A Storage Device Emulator for System Performance Evaluation

MING-JU WU and CHUN-JEN TSAI, National Chiao Tung University

The performance and characteristics of the storage devices used in embedded systems can have a great influence on the overall end user experience. When building embedded systems or designing new storage device components, it is important for the designers to be able to evaluate how storage devices of different characteristics will affect the overall system performance. Storage device emulation enables a system's performance to be evaluated with simulated storage devices that are not yet available. In storage device emulation, the emulated storage device appears to the operating system (OS) as a real storage device and its service timings are determined by a disk model, which simulates the behavior of the target storage device. In the conventional storage device emulators, because the OS is running continuously in the real-time domain, the amount of time that the emulators can spend on processing each I/O request is limited by the service time of each corresponding I/O request. This timing constraint can make emulating high-speed storage devices a challenge for the conventional storage device emulators. In this article, we propose an OS state pausing approach to storage device emulation that can overcome the timing constraints faced by the conventional storage device emulators. By pausing the state of the OS while the storage device emulator is busy, the proposed emulator can spend as much time as it needs for processing each I/O request without affecting the performance of the emulated storage device as perceived by the OS. This allows the proposed storage device emulator to emulate storage devices that would otherwise be challenging or even impossible for the conventional storage device emulators. In addition, the main task of storage device emulation is offloaded to an external computer to minimize the impact of the emulation workload on the target machine. The proposed storage device emulator is implemented with the Linux OS[1] on an embedded system development board. Experimental results show that the full-system performance benchmarks measured with the proposed storage device emulator are within 2% differences compared to the results of the reference system.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques; modeling techniques*; D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction; measurements; simulation*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Storage emulation, storage evaluation

ACM Reference Format:
Ming-Ju Wu and Chun-Jen Tsai. 2015. A storage device emulator for system performance evaluation. ACM Trans. Embed. Comput. Syst. 14, 4, Article 82 (September 2015), 27 pages.
DOI: http://dx.doi.org/10.1145/2785969

1. INTRODUCTION

NAND flash storage devices are widely used in embedded systems. The research results from the related fields, such as Chang [2010], Wu [2010], and Chang et al. [2012], have contributed to the continuous improvement of the design and performance of

[1]The complete source code of the proposed storage device emulator will be made available at http://www.cs.nctu.edu.tw/∼cjtsai/research/nctusde.

Q1 Authors' address:

DOI: http://dx.doi.org/10.1145/2785969

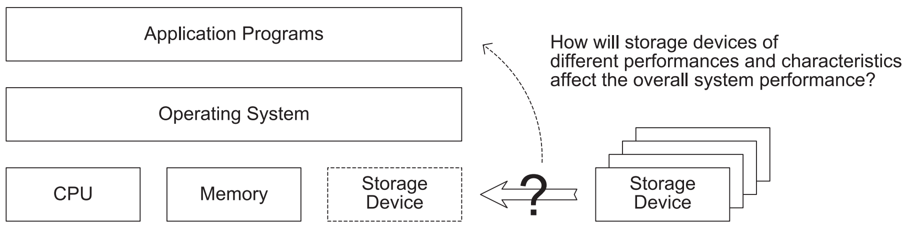
Application Programs
Operating System
CPU
Memory
Storage Device
?
Storage Device
How will storage devices of different performances and characteristics affect the overall system performance?

Fig. 1. The behaviors of the OS and application programs should be considered when trying to predict the overall system performance.

NAND flash storage devices. Other types of nonvolatile storage devices, such as Phase-Change Memory (PCM) [Zilberberg et al. 2013], Magnetoresistive RAM (MRAM), and Ferroelectric RAM (FeRAM) [Doh et al. 2007], are also being actively studied. Therefore, it can be expected that there will be an ever-increasing number of storage devices to choose from for building embedded systems. The performance and characteristics of the storage devices used in embedded systems can have a great influence on the overall end user experience [Kim et al. 2012]. Thus, it is important for the embedded system designers to be able to evaluate how storage devices of different characteristics will affect the overall system performance. It is also important for the storage device designers to be able to predict the overall system performance with simulated storage devices that are not yet available so that the design tradeoffs can be made with the overall system behaviors in mind.

While theoretical analysis or simulation using abstract models can provide quick estimates on how different storage devices might affect the overall system performance, the accuracy of these approaches can be rough and can, therefore, lead to design conclusions that are contradictory to the real-world results [Thekkath et al. 1994]. It has been shown that complex system-level interactions can hide or reduce the predicted effects of new storage device components [Ganger and Patt 1998; Traeger et al. 2008]. For another example, researchers from the networking field are also aware of the problems related to simulating the behaviors of real-world networking subsystems with simplified models [Wang et al. 2007]. Therefore, as illustrated in Figure 1, it is desirable that the behaviors of the OS and the real application programs are considered when trying to predict how storage devices of different performances and characteristics will affect the overall system performance.

1.1. Complete Machine Simulation

Complete machine simulation is one of the approaches which can allow the overall system performance to be evaluated with simulated storage devices that are not yet available. The concept of the complete machine simulation approach is illustrated in Figure 2. With complete machine simulation, a simulation model of the target machine hardware, including the storage device of interest, is first constructed. The OS and application programs are then executed on the simulated machine hardware to study the overall system performance. An attractive attribute of the complete machine simulation approach is that it can be used for studying a wide range of system configurations—anything that can be modeled and simulated can be studied. Because the entire virtual machine model is simulated in the discrete-time domain, the performances and characteristics of the storage devices that can be simulated will not be limited by the performance of the simulation host. For example, it is possible to use a slower conventional hard disk drive for simulating large and faster solid-state drive (SSD) storage devices.

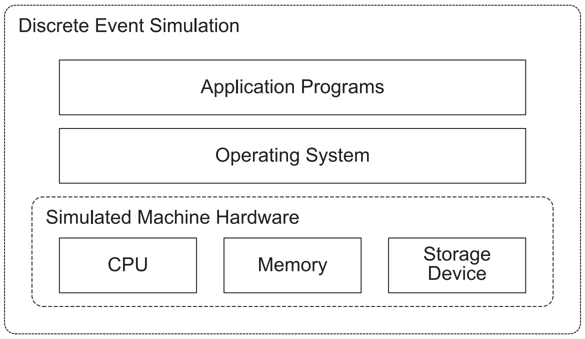

Fig. 2. Complete machine simulation.

Unfortunately, one problem with the complete machine simulation approach is that the simulation speed of the complete machine model can be much slower than the speed of the native machine hardware. For example, in the SimOS environment [Rosenblum et al. 1995], simulation using the detailed CPU model can be 3 to 4 orders of magnitude slower than running the same workloads on the native host.

Another problem with the complete machine simulation approach is the difficulties and efforts that are involved in constructing accurate complete machine models [Gibson et al. 2000; Gutierrez et al. 2014]. Moreover, the internal design details of commercial SoCs might not be made publically available by its manufactures [Eklov et al. 2011]. In particular, accurate memory subsystem models (including the cache controller, the memory controller, and the DRAM models, etc.) are often not available for commercial embedded SoCs.

1.2. Storage Device Emulation

Storage device emulation is another approach that can be used for evaluating the overall system performance with simulated storage devices that are not yet available [Griffin et al. 2002]. In storage device emulation, an emulated storage device is made available to the OS that is executing on the real machine hardware. The OS can interact with the emulated storage device similar to with a real storage device. The service timings of the emulated storage device are determined by a disk model that models the behavior of the target storage device. When an I/O request is submitted to the emulated storage device, the storage device emulator will need to carry out the following tasks:

1. Compute the response time of the I/O request by simulating the disk model.
2. Handle the actual data of the I/O request. If the I/O request is a write request, then the actual data is stored to the backing storage. Otherwise, the actual data is retrieved from the backing storage.
3. Reply the I/O response to the OS at the corresponding service time as determined by the disk model.

The conventional storage device emulators can conceptually be classified into two types of designs: (i) local virtual storage device emulation and (ii) remote actual storage device emulation. These two types of designs will be discussed in the following subsections.

1.3. Local Virtual Storage Device Emulation

An overview of the local virtual storage device emulation design (which will be referred to as the local emulation design for short for the rest of this article) is illustrated in

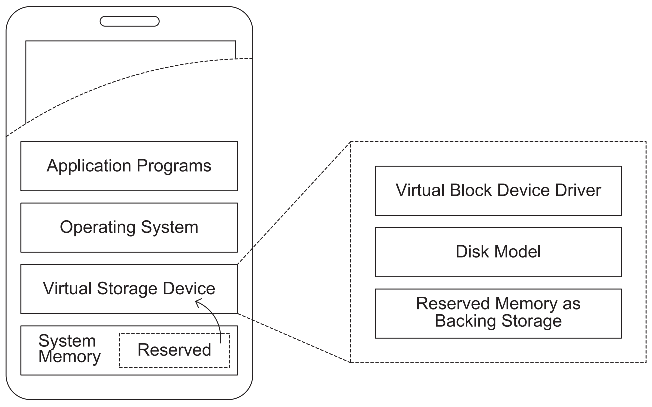

Fig. 3. Local virtual storage device emulation.

Figure 3. In the local emulation design, a virtual storage device is presented to the OS at the device driver level. The service timings of the emulated storage device are determined by a disk model which is simulated using the target system's CPU. A portion of the system memory is reserved and used as the backing storage for persisting the actual data of the emulated storage device.

While the local emulation design is convenient to implement, it has two notable limitations:

—The size of the emulated storage device will be limited by the amount of the system memory that can be reserved for the backing storage. Embedded systems are usually configured with no more than a few gigabytes of memory and, therefore, can greatly limit the size of the benchmark workload that can be used for evaluation. Furthermore, reducing the amount of the system memory that can be used by the OS will affect the original behavior of the system. If the remaining system memory is under a certain threshold, it can also prevent the proper execution of the desired benchmark workloads.

—Because the disk model is simulated using the target system's CPU, the emulator will consume CPU cycles of the target system and, therefore, will affect the original system behavior. More importantly, if the time that it takes to simulate the disk model exceeds the targeted service time of the corresponding I/O request, then the emulator will fail to emulate the intended behavior of the target storage device.

