Summary  
FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator

* **Need for FEMU:**We need an efficient emulator to perform all the research work as doing test work on actual SSD is not feasible, neither practically nor economically. Current SSD research is not done at kernel level extensions and the software used for full stack research are too costly for academic organizations.  
    
  Drawbacks of some current emulators:
  + FlashEm - based on Linux block level layer, hence less portable
  + QEMU - cannot emulate multiple IO channels, as OC-SSD does
  + VSSIM - not scalable, as built on QEMU Interface

This is where the need of an open-sourced, scalable, low latency, the extensible model gave birth to FEMU.

* **Brief of FEMU:**
  + **Scalability**:  
    FEMU focuses on an important aspect of current emulators, their limitation of not scaling well with multiple IO tasks in parallel, i.e with the increasing number of worker threads, the average latency should not increase drastically. They tackled this problem by having a modification in two departments of emulation.
    - Instead of Interrupt, they moved back to Polling based design.  
      Each IO task forced a switch from Guest OS to QEMU (as done in Memory Mapped IO)
    - Multiple asynchronous IO overheads is a bottleneck, so they created an image in QEMU space and worked on the logic of *DMA,* where CPU doesn’t even know about the concurrent tasks. Similarly, here the Guest OS does not even know about subsequently happening data transfers.

Resulting graphs depict that their model scales comparatively well in comparison to actual OC-SSD.

* + **Accuracy**:  
    The aim is to calculate the end-IO time (T-endio) for each IO task, as accurate as possible. They created two models for this, calling the former one *Basic-Delay-Model* and the latter approach as *Advanced-OC-Model.*  
      
    Basic Delay Model worked on adding overload time to an emulated task, based on experimental data. For ex: addition of 50us time to entry-time of an IO. This model is enough for FTL and Garbage-Collection (GC) emulation.  
      
    Then they moved to Advanced OC Delay Model, consisting of two major changes:
    - *double-register* planes:  
      consisting of data and cache registers, current models work on single page register, so in a Write-Read task combination, first a write occurs, then a read. In the advanced OC model, each plane maintains two registers, and both the non-overlapping task done congruently.
    - *non-uniform* page latency model:   
      pages mapped to upper bits of MLC (multi-level cells) incur higher latencies than those mapped to lower bits, so experimentally they hatched a pattern to store. They devised the plan in detail.

They tested both on open-source Filebench Workload (having various combinations of IO intensive scenarios), and they got 0.5-38% error rate in Advanced OC Delay Model, much better than other emulators and even there basic delay model.

* + **Usability and Extensibility:**Various other features discussed here
    - their choice of FTL and GC schemes
    - FEMU’s usage as both white-box and black-box SSD emulator
    - multiple devices support, and non-overlapping channels
    - distributed SSDs and page level fault injection, for random corruption and faults testing
* **Critical Review of FEMU:**In this paper, the authors do address some of the major concerns of building an efficient emulator. The primary reason being the lack of knowledge about the working of firmware logic of OC-SSD is not known. The working of internal channels is transparent via liblightnvm but the inner logic is not known that well. This is one of the reasons why they faced 38% error rate in Varmail subsection of Filebench workload testbench.  
  This, in fact, will also be a big problem when setting up FEMU for distributed SSD usage.  
    
  FEMU did not discuss the FTL and GC model that well, this can be one of the major areas of improvement. This is important as some researchers might just want to test the FTL layer of the tool for checking efficient work provisioning.  
    
  The Advanced OC Delay Model approach that they devise might not be compatible with their Scaling model, as they maintain double register per plane. Scalability might get affected when they use their “DMA” logic with this double register model.

They addressed this limitation: FEMU is DRAM backed, so large capacity SSDs can’t be emulated efficiently. If they could devise an approach of using a bigger swap memory in congruence with primary memory, a near accurate model can be made (neglecting the initial swap-in and swap-out times).  
  
The actual need of OC-SSD, maintaining IO task provisioning as per requirement and achieving less fragmentation, the FEMU paper seems to not have discussed it at all, how do they read the real-time user input (liblightnvm APIs) and implement it at the same time of IO tasks lining up. The task scheduling in this department needs to be discussed a bit more.

* **Doubts:**
  + How do they implement a non-volatile memory in all this? Does NVMe provide a basic architecture solution for integrating into your own tool?
  + The Filebench workload used for accuracy testing had scenarios based on simple File transfers and Network File Transfers, but what is Varmail scenario? I wasn’t able to find a detailed explanation of it.