interest in real-world deployments. Compared to full-featured virtual machines running commodity operating systems, they provide extremely small VMs, which is especially beneficial for NFV [8, 9] and microservices [4]. For example, a simple web server can have a binary size as low as 260 kB and memory requirements as low as 1.2 MB [4]. This facilitates large-scale consolidation, supporting use cases such as VNF chaining, while also keeping the strong isolation guarantees of virtualized systems, providing secure multi-tenancy.

On top of that, unikernels generally provide much higher performance than standard virtual machines. This is chiefly due to the fact that the application and the operating system are compiled into a single binary and share the same address space, which obviates time-consuming and performance-constraining system calls, memory translations and context switches. While this removes the innate security provided by system calls, this is not a problem for unikernels: they only run a single application, so gaining control of the "operating system" part of the unikernel provides no exploitable benefit to the application; for security between different virtual machines, unikernels instead rely on the isolation mechanisms of the underlying virtual machine monitor (hypervisor).

However, the adoption of unikernels as a new and highly-performant technology is hampered by the need to develop new unikernel-compatible software, or to adopt existing one. This is exacerbated by the lack of tools to support developers. Contrary to popular belief, debugging a unikernel is in fact simpler than debugging a standard operating system: Since the application and OS are linked into a single binary, debuggers can be used on the running unikernel to debug both application and OS code at the same time. While this requires support from the hypervisor, `gdb`, to give a popular example, contains support for both Xen and KVM virtual machine debugging. Nevertheless, especially in the area of profiling, unikernels still lack the rich toolsets that full operating systems such as Linux or BSD provide.

The goal of uniprof is to provide such a tool. uniprof is a stack profiler that provides the following functionality:

- uniprof profiles a running unikernel from the outside, requiring no changes to the code;
- it introduces only minimal overhead, allowing profiling of unikernels in production environments;
- it supports Xen [1] unikernels, running both on x86 and ARM, both with and without frame pointers;
- its output can be used to produce insightful visualizations, aiding the developer in identifying bottlenecks.

uniprof is available as open source at https://github.com/cnplab/uniprof/.

2 STACK PROFILING

Stack profiling is done by collecting a number of stack traces. Each stack trace gives a snapshot of the currently run functions, and the functions that lead to the current function being called. This is generally done by using the instruction pointer (IP) and frame pointer (FP) registers. The approach is shown in Figure 1. The IP gives the first address of the trace, to the current instruction, while the FP points to the bottom of the current function's stack frame. As each function is called, registers are pushed onto the stack before the function starts its execution. The order is defined by calling conventions, but in a typical one, the FP is the first register being pushed, producing a linked list of FPs down to the initial function. The return address is also pushed onto the stack in a fixed location, hence by traversing the FPs 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 (and offsets into the function). Note that one advantage of a unikernel is that, due to the unified address space, all function locations are static, and it is much simpler to create a symbol table than for normal operating systems with applications running of them: the unstripped binary

SIGCOMM Posters and Demos '17, Los Angeles, CA, USA
 978-1-4503-5057-0/17/08…$15.00
DOI: 10.1145/3123878.3131976

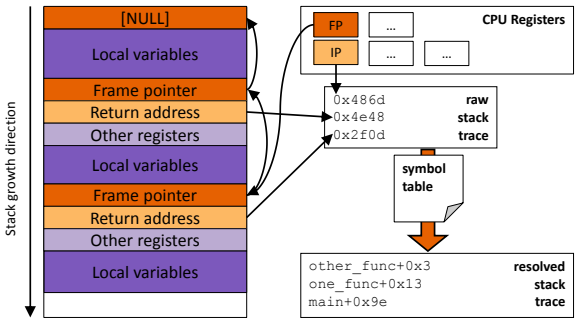

Figure 1: A stack walk. The IP provides the current execution address. The FPs point to the start of the frame, creating a linked list of frames. The return addresses provide the other stack trace entries.

of the unikernel will contain everything that is needed for address resolution.[1]

By regularly sampling the stack in this way, we get important information not only about which functions consume a large percentage of the unikernel's execution time, but also which call paths lead to those heavy hitters.

3 UNIPROF

One of the largest advantages of stack profilers is that investigated code does not need any special annotations or debug support. The stack trace is done from the outside, with uniprof stopping the unikernel momentarily – to prevent race conditions from the stack changing while it is being traversed – walking the stack, and unpausing. One of the most important factors of stack profilers is their performance. This does not only matter for convenience reasons, but also influences the fidelity of their results. A profiler that takes too long to walk the stack will keep the application paused for longer times, not only slowing it down, but also potentially changing its behavior (e.g., missing timeouts). Hence, performance is the core concern of uniprof.