1.4. Remote Actual Storage Device Emulation

An overview of the remote actual storage device emulation design (which will be referred to as the *remote emulation design* for short for the rest of this article) is illustrated in Figure 4. In the remote emulation design, an actual storage device is emulated by the emulator software running on a separate computer and is presented to the OS at the actual I/O interface level. For example, if the target system has a Secure Digital (SD) card reader, then a storage device that is compatible to the SD standard can be emulated by the storage device emulator and inserted into the SD card reader of the target system.

Because the emulator software runs on a separate computer, the remote emulation design can overcome some of the problems faced by the local emulation design. First, the size of the emulated storage device will not be limited by the size of the system memory available on the target system. It can be as large as the system memory or

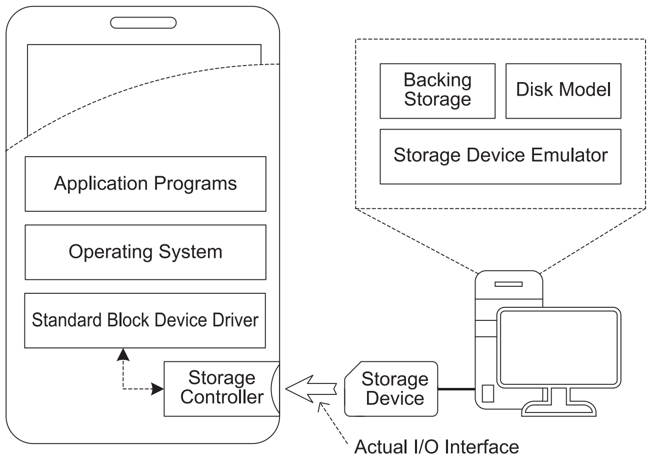

Fig. 4. Remote actual storage device emulation.

the backing storage available on the separate computer. Second, the CPU usage of the target system will not be affected by the emulator.

However, there are still some limitations with the remote emulation design:

—In both the local and remote emulation designs, the OS is running continuously in the real-time domain and the system time that the OS observes is progressing at the real-time speed. Therefore, in order to properly emulate the intended behavior of the target storage device, the conventional storage device emulators must be able to complete the processing of each I/O request in a timely fashion. The amount of time that is allowed for processing each I/O request is bounded by the service time of the particular I/O request. For example, if the computed service time of an I/O request is 1ms, then the total amount of time that can be used by the storage device emulator for simulating the disk model and accessing the backing storage cannot exceed 1ms. Otherwise, the performance of the emulated storage device will be slower than the intended target, which will lead to performance evaluation errors

—The latency and bandwidth of the emulated storage device will be limited by the latency and bandwidth of the actual I/O interface. For example, the UHS-I bus interface defined by the SD standard has a maximum bus speed of 104MB/s; therefore, storage devices that are faster than 104MB/s cannot be emulated over the UHS-I interface. This kind of limitation can prevent the emulation of faster storage devices over existing I/O interfaces.

—The features that can be supported by the emulated storage device will be limited by the characteristics of the actual I/O interface. For example, because the SD interface protocol does not support more than one outstanding I/O requests concurrently, storage devices that have features such as Native Command Queuing (NCQ) cannot be emulated over the SD interface.

1.5. Proposed OS State Pausing Approach to Storage Device Emulation

In this article, we propose a novel OS state pausing approach to storage device emulation that can overcome many of the problems faced by the conventional storage device emulation approaches. The key idea of the proposed design is to pause the state of the OS as necessary whenever the storage device emulator is busy processing the I/O requests. Because of the pausing made to the state of the OS, the system time that is observed by the OS is no longer hardwired to the real-world time. Instead, the OS is modified to observe a virtual time that can be controlled by the storage device emulator. This allows the proposed storage device emulator to spend as much time as it requires for processing each I/O request without affecting the perceived service times of the emulated storage device to the OS. Therefore, in the proposed storage device emulator, disk models of arbitrary complexity can be used for modeling the target storage devices and slower backing storages can also be used for emulating faster storage devices.

Gupta et al. [2006] proposed a technique called *time dilation* to make the OS to observe a passage of time that is a constant time slower than the real-time clock. From the OS's perspective, the physical resources in the external world will appear to be faster than their original speeds. For example, if the passage of time observed by the OS is slowed down by 10 times, then data arriving from a network interface at a physical rate of 1Gbps would appear to the OS as arriving at 10Gbps. They have demonstrated that *time dilation* is an effective method for emulating network interfaces of different speeds to the OS for end-to-end system behavior experimentations.

Although the proposed storage device emulator also makes the OS to observe a passage of system time that is different than the real-time clock, the goal and mechanisms are different from those of *time dilation*:

—In *time dilation*, the passage of system time is only slowed down by a constant factor but will never be stopped. In other words, the OS is always executing continuously. In contrast, in our proposed OS state pausing approach, the execution of the OS will be paused as necessary whenever the storage device emulator is busy processing the I/O requests. The goal of *time dilation* is to trick the OS into believing that the external resources are faster than what they actually are. On the other hand, the goal of OS state pausing is to temporarily freeze the execution of the OS so that the storage device emulator can spend unlimited amount of time on processing the I/O requests.

—The slowing down of the passage of time in *time dilation* is achieved by reducing the frequency that the timer interrupts are delivered to the OS and also by appropriately scaling down the hardware time counter that is used by the OS. For example, to slow down the passage of time observed by the OS by 10 times, the timer interrupt frequency is reduced by 10 times and the hardware time counter is also scaled down by 10 times. In comparison, pausing the state of the OS requires stopping the hardware time counter and preventing the CPU from doing any work for the OS.

—The design of the proposed storage device emulator can be extended with *time dilation*. In the proposed emulator design, the OS can be either in the paused state or in the running state. When in the running state, it is possible to apply *time dilation* so that the OS will observe that it is running on a CPU which is faster than the CPU of the target system. However, in order to keep our focus, we do not apply *time dilation* to the proposed storage device emulator in this article.

The remainder of this article is organized as follows. Related work is discussed in Section 2. The proposed storage device emulator is described in Section 3, and the experimental results are presented in Section 4. Finally, conclusions are given in Section 5.

2. RELATED WORK

Timing-accurate storage emulation has been explored in detail by Griffin et al. [2002]. Their work gives support that storage device emulation is an effective method for studying the overall system performance with simulated virtual storage devices. In their remote emulation setup, the emulator is executed on a standalone computer and connected to the target system over the SCSI interface. The DiskSim simulator [Bucy et al. 2008] is used for simulating the service times of the Seagate Cheetah X15 hard disk drive. Experimental results show that the benchmark results measured with the emulated storage device are close to the results measured with the real storage device.

VSSIM [Yoo et al. 2013] is a virtual machine (VM)–based SSD simulator that makes emulated SSD devices available to the VM. The emulated SSD device appears to the VM as a storage device connected to the IDE interface. VSSIM runs in real-time and allows the user to measure both the host performance and the SSD behavior under various design choices. The design of VSSIM can conceptually be classified as similar to the remote emulation design. The difference is that the target machine that the emulated storage device is "connected" to is a VM; therefore, VSSIM does not need to emulate the actual electrical signals of the IDE interface.

In the work of Maghraoui et al. [2010], a local emulation type storage device emulator is developed to emulate flash based SSD for the AIX OS. The emulator is implemented as a dynamic loadable kernel module and appears to the OS as a disk device. A chunk of system memory is pined for use as the backing storage, and high-resolution nanosecond granularity timers are used for simulating the delays of the I/O operations. Lee et al. [2012] have proposed another local emulation type SSD emulator. The target platform is the Linux OS and it uses a user mode SSD simulation engine for calculating the latencies for the I/O requests. The backing storage used can be either the system memory or some other external DRAM-based storage devices.

David [Agrawal et al. 2012] is another local emulation type storage device emulator which is mostly concerned about the required backing storage size for emulating large storage devices. The main idea of David is to reduce the data that needs to be saved to the backing storage so that larger storage devices can be emulated using less storage space. The key idea is that for certain types of benchmark workloads, the actual content in the file system files can be omitted and only file system metadata needs to be persisted onto the backing storage. By not persisting the actual content in the files, David is able to reduce the storage size requirements by orders of magnitude. David has been demonstrated to work with the *ext3* and the *btrfs* file systems, and in theory, it can be extended to work with other types of file systems.

The SimOS environment [Rosenblum et al. 1995, 1997; Witchel and Rosenblum 1996] is a complete machine simulator that can simulate the hardware of an SGI machine in enough detail to run the Irix OS. SimOS supports switching among a number of hardware component models which are different in terms of modeling detail and accuracy. The less-detailed models can run faster and can be used to boot and position the system into a state at which interesting workloads will begin. The simulation can then switch to use a more accurate model for detailed profiling. Griffin et al. [2000] has integrated a MEMS-based storage device simulator into the SimOS environment for studying the performance and characteristics of MEMS-based storage devices.

Finally, Canon et al. [1980] extended the standard virtual machine (VM) environment to include virtual time emulation. In contrast to the conventional VM environment, in which the real time of day (RTOD) clock is observed by the VM, the modified VM environment makes the VM to observe a virtual time of day (VTOD) clock. The VTOD clock progresses as program instructions are executed on the virtual CPU of the VM. In the modified VM environment, the system performance is evaluated against the VTOD clock. The VM environment proposed by Canon et al. can be viewed as analogous to the SimOS environment configured with the direct execution model, in which the machine instructions are executed on the native CPU whenever possible and that the internal architecture of the CPU is not modeled.

