COMP9242 Final Exam

12 November, 2013

Paper 1

1 Summary

Most traditional (monolithic) kernels run device drivers in kernel space. As such, if—for example—a device driver crashes or misbehaves, it is possible for it to bring the entire system down, as there is no isolation between components. *VirtuOS* separates out whole "vertical slices" of OS services (eg storage or network services, compared to say, isolation between the user, the block device driver *and* underyling device driver) to run in virtualised environments (service domains).

The authors make three main claims regarding the novelness of their paper:

- virtualization into service domains, (each directly handling specific system calls)
- "exception-less" system calls between service domains
- and that this provides resilience needed of service domains

The authors have implemented a prototype atop of Linux and Xen, which they discuss and provide performance results for. They also demonstrate the ability for their implementation to handle termination and restarts of these service domains in case of a failure, and attempt to show the performance benefits that their "exception-less" system provides.

2 Evaluation

2.1 Strengths

- Concept is sound - in Xen's case, lots of vulnerablilties in Dom0 caused by device drivers, so this is a good thing to address.
  - Reliablilty too is also important (appears to be more the issue of concern for the authors of this paper)

- Implementation of "exceptionless" system calls between virtual machines certainly appears to be a novel contribution.
- Discussion of storage domain performance is good.
- Failure recovery is transparent, demonstratable and works as intended.

2.2 Weaknesses

- Fails to spell out that drivers can run unmodified (I understand it's a VM, but surely this is a key feature)
- Does not explain very well how this idea is different from a microkernel-based operating system (using service domains as servers, rather than say individual servers for each driver),
  - paper does note that they are "in the same design space", but goes no further other than mentioning them in passing in *Introduction*, while in *Related Works*, only provides a single sentence – "differ from VirtuOS in the methods used for communication and protection"
- Does not explain how it is different from Xen isolated driver domains.
- Does not explain well the motivation between service domains and whole subservices.
  - One reason might because you get higher performance since you're not consistently doing context switches between subcomponents of a particular service domain
  - However, microkernels let you get away without prescribing this design decision on you
  - Perhaps it would've been a better idea to compare service domain implementation vs multiserver implementation?
- No detail on the consequences of "exception-less" system calls are on cache and TLB pollution (a major issue).
- Paper was generally difficult to read – too much irrelevant wafting?
  - The term "service domains" is defined in an odd spot, and I only noticed it after having a read through the second time. It is defined towards the end of the introduction, despite the fact that the term is used several times previous to that. Moreover, it appears in the abstract, and as such it should be defined there.
  - On the plus side, all the details were definitely there! ;)

- Arguable that PVHVM is not explained properly – I certainly didn't understand what it was from the paper (apart from working out what the acronym stood for). Further background may have been useful, although perhaps only an additional sentence or two is required (enough background on paravirtualisation is already provided).
- Would've hoped to see similar performance to FlexSC, instead of huge performance overheads (see next section)
- Failure recovery does not store state, so quite a performance hit on restarting service domains while waiting for it to start. Would be nice if the authors mentioned how they might address this issue.

3 Criticism

3.1 Novelty

The Xen project is currently in the process of disagregating Dom0 to different service domains. This was based on earlier effort called *Xoar*[1], which was actually presented at SOSP11. They managed to achieve similar performance results compared to the baseline version of Xen available at the time (1-2% throughput overhead; annoyingly CPU overhead is not discussed). The overhead provided by this implementation is *much* higher (discussed later).

The Qubes project[2] also appears to follow a similar concept - ie separate out network and storage subsystems into separate virtual machines for security – however they leave Dom0 to act at the graphics and input subsystems (although the possibility of separating this out is briefly discussed). Their paper was released in 2010.

Additionally, although the *Background* section claims that "VirtuOS does not use a split driver model," in section 3.3.2, the authors state clearly that they are using front-end and back-end drivers in their implementation, nonetheless. As such, I see almost no difference between VirtuOS and Xen's isolated driver domains, apart from the communiation method between virtual machines.

In my view the authors claims regarding novelty have been significantly dampened. The only novel contribution in my opinion is using "exception-less" system calls to communicate between virtual machines rather than between an application and the kernel directly, and its performance results.

3.2 Effort

Before we go into implementation details, it's worth pointing out Table 1. It says that a total of 20K lines are added, made up of the "front-end and back-end" drivers, the C library stack and libaio (which is not mentioned in the text at all), and the kernel and virtualisation environment. Firstly, mixing new and modified code into the same table is confusing, and gives little indication into the amount of effort required. For example, was the number of lines changed

in "uClibc + NPTL" because 11K lines were simply added from a code tarball? Or were 11K lines changed in order for the implementation to run correctly?

The text mentions that "the relatively small number of changes needed to the Linux kernel shows that our ... approach enabled vertical slicing with comparably little effort". Said changes to the Linux kernel–according to the table–only accounted for 5% of the total codebase change! Even if we assume away the uClibc entry, the back- and front-end driver code is still 3x the number of lines claimed in the body text. Picking out the number of code changes of one component while completely ignoring other *critical* components in the implementation is not at all represenative of the total amount of work required; contrary to what is suggested. Moreover, it is not possible to detemine if the number of lines of the `libc-sclib` component (see Figure 1) is included or not.

3.3 Insights

These two final sections discuss primarily benchmarking crimes. I would go so far as to argue that some of them are attempts to obscure the poor performance figures from the authors' implementation. Some of these belong in weaknesses, while others are true criticisms of the data, but I have grouped them together here.