A profiling cycle starts with instructing Xen to pause the VM. uniprof then receives the content of the virtual machine's "virtual CPU" registers via `getvcpucontext`. After doing the actual stack walk, the VM is unpaused again. Figure 2 shows the performance of uniprof for two different supported machines, an Intel Xeon E5 CPU at 3.7 GHz and an ARM Cortex A7 at 1 GHz. The time the stack walk itself takes depends on the depth of the stack. These tests were run on a unikernel that kept a constant stack depth of 10, not an unusually low depth for unikernels, which tend to have fewer indirection steps due to their unified binary setup. Stack walks finish in 12 µs on the x86 machine, and in 51 µs on the ARM one, roughly correlating with the speed difference of the machines. Thus, running about 1000 samples/s produces an overhead of only 1% for x86 and 5% for ARM, and only roughly a tenth of that at 100 samples/s.[2] This means that uniprof's impact on the performance

[1] *Virtualized* addresses seen by the VM due to its virtualization still need to be translated, but this is independent of the symbol table and is done by uniprof on-the-fly.

[2] Uneven or prime sample rates are a standard practice to reduce the risk of sampling in lock-step with regular events in the profiled code, which would skew the results.

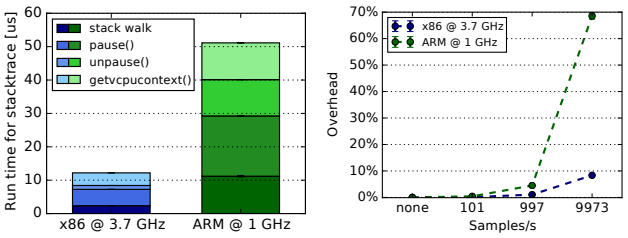

Figure 2: uniprof does a single stack walk in a few microseconds. Thus even at nearly 1000 samples/s, the overhead is 1% for a fast x86 machine and below 5% for a slower ARM.

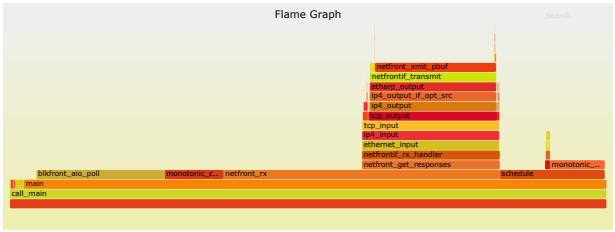

Figure 3: Stack profiling visualization as flame graph [3] shows long-running and/or repeatedly-called functions and identifies potential bottlenecks.

of the unikernel under test is very low, even at high sampling rates, allowing it to be used even on production systems.

One further advantage of uniprof is that it does not require the availability of frame pointers. While this is the typical approach to creating stack traces, some highly optimized software repurposes the FP register as another general-purpose register to increase performance. To profile such code, uniprof uses libunwind [5], which in turn uses DWARF [2] call frame information, as produced by compilers such as `gcc`. While not changing the behavior of the binary itself, it contains information that allows for every point in the code to reconstruct the size of the frame at that point. Iteratively, the beginning of the current frame can be found, the return address be extracted, and the frame size for that address be calculated, until the beginning of the stack is reached. A libunwind version patched for use with Xen is available at https://github.com/cnplab/libunwind.

Finally, the output of uniprof can be fed into visualization tools. It is set up to directly interface with flame graphs [3], a popular tool to identify performance bottlenecks in software. Figure 3 shows an example for a unikernel doing network I/O. In this example, `blkfront_aio_poll`, `netfornt_rx`, and `netfront_xmit_pbuf` are the main bottlenecks.

ACKNOWLEDGMENTS

This paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 671566 ("Superfluidity"). This paper reflects only the author's views and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[2] Free Standards Group–DWARF Debugging Information Format Workgroup. DWARF Debugging Format, Version 3. http://dwarfstd.org/doc/Dwarf3.pdf. (Dec. 2005).

[3] Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* 59, 6 (May 2016), 48–57. https://doi.org/10.1145/2909476

[4] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 15–29. https://doi.org/10.1145/3050748.3050757

[5] The libunwind project. http://www.nongnu.org/libunwind/. ([n. d.]).

[6] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGPLAN Not.* 48, 4 (March 2013), 461–472. https://doi.org/10.1145/2499368.2451167

[7] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: Rise of the Virtual Library Operating System. *Commun. ACM* 57, 1 (Jan. 2014), 61–69. https://doi.org/10.1145/2541883.2541895

[8] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[9] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMIddlebox '16)*. ACM, New York, NY, USA, 44–49. http://doi.acm.org/2940147.2940149