3. PROPOSED STORAGE DEVICE EMULATOR

An overview of the proposed storage device emulator is illustrated in Figure 5. The proposed emulator adopts a hybrid local/remote emulation model. Similar to the local emulation design, a RAMDISK device is emulated by the emulator kernel and presented to the OS at the block device driver level. However, instead of doing all the

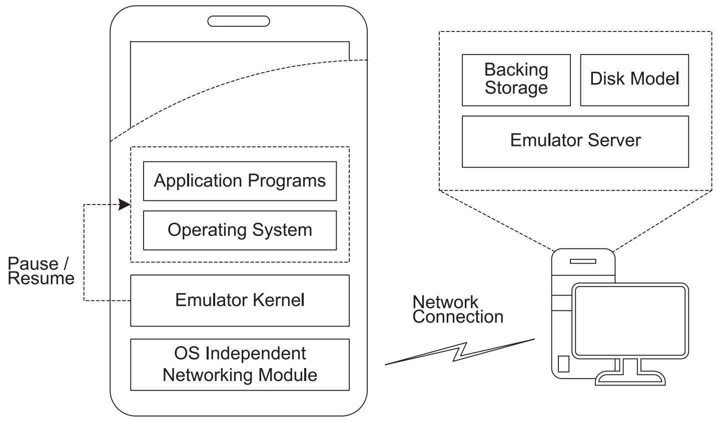

Fig. 5. An overview of the proposed storage device emulator.

work of emulating the virtual storage device on the target host, an emulator server running on a separate computer is responsible for carrying out the work of disk model simulation and data persistency. The emulator kernel and the emulator server communicate with each other over a network connection. The key idea of the proposed storage device emulator is to pause the state of the OS whenever the emulator kernel is waiting responses from the emulator server. When an I/O request is submitted to the emulated storage device, the state of the OS will be paused and the emulator kernel will forward the I/O request to the emulator server for processing. The design of the proposed storage device emulator has several advantages, which are discussed as follows.

—By pausing the state of the OS as necessary whenever the storage device is being emulated, the proposed storage device emulator can overcome the real-time timing constraints faced by emulators that are based on the conventional designs. This allows the proposed emulator to emulate storage devices that would be challenging or even impossible for the conventional storage device emulators. For example, the proposed emulator can allow disk models of arbitrary complexity to be used for simulating the behavior of the target storage device. Another problem with the conventional storage device emulation designs is that the backing storage must be faster than the storage device being emulated. In reality, faster storage devices tend to be more expensive and smaller in size. With the proposed storage device emulator, because the state of the OS is paused while the emulator is accessing the backing storage, slower storage devices can be used as the backing storage for emulating faster storage devices. For example, it is possible to use a slower conventional hard disk drive as the backing storage for emulating fast SSD devices that would otherwise be too large to fit into RAM.
—Unlike the complete machine simulation approach, there is no need to construct a complete machine simulation model for experimentation. Because the OS and the application programs are run directly on the native machine hardware, the detailed behaviors of the target machine hardware, such as the memory and cache hierarchies, the bandwidth and latencies of the system bus interconnects, and the microarchitecture of the CPU, can automatically be taken into account. Note that because the emulator kernel is also run on the target system's CPU, the cache state of the CPU might be affected. To minimize the disturbances that the proposed emulator might cause to the target system, the data buffers used by the proposed emulator can

be allocated from noncached memory regions. It is also worth noting that because of the way that the state of the OS is paused, the proposed storage device emulator also works with multiprocessor systems.

—The task of disk model simulation and actual data persistency is done by an emulator server running on a separate computer. This minimizes the interferences that the storage device emulator might cause to the target system. For example, the system memory of the target system does not need to be reserved for use as the backing storage. While it is possible to simulate the disk model directly on the target system while the OS is paused, simulating the disk model on a separate computer can reduce interfering with the cache localities of the target system.

—Network connection is used as the communication channel between the emulator kernel and the emulator server. Therefore, the features that can be supported by the emulated storage devices will not be limited by the actual I/O interface available on the target system. Furthermore, because the state of the OS is paused while the communications between the emulator kernel and emulator server are taking place, the reading and writing of the actual data for the emulated storage device can take as long as they are required to be transferred over the network channel. For example, for a read operation, the emulator kernel will complete the reading of the actual data of the I/O response from the emulator server before resuming the OS to the running state. This means that the bandwidth of the emulated storage device as perceived by the OS can exceed the bandwidth of the network channel being used. For example, it is possible to emulate storage devices that have transfer rates of over 100MB/s using a physical 802.11b wireless link, which only has a maximum transfer rate of 11Mbps. Moreover, the evaluation results will not be affected by the latencies and jilters of the network communication channel because all data transfers are carried out while the OS is in the paused state.

Table I gives a summary of when and how the proposed storage device emulator could be advantageous compared to the other approaches.

The proposed storage device emulator will be discussed in the following subsections. In Section 3.1, the concept of OS state pausing is explained. Section 3.2 gives an overview of how OS state pausing is utilized in the proposed storage device emulator. Section 3.3 discusses the emulator server and explains the concept of disk model simulation time rollback, which is required for emulating storage devices that can support concurrent I/O request processing and I/O request prioritization. The detailed operation of the proposed storage device emulator is presented in Section 3.4, and finally, the OS independent network subsystem is discussed in Section 3.5.

3.1. Operating System State Pausing

The state of the OS can be viewed as composed of the combined state of all the processes running in the OS and the current values in the clock counter and the hardware timers used by the OS. By freezing the execution of all the processes in the OS and stopping the clock counter and the hardware timers, the state of the OS can be paused. In the proposed storage device emulator, the state of the OS is paused by the emulator kernel using the following procedure: (i) Stop the clock counter and the hardware timers by programming their control registers. (ii) If the target platform is a symmetric multiprocessor system, use IPIs (interprocessor interrupts) to interrupt all the other CPUs in the system and schedules a high-priority busy waiting loop on them. The busy waiting loop will prevent the other CPUs from executing any processes that belongs to the OS.

The execution of the OS is resumed from the paused state using the following procedure: (i) Resume the counting of the clock counter and the hardware timers. (ii) Terminate the busy waiting loops so that all the CPUs will resume to their original work.

Table I. Comparison of the Proposed Approach with the Other Approaches

Compared to	Possible Problems with the Other Approaches	Advantages of the Proposed Approach
Complete machine simulation	The fidelity of the complete machine simulation model might not be a good approximation of the target system due to lack of exact models for critical SoC components. The speed of the complete machine simulation model might be too slow for conducting large full-system benchmarks.	By conducting the experiments using the real hardware of the target system, high modeling fidelity and fast simulation speed can be achieved.
Conventional real-time storage device emulation	The time needed for simulating the disk model might be longer than the desired I/O response timings. The speed of the backing storage might be slower than the desired I/O response timings.	By pausing the state of the OS while the disk model is being simulated and the backing storage is being accessed, the proposed storage device emulator can spend as long as it requires on preparing the I/O responses.
Local virtual storage device emulation	The main memory available on the target system might be too small for emulating larger storage devices.	By offloading the emulation task to a separate remote server, the size of the emulated storage device can be as large as the backing storage available on the remote server.
Remote actual storage device emulation	The physical I/O interface available on the target system might not be fast enough (in terms of bandwidth and latency) for emulating the desired storage device.	By pausing the OS and transferring the actual data over the network links, storage devices of any speed can be simulated and made available to the OS.

The pausing and resuming of the OS is entirely transparent to the applications running on the OS, and the OS can still correctly service any system calls requested by the applications. Therefore, all user space applications can be run unmodified on the proposed system.

While the OS is in the paused state, the emulator kernel will not be able to utilize certain OS services such as interrupts or timers. Therefore, a polling-based OS independent networking module that does not rely on interrupts or system timers is design for use by the emulator kernel to communicate with the emulator server. The details of the OS independent networking module will be further discussed in Section 3.5.

Because of the pausing made to the OS, the system time that the OS observes is no longer linked to the real-world time. Instead, a virtual system time that is derived from the value in the clock counter is observed by the OS. To preserve the original system behavior, all the external events to the OS should be scheduled according to the virtual system time. In the proposed storage device emulator, the only external events that need to be considered are the completion signals from the emulated storage device and the timer interrupts. The design of the proposed storage device emulator makes sure that the response signals from the emulated storage device are delivered to the OS according to the virtual system time.

In the ideal case, the state of the OS right before being paused and right after being resumed should be exactly the same. However, there are three factors that can affect the deviation of the OS states before and after pausing:

—The first factor is the resolutions of the clock counter and the hardware timers. Each time that the counting of the clock counter and the hardware timers are paused and then resumed, the fraction values that are currently accumulated in the digital circuits will be lost. For example, if the clock counter has a resolution of 1μs, then any passage of time that is less than 1μs will be lost when the clock counter is paused by writing to its control registers. In our experimental environment, the resolutions of the clock counter and the hardware timers used by the Linux OS is approximately 3ns.

—The second factor is how fast the other CPUs in the system can be put into the busy waiting loop. In the ideal case, the other CPUs in the system should be switched immediately to execute the busy waiting loop as soon as the clock counter and the hardware timers are paused. However, it is possible that the other CPUs in the system will have their interrupts disabled when the IPIs are issued. In this case, the busy waiting loop will have to wait until the interrupts are enabled again. During this period of time, the other CPUs would have completed additional tasks that are not supposed to be done when the OS is in the paused state. On systems that can support nonmaskable IPIs, such as the NMI in the x86 processors [Intel 2013] or the FIQ in the ARM processors, the nonmaskable IPIs can be used to immediately put the other CPUs into the busy waiting loop regardless of their interrupt enabling states. In practice, the CPUs in a properly designed system should not be in the interrupt-disabled state for any long periods of time.

—The third factor is the effect of the CPU cache disturbance. When the OS is in the paused state, the CPU of the target system will need to execute instructions for handling the communication between the emulator kernel and the emulator server; therefore, the cache state of the CPU will be different before and after OS pausing. To get an ideal of the scale of the disturbance that the proposed emulator will have on the target system's CPU cache states, we used the *ftrace* [Kernel 2014] function tracer in the Linux kernel to trace the kernel execution path for handling I/O requests. With the proposed emulator emulating a virtual storage device, the sizes of instruction codes executed by the CPU for handling a read or write request from user space is approximately 101KB and 151KB, respectively. Compared to a simple local storage device emulation type emulator, the sizes of instruction codes executed by the CPU for handling a read or write request is approximately 91KB and 140KB, respectively. Since the L2 cache size of the CPU used in our experimental platform is 512KB, the additional cache overhead introduced by the proposed emulator is only less than 2% of the L2 cache. Furthermore, modern embedded processors are likely to have larger caches. For example, the Apple A7 SoC [Wikipedia 2014a] incorporates a 1MB L2 cache and a 4MB L3 cache. The cache disturbance that the proposed emulator will have on those processors would be even less.