Frustratingly, a number of "why"s are not discussed during the paper – things are noted but no evaluation or insight is provided into their reasoning process.

There is no explaination behind why particular benchmarks were chosen. For "System call dispatch & spinning," no reason is given as to the choice of `fcntl`. After a little thought, it does appear obvious as the implementation consists only of network and storage domains, so a simple system call was chosen that would provide communication over different domains. The authors should have noted this.

Additionally, explaination behind benchmark results are sometimes missing. The authors note that "for block sizes less that 64K," performance is better using their system than using Linux. This is likely due to the data being copied, which is in their implementation is a shared memory page, remaining in the L1 cache (which is 64K), but this is not mentioned in the paper. Buffer sizes greater than 64K cause Linux to have higher throughput figure (Figure 3). Interestingly, towards the end of the graph, VirtuOS's performance appears to tail off as the buffer size passes 1M, although there isn't enough data here to make a meaningful conclusion about this result. Similarly, there is no reason given as to why VirtuOS might be exceeding Linux's performance for the MySQL benchmark (which is likely due to page cache separation, since most CPU work is done in Dom0 the overhead of spinning isn't visible).

There is no rationale explaining the choice of the compared systems, particularly for the TCP benchmark. Relately, no explaination for why the buffer sizes used in the TCP or disk benchmarks only range up to 16 KB.

3.4 Benchmarks

No information about how the "slowdown" figures were reached in the "Process coordination" benchmark. The Y axis of "Slowdown" has no units. We are not told what the base case is (we assume normal Linux). For the average use case, it is difficult to work out what the performance penalty would be; in a desktop environment, for instance, most applications do not have many concurrent processes, and as such there would be a very large penalty (let's say 2500% where we assume N=|cores|) while in server environments this would probably reduce down to their quoted 800%. In any case, stll big.

Most of the implementation's poor performance comes from its dependence on spinning while waiting for outstanding system call requests to complete. The first and second figures for system call overhead, firstly, compare the variables in reverse. A more opaque way to phrase it would be to say "without blocking, we add 30% overhead". This is above the ∼10% overhead provided by most other comparable solutions. The second figure, which provides an overhead for when the thread gives up spinning and goes to sleep to save CPU cycles, is "rougly 14x" – or 1400%! Unless it is amortised, this is a huge performance penalty. The authors then go on to say that they "needed a much larger spinning threshold (1000 iterations) to achieve the best performance for our macrobenchmarks". Essentially, they are trying to cheat and mitigate a sleep event. Also note that there is no data for this benchmark either, nor any information on how performance figures were calculated.

Spinning to offset the performance hit of having to sleep and then wake back up comes at a cost to CPU usage, which, to the authors' credit, they note in section 5.2.1. However, this cost is only mentioned once in that section, and nowhere else in the paper. They provide an overhead figure of around 18% (20% CPU from spinning compared to 1-6% on Linux). I disagree with the assertion that the "overhead is expected to decrease as the number of concurrent system call requests ... increases," since this assumes that the speed at which system calls are handled increases more than linearly to the number of outstanding requests, which makes no sense. I also reject the conclusion that "it performs better than alternative forms of driver isolation using Xen domains" since they have not compared CPU usage between systems, only throughput. The same trick of trading CPU usge for latency could surely be implemented in Xen domains, to match or beat performance. Also, the colours in Figure 6 are terrible. The two yellow bars look identical - which is which?!

The MySQL benchmark does not properly stress the system, as most of the work happens within Dom0 and does not utilise much of the VirtuOS infrastructure (especially since it's quite likely also MySQL caches disk access itself). Disk I/O performance hit is quoted as being 30-40%.

Thus, I reject two of the three claims of the paper that argue that they have mangaged to acheieve the goal of "[imposing] tolerable overhead for general workloads", and "performance advantages for exceptionless system call dispatch", and especially the claim in their conclusion that they are able to "efficiently resume thrads upon system call completion" – 30% time overhead while enduring

almost 20% CPU overhead is not efficient. The resilience of service domains claim I accept, however.

Despite all this, since "exception-less" system calls have not been used to communicate between virtual machines, the paper's performance results may be of interest to others.

3.5 General

The term "exception-less system calls" is misleading - the system calls are *not* exception-less. Perhap the term "deferred system calls" is better. In any case, some blame must also be portioned at the original authors of the FlexSC paper. Oddly, FlexSC is not introduced properly until the end of this paper.

4 Corrigenda

It is expected that the section entitled *Related Works* would be placed towards the front end of the paper – perhaps after the Introduction or Background sections – rather than the rear.

Section 5.2.4, page 127 - "resulted in approx. 15 higher throughput" – compared to which system?

"This paper presented VirtuOS" – no, I completely forgot what I just spent the last 30-40 minutes reading. This sentence should be rephrased, or removed.

The sentence in 3.2.1 introducing FlexSC implies that it is *only* concerned with native system calls, rather than "exception-less" system calls! This should definitely be reworded.

5 Questions

The trusted computing base for service domains still remains quite large – you're running a (comparatively) large Linux instance in each VM. How would you suggest mitigating this?

What was the motivation behind using "service domains" apart from making the system more modular?

Do user processes only run in Dom0? That's what Figure 1 wants me to believe. If so that's a major failing, and would certainly affect scaling of the system with its "exception-less" system calls, if you wanted to have multiple VMs. However, I will assume this is just a misleading diagram?

References

[1] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, New York, NY, USA, 2011. ACM.

[2] Joanna Rutkowska and Rafal Wojtczuk. Qubes os architecture. *Invisible Things Lab, Tech. Rep*, 2010.