Even though there could be some slight deviations of the OS states before and after pausing, if the scale of the deviations is small compared to the frequency of its occurrence, then the overall system behavior would still be close enough to the original target system for conducting performance studies. In the proposed emulator, the frequency that the OS state pausing is happening is roughly proportional to the frequency that the I/O requests are processed by the emulated storage device. The scale of the deviation of our experimental platform is at the nanoseconds level, and from the experimental results, we have validated that the proposed OS state pausing approach is effective for emulating storage devices that have service times at the millisecond and microsecond levels.

3.2. Operating System State Pausing and Storage Device Emulation

This section discusses how OS state pausing is utilized in the proposed storage device emulator. During operation, the OS in the proposed storage device emulator is either in the paused state or in the running state. To avoid having a real-time timing constraint on the progressing speed of the emulator server and the performance of the

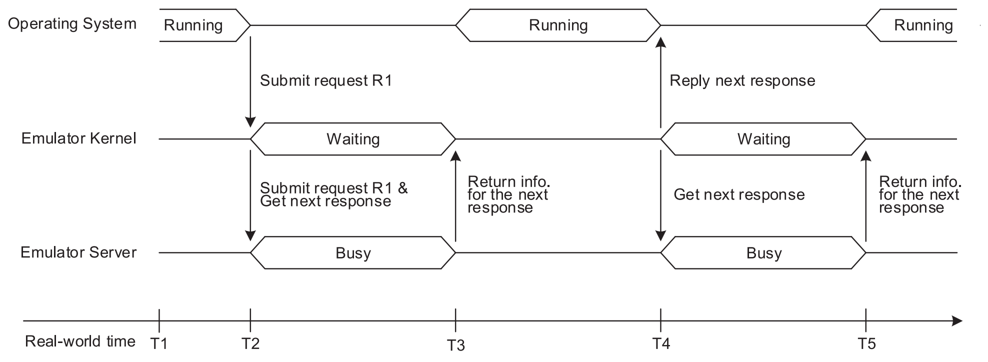

Fig. 6. Interactions between the OS, the emulator kernel, and the emulator server.

communication channel between the emulator kernel and the emulator server, the emulator kernel is designed to only make requests to the emulator server when the OS is in the paused state. The OS will be put into the paused state whenever the emulator kernel is waiting for the responses from the emulator server for the information that it needs for emulating the storage device. That is, the OS and the emulator server will never be in the operating state at the same time. In other words, the synchronization between the emulator kernel and the emulator server only happens when the OS is in the paused state.

Before resuming the OS back to the running state, the emulator kernel will need to have all the information that it needs for emulating the storage device until the next time point that it will communicate with the emulator server. To achieve this, the emulator server is designed to simulate the disk model to a future time point at which the information of the next I/O response can be determined while the OS is in the paused state.

When the OS is put back to the running state, the emulator kernel will use the information provided by the emulator server to set a timer interrupt at the completion time of the next I/O response. Because the timer service provided by the OS is utilized, the completion times of the I/O responses will automatically be scheduled according to the virtual system time of the OS. When the emulator kernel is notified by the timer interrupt, the corresponding I/O response will then be replied to the OS at the desired completion time as indicated by the disk model.

A sequence diagram is illustrated in Figure 6 to show an example interaction between the OS, the emulator kernel, and the emulator server. As previously shown in Figure 5, the emulator kernel is run on the same machine as the OS and the emulator server is run on a separate computer.

At T1, the OS is running continuously and the emulator server is in the idle mode waiting for commands from the emulator kernel. At T2, an I/O request, R1, is generated by the OS. As soon as R1 is received, the emulator kernel will pause the state of the OS and submit the parameters about R1 to the emulator server. The actual data of R1 is also submitted to or retrieved from the emulator server depending on whether R1 is a read or a write request.

After submitting the I/O request R1, the information for the next I/O response is requested by the emulator kernel. The information of an I/O response includes the ID and the completion time of the I/O response. Because the OS is paused while the emulator server is processing the commands from the emulator kernel, the time between T2 and T3 can be arbitrary long. Before resuming the execution of the OS at T3, the emulator kernel will set a timer event to interrupt itself at the corresponding completion time of the next I/O response. When woken up by the timer interrupt, the emulator kernel can then reply the next I/O response to the OS.

Notice that the emulator kernel asks for the information for the "next" I/O response and not specifically for the information for R1, which is the most recently submitted I/O request. This is because that the next I/O request that will be completed by the emulated storage device might not be R1. For example, if the emulated storage device supports I/O prioritization and that it is currently servicing an I/O request that has a higher priority than R1, then R1 will not be serviced until the previous I/O request has been completed. Another example is that if the emulated storage device supports processing of concurrent I/O requests and that there is an I/O request, for example R0, which is currently being processed, then it is possible that R0 will be completed before R1.

Please note that as far as the emulator kernel is concerned, it does not need to know the reason of why the "next" I/O response is different from the most recently submitted I/O request. The important thing is that the interface between the emulator kernel and the emulator server will permit the emulator server to change the answer for the next I/O response due to I/O requests that are later submitted to the emulator server.

After the I/O response is replied to the OS at T4, the emulator kernel will communicate with the emulator server again to determine if there will be another I/O response that is going to be completed by the emulated storage device. If there is, then the emulator kernel will set a timer event to interrupt itself at the corresponding completion time of that next I/O response. On the other hand, if there is no further I/O response from the emulated storage device, then the emulator kernel will leave the OS in the running state until future I/O requests are generated by the OS.

3.3. Emulator Server with Disk Model Simulation Time Rollback Support

The emulator server is responsible for simulating the disk model and persisting the actual data for the emulated storage device. It is designed to handle four types of commands from the emulator kernel, which are described in Table II.

In order to support the *get next response* command, the emulator server will need to simulate the disk model past the current system time of the OS. When simulating the disk model between the current system time of the OS and until the completion time of the next I/O response, the disk model is simulated under the assumption that the OS will not submit any additional I/O requests to the emulated storage during this period of time. For example, if an I/O request is submitted to the emulator server at V1 and the next I/O response is determined to be completed at V3, then the disk model is simulated under the assumption that the OS will not submit any I/O request to the emulated storage device between V1 and V3. This assumption is required because the disk model is being simulated into the "future" time. However, this assumption might not always be true if the emulated storage device can handle more than one I/O requests from the OS. For example, if another I/O request is generated by the OS between V1 and V3, then the state of the disk model will need to be *rolled back* to a prior simulation time at which when the new I/O request is generated. The disk model can then be simulated again with the new I/O request being taken into account.

An example is illustrated in Figure 7 to show a scenario when disk model simulation time rollback is performed by the emulator server. At virtual time V1, an I/O request, R1 is generated by the OS. Assume that R1 is the only I/O request currently submitted to the emulated storage device and that R1 will be completed by V3 if the emulated storage device does not receive any other I/O requests before V3. When the *get next response* command is issued to the emulator server at V1, the disk model will be simulated to

Table II. Commands Supported by the Emulator Server

Command Type	Parameters	Description
Reset	*Disk size* *Backing storage configuration* *Disk model configuration*	The *reset* command initializes the state of the emulator server to prepare for emulating a new storage device. The *disk size* parameter specifies the size of the storage device to be emulated. The *backing storage configuration* is used for selecting the backing storage device to use (e.g., RAM or SSD). The characteristics of the target storage device to be emulated can be controlled by the *disk model configuration*. For example, the disk model in our experimental implementation can support the configuration of *speedup factor* and the *number of channels*.
Data Transfer	*Sector number* *Read/write type* *Sector data*	The *data transfer* commands are used for the persistence of the actual data. The write data transfer command is used to save the actual data to the backing storage, and the read data transfer command is used for retrieving the actual data from the backing storage.
Submit Request	*Request time* *Start sector* *Sector count* *Read/write type*	The *submit request* command is used for submitting I/O requests to the emulator server. The emulator server will keep track of the I/O requests that have been received and it will simulate the disk model accordingly.
Get Next Response	*OS system time*	The emulator kernel uses the *get next response* command to get the information for the next I/O response that will be completed by the emulated storage device. If there are no I/O requests being processed by the emulated storage device, then the return value of this command will indicate that there will be no next response.

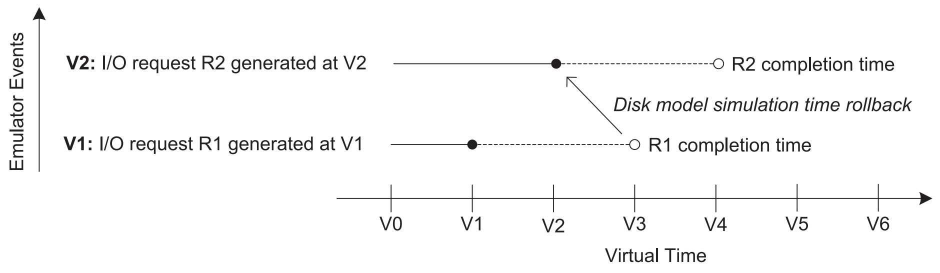
Emulator Events
V2: I/O request R2 generated at V2
R2 completion time
Disk model simulation time rollback
V1: I/O request R1 generated at V1
R1 completion time
V0
V1
V2
V3
V4
V5
V6
Virtual Time

Fig. 7. A scenario showing the emulator server performs a disk model simulation time rollback. Solid circles represent the virtual system time of the OS, and white circles represent the simulation time of the disk model.

until time V3, at which it is the time that R1 will be completed. The emulator kernel will use the replied information to set a timer to be expired at V3 so that R1 will be replied to the OS when the system time of the OS reaches V3. After setting the timer, the emulator kernel will put the OS back into the running state. Now, assume that the OS has generated another I/O request, R2, when the system time reaches V2. The emulator kernel will then submit R2 to the emulator server and issue another *get next response* command. At this time, the simulation time of the disk model will need to be rolled back to V2 so that I/O request R2 can be properly considered by the disk model.

For example, if R2 has a higher priority than R1, then after the disk model is rolled back to V2, the disk model will determine that the processing of R1 should be preempted by R2. Therefore, when the emulator kernel issues a get *next request command* at V2, the emulator server will reply that the next I/O response is R2. For another example, if the emulated storage device supports concurrent I/O request processing and that R2 has a shorter service time than R1, then even that R2 is received after R1, its completion time might still be earlier than V3. In this case, even if the processing of R1 is not interrupted by R2, the next I/O request that will be completed by the emulated storage device will still change to R2.

In the actual implementation of the emulator server and the disk model, the simulation time of the disk model might not be able to be rolled back to any arbitrary point of time. To allow the rollback of the disk model to any arbitrary point of time would require that every state progression of the disk model to be saved, which might not be practical. A design technique can be used so that the state of the disk model is only saved at the time point at which the *get next response* command is received. With this design technique, the simulation time of the disk model is first rolled back to the previous state at which the previous *get next response* command is issued, and the disk model can then be simulated from there on. Using the scenario illustrated in Figure 7 as an example: when the *get next response* command is received at V2, the disk model is rolled back from V3 to V1 and then simulated starting from V1. Between V1 and V2, the disk model will only consider R1. And from V2 and on, the disk model will consider both R1 and R2. With this design technique, supporting disk model simulation time rollback would only require that the emulator server to be able to roll back the disk model to the states at which the previous *get next response* commands are received. This can greatly simplify the design and implementation of the emulator server and the disk model.

For each rollback operation, the amount of work that is discarded is approximately equal to the amount of time for simulating one I/O request. For example, if on average it takes the disk model T units of time to simulate an I/O request, then the cost of each rollback operation, which equals to the discarded simulation progress of the disk model, will be approximately T units of time.

The upper bound for the frequency at which the rollback operations will be occurring during simulation is no more than the number of I/O requests being submitted to the emulator server. For each I/O request submitted to the emulator server, at most one rollback operation could be triggered. That is, if the submission time of the I/O request is earlier than the earliest completion time of the I/O requests already in the disk model, then one rollback operation will be performed on the disk model. Therefore, for N I/O requests, the maximum number of rollback operations that could have occurred will not exceed N.

3.4. Detailed Operation of the Proposed Storage Device Emulator

A more detailed view of the proposed storage device emulator is illustrated in Figure 8. The emulator kernel is responsible for emulating a virtual storage device to the OS at the block device driver level and is composed of two primary components: the request handling function, (described in pseudocode listing 1) and the response replying thread (described in pseudocode listing 2). The request handling function is part of the block device driver that is registered to the OS. Whenever the OS generates I/O requests toward the emulated storage device, it will invoke the request processing function.

The response replying thread is a high-priority background kernel thread used for replying the I/O responses to the OS. When the completion time of an I/O response is determined, the emulator kernel will set a timer event, which will wake up the response replying thread at the corresponding completion time. When the response replying thread is woken up, it will then reply the I/O response to the OS.

The detailed operation steps of the proposed storage device emulator depicted in Figure 8 are discussed as follows.

Step 1: Receive I/O request from the OS. When an I/O request is generated by the OS, the request handling function in the emulator kernel will be invoked. A set of

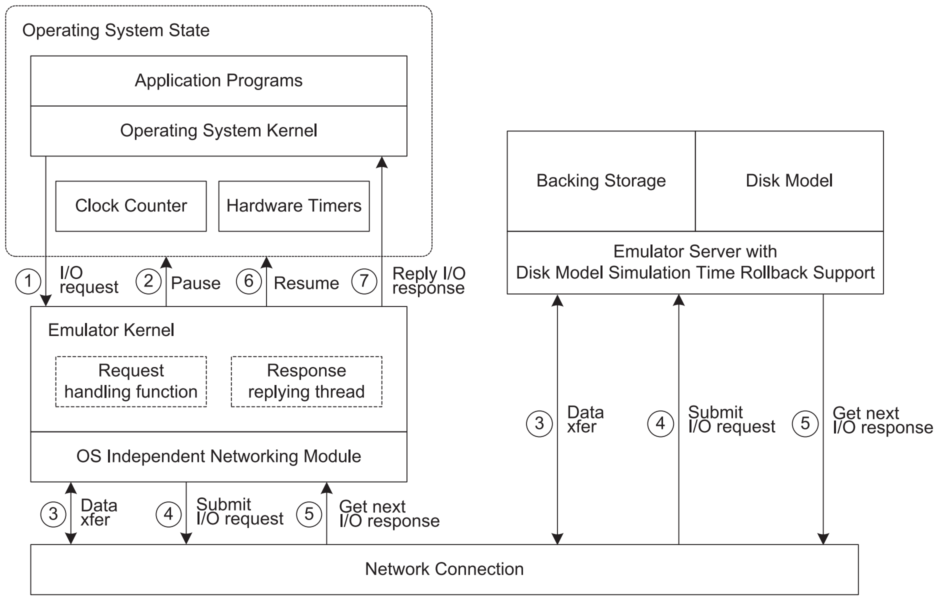

Fig. 8. Detailed operation of the proposed storage device emulator.

noncacheable memory buffers in the emulator kernel are used to simulate the memory mapped I/O interface of the storage device controller. If the I/O request from the OS is a write request, then the actual data will be copied to the write buffer (the *temp-WriteBuffer* in the pseudocode listings) of the emulator kernel at this time to mimic the transferring of the actual data to the storage device controller.

Step 2: Pause the state of the OS. The state of the OS will be paused by the emulator kernel before initiating communication to the emulator server. This allows the emulator server to take as long as it requires for processing the commands from the emulator kernel without affecting the behavior of the emulated storage device as perceived by the OS.

Step 3: Transfer the actual data of the I/O request. If the I/O request is a write request, then the data that was copied to the write buffer (*tempWriteBuffer*) in step 1 will be transferred to the emulator server for persistence. Otherwise, if the I/O request is a read request, then the actual data of the I/O request will be retrieved from the emulator server and stored in the noncacheable read buffer (the *tempReadBuffer* in the pseudocode listings) of the emulator kernel. The retrieved actual data is not copied directly to the I/O response buffers that belong to the OS at this time to mimic the behavior of real storage devices. For real storage devices, the requested actual data will not be available until the I/O request is completed. If the actual data is copied to the I/O response buffers that belong to the OS at this time, then the temporal cache locality of the actual data will be different from that of the real-world cases. The retrieved actual data will temporarily wait in the read buffer until the I/O response is actually going to be replied to the OS in step 7. At that time, the actual data will then be copied from the read buffer to the I/O response buffers that belong to the OS.

Step 4: Submit the I/O request to the emulator server. The parameters of the I/O request, which includes the request time, request type (read or write), start sector, and number of sectors, is submitted to the emulator server. This information is used by the disk model to simulate the completion time of the I/O requests.

Step 5: Get the information of the next I/O response. Before resuming the OS from the paused state, the emulator kernel needs to know the completion time of the next I/O response. A timer event will be set to wake up the response replying thread at the corresponding completion time. If there is a timer event already set for the response replying thread, then the timer event will be updated to the new completion time if it is earlier than the current value set in the timer.

Step 6: Resume the OS from the paused state. After the response replying thread is properly set to be woken up at the completion time of the next I/O response, the OS will be resumed from the paused state. The OS then runs continuously until one of the two following scenarios happens: (i) Another I/O request is generated by the OS. In this case, the newly generated I/O request is handled following the same steps beginning from step 1. (ii) The timer event for the response reply thread expires and the response replying thread is woken up. In this case, the processing continues to step 7.

Step 7: Reply the I/O response to the OS. When the system time of the OS has advanced to the completion time of the next I/O response, the response replying thread will be woken up by the timer event. If the I/O response is for a write request, then the response relying thread simply notifies the OS that the write request has been completed. Otherwise, if the I/O response is for a read request, then the actual data in the *tempReadBuffer* will be first copied to the corresponding I/O response buffers that belong to the OS, and then the OS will be notified about the completion of the read request.

PSEUDOCODE 1. Request Handling Function

```
function request_fn
        Request = get next pending request from OS;
        if Request is a write request then
        Copy sector data to Request.tempWriteBuffer;
    end
    Pause OS;
    if Request is a write request then
        Transfer Request.tempWriteBuffer to emulator server;
    else
        Request.tempReadBuffer = Retrieve sector data from emulator server;
    end
    Get HasNextResponseTemp, NextResponseTimeTemp, NextResponsesTemp from
     emulator server;
    if HasNextResponseTemp == TRUE and
            ( HasNextResponse == FALSE or
            NextResponseTimeTemp < NextResponseTime )
    then
        HasNextResponse = TRUE;
        NextResponseTime = NextResponseTimeTemp;
        NextResponses = NextResponsesTemp;
        Signal the response replying thread;
    end
    Resume OS;
end function
```

Remarks on pseudocode 1: In Linux, the request handling function is invoked by the kernel whenever there are I/O requests to be submitted to the storage device. Request.tempWriteBuffer is a noncached memory buffer for temporarily storing the sector data of a write I/O request. Request.tempReadBuffer is a noncached memory buffer for temporarily storing the sector data of a read I/O request. The temporary buffers are allocated from a noncached memory address space to simulate what a real device driver would do. That is, the sector data need to be transferred between the CPU and the device controller. The HasNextResponseTemp, NextResponseTimeTemp, and NextResponsesTemp variables store the information regarding the next I/O response from the emulator server. The *if* clause in line 14 tests if the completion time of the new next I/O response will be earlier than the current timer interrupt of the response replying thread. If true, then the response replying thread will be signaled so that the timer interrupt can be updated to an earlier time point corresponding to NextResponseTimeTemp. Note that the request handling function only makes requests to the emulator server while the OS is being put into the paused state.

PSEUDOCODE 2. Response Replying Thread

```
1   HasNextResponse = FALSE;
2   repeat
3       if HasNextResponse == FALSE then
4           Sleep until being signaled;
5       end
6       repeat
7           Sleep until NextResponseTime has arrived or being signaled;
8       while thread is woken up because of being signaled by the request handling function;
9       for each Response in NextResponses do
10          if Response is a read response then
11              Copy Response.tempReadBuffer to the response buffers belonging to the OS;
12          end
13          Reply Response to operating system;
14      end
15      Pause OS;
16      Get HasNextResponse, NextResponseTime, NextResponses from emulator server;
17      Resume OS;
18  until Storage emulator is stopped;
```

Remarks on pseudocode 2: The response replying thread is a high-priority background kernel thread responsible for replying the I/O responses to the OS. In lines 3 to 5, if there is currently no I/O response pending to be replied to the OS, the response replying thread will go into an infinite sleep until being signaled by line 21 of the request handling function. The *while* loop between lined 6 and 8 ensures that the timer interrupt that will wake up the response replying thread is set to the most current value assigned by the request handling function (line 19 of the request handling function). Lines 9 to 14 reply the I/O response to the OS when the completion time of the I/O request has arrived. After the I/O response has been replied to the OS, line 16 gets the information for the next I/O response from the emulator server. Note that the response replying thread only makes requests to the emulator server while the OS is put into the paused state.

3.5. Operating System Independent Networking Module

An OS independent networking module is designed in the proposed storage device emulator to be used by the emulator kernel for communicating with the emulator server. The OS independent networking module is used instead of the standard networking stack that comes with the OS because of the following reasons:

—While the OS is in the paused state, the CPUs will have their interrupts disabled. The network device drivers that come with the OS cannot be used because they depend on interrupts for working with the network hardware. In the OS independent networking module, a special polling-based network device driver is implemented for working with the network hardware. The special device driver provides its own set of polling-based APIs for use by the emulator kernel.

—To minimize the interference that the OS independent networking module will have to the original system behavior, the packet buffers which are used by the network hardware are not allocated from the OS kernel. Instead, a packet buffer manager is designed for allocating and reclaiming the packet buffers from a noncached memory pool owned by the emulator kernel.

—When the OS is in the paused state, the timer service provided by the OS will not function properly because the clock counter and the hardware timers are stopped. This means that the TCP implementation in the standard OS networking stack will not work because the retransmission timers will not be functioning correctly. In the proposed storage device emulator, the emulator kernel communicates with the emulator server using the UDP protocol. The UDP packets are assembled and parsed by the emulator kernel without going through the full OS networking stack. To transmit a UDP packet, the UDP packet is assembled by the emulator kernel and handed to the special polling-based device driver in the OS independent networking module directly. To receive the response packet, the emulator kernel uses the polling-based API to poll the network hardware repeatedly until the expected packet has arrived. If the expected response packet is not received after a certain number of polling attempts, the request packet will be retransmitted again by the emulator kernel.

4. EXPERIMENTAL RESULTS

We have implemented the proposed storage device emulator with the Linux OS. The target hardware platform used is the ZedBoard development board [ZedBoard 2014]. ZedBoard contains a 667MHz dual-core ARM Cortex-A9 MPCore processor and 512MB of DDR3 main memory. The Linux kernel used is based on the kernel version 3.10.0 from the Xilinx source repository [Xilinx 2014]. The root file system used is based on the *nano* build of the Linaro release version 13.11 [Linaro 2014]. The device driver for the ARM global timer [Xilinx 2013] is backported from the mainline kernel [Kernel 2013a], and the Linux kernel is configured to use the ARM global counter as its *clocksource* and *clockevent* devices [Stultz et al. 2005; Gleixner and Niehaus 2006; Gleixner and Molnar 2006; Gleixner 2007]. The ARM global timer runs at half the speed of the ARM processor. At 333MHz, it provides the *clocksource* and *clockevent* devices a timing resolution of approximately 3ns. In other words, the resolution of the clock counter and hardware timers used by the Linux OS is approximately 3ns. The emulator server is implemented as a user space program on a separate computer running Ubuntu Linux. The host computer for the emulator server is equipped with an Intel Core i5-3210M CPU running at 2.5GHz, 4GB of RAM, and a Seagate Momentus Thin 320GB hard disk drive as the backing storage. ZedBoard is connected to the emulator server using a 1Gbps Ethernet LAN.

To measure how much disturbance the operations of the proposed emulator (e.g., OS pausing, OS resuming, network communication) will have to the target system, the simulation results of the proposed emulator are compared to a reference system.

The reference system is an unmodified Linux OS, which runs on the same ZedBoard platform, and has a RAMDISK storage device emulated by a local emulation type emulator. The parameters of the RAMDISK device are designed so that its performances are comparable to UHS-1 SD cards. The same disk model will be used by both the proposed emulator and the reference system. Ideally, the disturbances caused by the operations of the proposed emulator should be low and the evaluation results from the proposed emulator should match closely to the results from the reference system.

The reasons of why a RAMDISK device, which is emulated by a local emulation type emulator, instead of a real storage device is used in the reference system are given as follows:

—The structure of the emulator kernel is very similar to how a local emulation type emulator would be implemented. The main differences between the proposed emulator and a local emulation type emulator are the extra steps needed for pausing and resuming the OS and for performing network communications. Using a local emulation type emulator in the reference system allows us to measure how much disturbance the extra operations of the proposed emulator has caused to the target system.
—In our experience, the work needed for accurate modeling of real storage devices is not trivial. For example, to model the exact behavior of a NAND storage device that contains a flash translation layer would require that the firmware of the flash translation layer to be available; otherwise, the exact sector data placement in the emulated storage device will be different from the real device, and this will lead to different I/O response timings. If the behaviors of the real storage device are not modeled exactly, then testing the proposed emulator using an inexact disk model will mean that we will not be able to tell if the inaccuracies are caused by the disturbances of the proposed emulator or the inaccuracy of the disk model.
—The physical I/O interface on the target platform might limit the performances or characteristics of the real storage devices that can be tested. For example, the SD interface supported by the ZedBoard is at version 2.00, which can only support a maximum bus speed of 25MB/s. Therefore, it is not possible to validate the proposed emulator with a real SD device faster than 25MB/s on the ZedBoard platform. With a local emulation type emulator, we were able to emulate storage devices with the speeds of over 100MB/s for validating the proposed storage device emulator, which is not possible if real storage devices are used. Moreover, next generation standards, such as the JEDEC UFS (Universal Flash Storage) [JEDEC 2013], will support the command queuing feature and, therefore, cannot be emulated over the SD interface. By comparing against a local emulation type emulator, we were able to validate the proposed emulator under concurrent I/O scenarios.
—Besides using different I/O workloads for testing the proposed storage device emulator, we also want to test if the proposed emulator can handle storage devices of different speeds and concurrency levels. By using a local emulation type emulator as the comparison target, storage devices of different speeds, and concurrency levels can easily be modeled for testing.

The random and sequential read/write performances of a Transcend UHS (Ultra High Speed) Speed Class 1 SD memory card are measured and used to model a hypothetical baseline storage device. The measured performance numbers are shown in Table III. It is important to note that the baseline storage device is by no means trying to simulate the actual behavior of the Transcend SD memory card. These performance numbers are used for modeling the baseline storage device so that the performance of the baseline storage device is in line with the current top of the line storage devices.

The characteristics of the baseline storage device can be controlled by two parameters: the *speedup factor* and the *number of channels* parameters. Both parameters are varied from x1 to x4 for deriving a total of 16 storage device configurations for validating the proposed storage device emulator. The storage device configuration *ChN-xS* means that the *number of channels* the storage device has is *N* and it has a *speedup factor* of *S*. The *number of channels* parameter configures the number of concurrent I/O requests that the storage device can process simultaneously. For example, if four

Table III. Performance Numbers Used for the Baseline Storage Device

	Random Workload		Sequential Workload	
Transfer Size	Read (ms)	Write (ms)	Read (ms)	Write (ms)
512 bytes	0.0051	1.1136	0.0019	0.0092
1KB	0.0053	1.1123	0.0020	0.0095
2KB	0.0054	1.1136	0.0023	0.0099
4KB	0.0057	1.1601	0.0027	0.0115
8KB	0.0063	1.2853	0.0033	0.0136
16KB	0.0071	1.2579	0.0036	0.0035
32KB	0.0096	2.2222	0.0068	0.0070
64KB	0.0151	2.7027	0.0139	0.0137

I/O requests are submitted to an emulated storage device that has four channels, then all four I/O requests will be processed concurrently. The *speedup factor* parameter configures how much faster the storage device is compared to the baseline storage device. For example, a speedup factor of two means that the response times of the emulated storage device will be half of the baseline storage device.

The total amount of memory available on ZedBoard is only 512MB. Therefore, the size of the emulated storage device is set to be 400MB, which leaves 112MB of memory for use by the Linux OS. A 12MB *tmpfs* temporary file storage is created and used for temporarily storing the outputs from the benchmark applications. Note that for the proposed storage emulator, because the actual data is persisted on the remote host, the Linux OS will be able to use the complete 512MB of main memory. However, we still configure the Linux OS to use only 112MB of memory so that the testing conditions are equivalent to the reference system.

The benchmarks are executed using the auto-pilot automation framework [Wright et al. 2005; Wright and Zadok 2007]. Each benchmark is run at least 10 times and until the radius of the 95% confident interval is less than 5% of the mean. The *cron* and *rsyslog* services in the Linux OS are disabled while the benchmarks are performed to reduce possible interferences. The system is rebooted after each benchmark run. The evaluation results are presented in the following subsections.

4.1. Sequential Read Bandwidth

In the first test, the sequential read bandwidth of the emulated storage device is measured using the *hdparm* utility. The I/O scheduler for the emulated storage device is set to the *noop* scheduler [Wikipedia 2014b]. Setting the *noop* I/O scheduler disables the anticipatory I/O scheduling [Iyer and Druschel 2001] that would otherwise be performed by the default I/O scheduler in the Linux kernel. With anticipatory I/O scheduling disabled, the read requests generated by the *hdparm* utility will be submitted to the emulated storage device as soon as they are created. The measured results are as illustrated in Figure 9. All results from the proposed storage device emulator are within 1% differences to the results from the reference system.

4.2. Postmark Workload

The postmark workload is used as the next set of workloads for validation. The *ext3* file system is created on the emulated storage device and both scenarios for the *noop* I/O scheduler and the Completely Fair Queuing (CFQ) I/O scheduler are tested. To test how the proposed storage emulator works under multithreaded environment, we run 8 concurrent postmark threads to generate a total of 100,000 transactions over a set of 10,000 files. Each postmark thread will generate 12,500 transactions. All results from the proposed storage device emulator are within 2% differences to the results

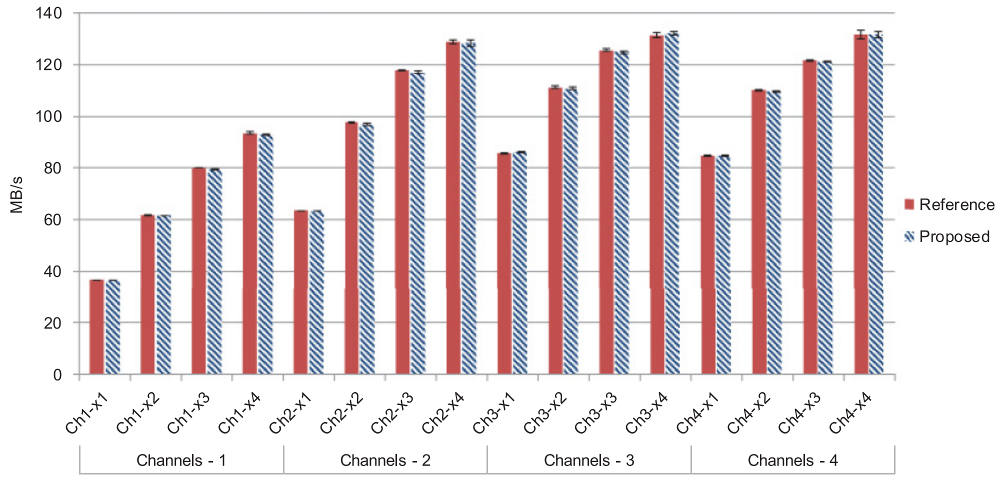

Fig. 9. Sequential read bandwidth. Error bars represent the 95% confidence intervals of the means.

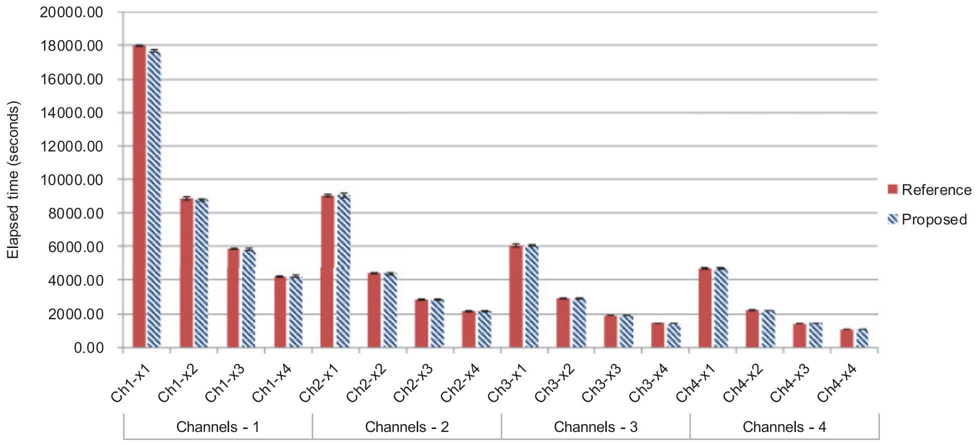

Fig. 10. Elapsed time of the postmark workload. Error bars represent the 95% confidence intervals of the means.

from the reference system. To save space, only the evaluation results with the CFQ I/O scheduler are shown in Figure 10.

4.3. Linux Kernel Source Archive Extraction and Word Count

Besides using synthetic workloads, the proposed storage device emulator is also validated using real application programs. In this test, the *ext4* file system is created on the emulated storage device and both scenarios for the *noop* I/O scheduler and the CFQ I/O scheduler are tested. The Linux kernel source archive (*linux-3.10.tar.bz2)* downloaded from kernel.org [Kernel 2013b] is first copied to the *ext4* file system and then extracted in place. Because the total size of the entire working set will be over 400MB, the contents under the *Documentation* and the *drivers* directories in the source archive are omitted. After extraction, the total number of words in the source files is then counted using the *wc* utility. The total elapsed time for extracting the kernel achieve and performing the word count is measured. All results from the proposed storage device emulator are within 2% differences to the results from the reference

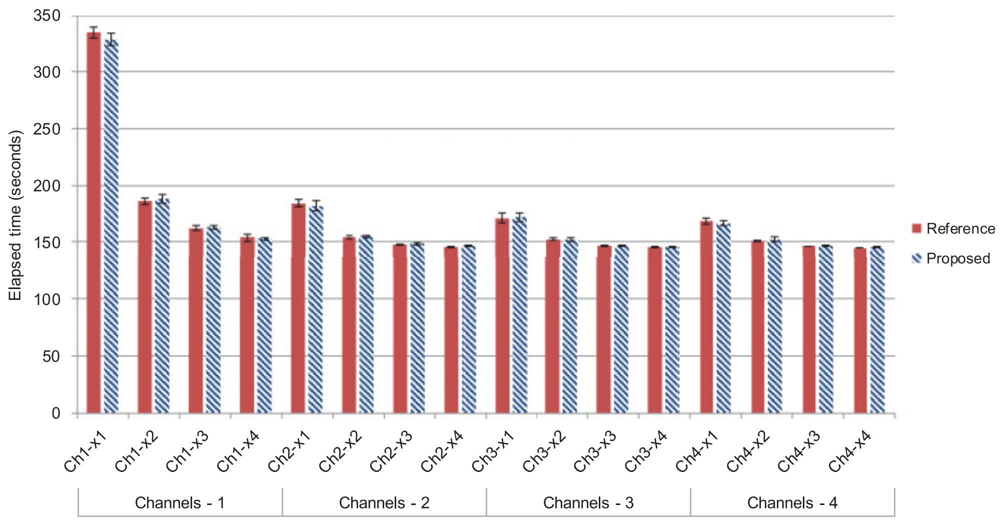

Fig. 11. Elapsed time of Linux kernel source archive extraction and word count. Error bars represent the 95% confidence intervals of the means.

system. To save space, Figure 11 only illustrates the evaluation results with the CFQ I/O scheduler.

This test scenario highlights a limitation of the reference local emulation type emulator: the size of the storage device that can be emulated is limited to the size of the available memory which can be used as the backing storage. In this test, the *Documentation* and the *drivers* directories in the kernel source archive need to be omitted so that the total working set will fit into the 400MB storage device. In comparison, the proposed storage device emulator does not have this kind of limitation. The size of the storage device that can be emulated by the proposed storage device emulator can be as large as the backing storage available on the emulator server. Moreover, because the state of the OS is paused while the backing storage is being accessed, the performance evaluation results will not be affected by the speed of the backing storage used. This means that slower backing storages can be used for emulating faster storage devices. In our experiments, the backing storage device used by the proposed storage device emulator is a traditional hard disk drive, and the proposed storage device emulator have no problem using it for emulating storage devices that have submillisecond response times.

4.4. Video Encoding and Decoding Workload

In this test, the *ext3* file system is created on the emulated storage device and both scenarios for the *noop* I/O scheduler and the CFQ I/O scheduler are tested. In the test, the raw CIF resolution *foreman* video sequence is first copied to the *ext3* file system. The raw video sequence is then encoded to the MPEG2 format using the *avconv* [Libav 2014] video encoder. After encoding, the encoded MPEG2 sequence is decoded back to the raw format using the *avconv* video decoder. The total elapsed time for encoding and decoding the video sequence is measured. All results from the proposed storage device emulator are within 2% differences to the results from the reference system. To save space, only the evaluation results with the CFQ I/O scheduler are shown in Figure 12.

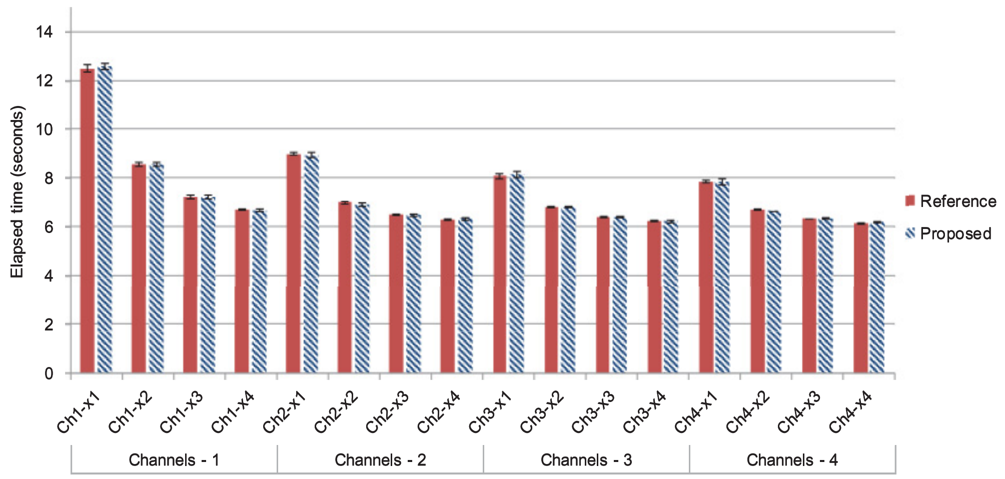

Fig. 12. Elapsed time of MPEG2 encoding and decoding the CIF resolution *foreman* sequence. Error bars represent the 95% confidence intervals of the means.

4.5. The Effect of Disk Model Complexity on Conventional Storage Device Emulators

One problem with the conventional storage device emulators is that if the time it takes to simulate the disk model exceeds the response times of the corresponding I/O request, then the emulator will fail to emulate the intended behavior of the target storage device. For example, if it takes the emulator 2ms to simulate the disk model for an I/O response, then the emulator will inevitably fail to properly emulate I/O responses that are supposed to be completed faster than 2ms. The faster the storage device is, the more likely that the emulator will fail to emulate the correct response timings for the I/O requests. In this section, we evaluate how the amount of time needed for simulating the disk model can affect the local emulation type emulator used in the reference system.

To get a sense of how much time that a typical disk model simulator would require for simulating an I/O request, we ported the DiskSim simulator version 4.0 [Bucy et al. 2008] to ZedBoard and measured its operation speed. In our measurement, it took DiskSim approximately 37 seconds to simulate 10,000 I/O operations for a disk device. That is, the amount of time required for simulating each I/O request is about 3.7ms.

To study the effect of disk model complexity on the conventional storage device emulators, we introduce artificial delays to the disk model used by the reference system to simulate disk models of different complexity. The timing delays introduced to the disk model are ranged from 0.25ms to 2ms in 0.25ms increments. The sequential read bandwidth benchmark described in Section 4.1 is used for testing. The evaluation results are illustrated in Figure 13. To save space, only the results for the single channel storage device configurations are shown. From the results, we can see that the faster the emulated storage device is, the more impact that disk model complexity will have on the reference system. For example, when emulating faster storage devices, such as the storage device with the *Ch1-x4* configuration, even a disk model processing time of just 1ms can affect the predicted performance by more than 50%.

On the contrary, in the proposed emulator, because the OS is put into the paused state while the emulator server is simulating the disk model, the evaluation results from the proposed emulator will not be affected by the complexity of the disk model. This means that disk models of arbitrary complexity can be used in the proposed emulator for modeling complex storage device behaviors.

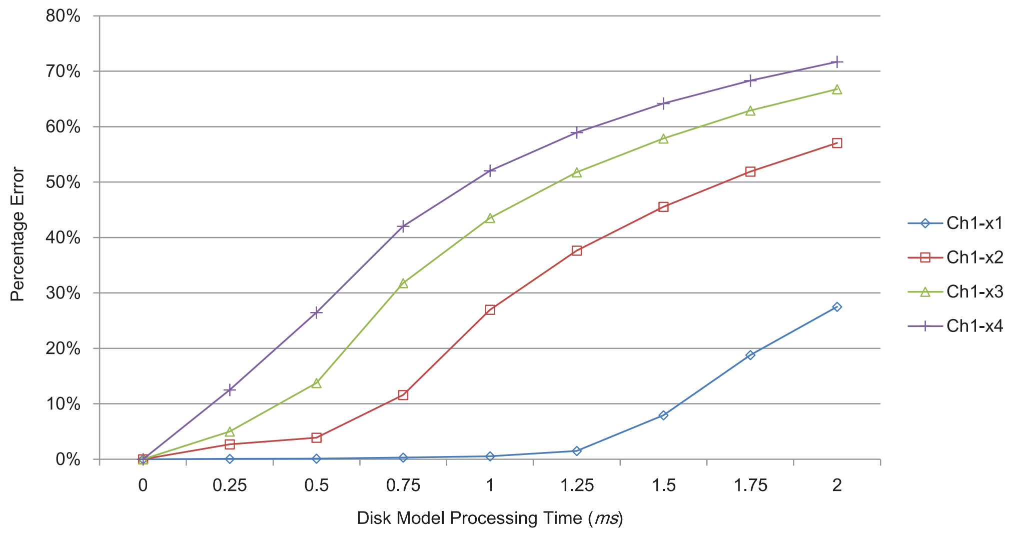

Fig. 13. In conventional storage device emulators, disk model processing time will affect the behavior of the emulated storage device and thus cause performance evaluation errors.

5. CONCLUSION

This article describes a novel OS state pausing approach to storage device emulation that can overcome many of the limitations faced by the conventional storage device emulators. In the proposed storage device emulator, the OS is made to observe a virtual system time that can be controlled by the storage device emulator. The state of the OS is paused as necessary whenever the emulator is busy processing the I/O requests of the emulated storage device. This allows the proposed storage device emulator to spend as much time as it requires for processing each I/O request without affecting the response times of the emulated storage device as perceived by the OS. Furthermore, the emulator server is run on a remote machine; therefore, the size of the storage device that can be emulated will not be limited by the resources available on the target system.

The proposed storage device emulator is implemented with the Linux OS on an embedded system development board. Experimental results show that the performances measured using the proposed storage device emulator are within 2% differences compared to the results from the reference system. The complete source code of the proposed storage device emulator will be made available to the public domain (http://www.cs.nctu.edu.tw/∼cjtsai/research/nctusde) so that the other researchers can duplicate and verify our results.

REFERENCES

Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Emulating goliath storage systems with David. *Trans. Storage* 7, 4 (February 21012), Article 12, 21 pages. DOI:http://doi.acm.org/10.1145/2078861.2078862

John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. 2008. *The DiskSim Simulation Environment Version 4.0 Reference Manual*. CMU-PDL-08-101, Parallel Data Laboratory, Carnegie Mellon University.

M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriquez-Rosell. 1980. A virtual machine emulator for performance evaluation. *Commun. ACM* 23, 2 (February 1980), 71–80. DOI:http://doi.acm.org/10.1145/358818.358821

Li-Pin Chang. 2010. A hybrid approach to NAND-flash-based solid-state disks. *IEEE Trans. Comput.* 59, 10 (October 2010), 1337–1349. DOI:http://dx.doi.org/10.1109/TC.2010.14

Yuan-Hao Chang, Po-Liang Wu, Tei-Wei Kuo, and Shih-Hao Hung. 2012. An adaptive file-system-oriented FTL mechanism for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.* 11, 1 (April 2012), Article 9, 19 pages. DOI:http://doi.acm.org/10.1145/2146417.2146426

In Hwan Doh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2007. Exploiting non-volatile RAM to enhance flash file system performance. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT'07)*. ACM, New York, NY, 164–173. DOI:http://doi.acm.org/10.1145/1289927.1289955

David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. 2011. Cache pirating: Measuring the curse of the shared cache. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*. IEEE Computer Society, Washington, DC, 165–175. DOI:http://dx.doi.org/10.1109/ICPP.2011.15

Gregory R. Ganger and Yale N. Patt. 1998. Using system-level models to evaluate I/O subsystem designs. *IEEE Trans. Comput.* 47, 6 (June 1998), 667–678. DOI:http://dx.doi.org/10.1109/12.689646

Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy, and Mark Heinrich. 2000. FLASH vs. (Simulated) FLASH: closing the simulation loop. *SIGARCH Comput. Archit. News* 28, 5 (November 2000), 49–58. DOI:http://doi.acm.org/10.1145/378995.379000

Thomas Gleixner and Ingo Molnar. 2006. Hrtimers - subsystem for high-resolution kernel timers. (Jan. 2006). Retrieved July 10, 2013 from https://www.kernel.org/doc/Documentation/timers/hrtimers.txt.

Thomas Gleixner and Douglas Niehaus. 2006. Hrtimers and beyond: Transforming the Linux time subsystems. In *Proceedings of the Linux Symposium*. 333–346.

Thomas Gleixner. 2007. High resolution timers and dynamic ticks design notes. Retrieved July 10, 2013 from https://www.kernel.org/doc/Documentation/timers/highres.txt.

Intel. 2013. *Intel 64 and IA-32 architectures software developer's manual. Order Number: 325462-048US.* Intel Corporation.

Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of error in full-system simulation. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 23–25. DOI:10.1109/ISPASS.2014.6844457

Sitaram Iyer and Peter Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. *SIGOPS Oper. Syst. Rev.* 35, 5 (October 2001), 117–130. DOI:http://doi.acm.org/10.1145/502059.502046

John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle. 2000. Modeling and performance of MEMS-based storage devices. *SIGMETRICS Perform. Eval. Rev.* 28, 1 (June 2000), 56–65. DOI:http://doi.acm.org/10.1145/345063.339354

John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. 2002. Timing-accurate storage emulation. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association, Berkeley, CA, 6.

Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. 2006. To infinity and beyond: time-warped network emulation. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3 (NSDI'06)*, Vol. 3. USENIX Association, Berkeley, CA, 7.

Libav. 2014. Open source audio and video processing tools. Retrieved June 2014 from http://libav.org/.

Linux Kernel. 2013a. Mainline Linux kernel source code. Retrieved from https://www.kernel.org/.

Linux Kernel. 2013b. Linux kernel archive version 3.10. Retrieved from https://www.kernel.org/pub/linux/kernel/v3.0/linux-3.10.tar.bz2.

Linux Kernel. 2014. ftrace - Function Tracer. Retrieved from https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting storage for smartphones. *Trans. Storage* 8, 4 (December 2012), Article 14, 25 pages. DOI:http://doi.acm.org/10.1145/2385603.2385607

Ying-Chieh Lee, Chin-Ting Kuo, and Li-Pin Chang. 2012. Design and implementation of a virtual platform of solid-state disks. *IEEE Embed. Syst. Lett.* 4, 4 (December 2012), 90–93. DOI:http://dx.doi.org/10.1109/LES.2012.2213795

Linaro. 2014. Linaro 13.11 nano build. Retrieved from http://www.linaro.org/downloads/1311.

Kaoutar El Maghraoui, Gokul Kandiraju, Joefon Jann, and Pratap Pattnaik. 2010. Modeling and simulating flash based solid-state disks for operating systems. In *Proceedings of the first joint WOSP /SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10)*. ACM, New York, NY, 15–26. DOI:http://doi.acm.org/10.1145/1712605.1712611

Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel Distrib. Technol.* 3, 4 (December 1995), 34–43. DOI:http://dx.doi.org/10.1109/88.473612

Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.* 7, 1 (January 1997), 78–103. DOI:http://doi.acm.org/10.1145/244804.244807

John Stultz, Nishanth Aravamudan, and Darren Hart. 2005. We are not getting any younger: A new approach to time and timers. In *Proceedings of the Linux Symposium*. 219–232.

Chandramohan A. Thekkath, John Wilkes, and Edward D. Lazowska. 1994. Techniques for file system simulation. *Softw. Pract. Exper.* 24, 11 (November 1994), 981–999. DOI:http://dx.doi.org/10.1002/spe.4380241102

Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. 2008. A nine year study of file system and storage benchmarking. *Trans. Storage* 4, 2 (May 2008), Article 5, 56 pages. DOI:http://doi.acm.org/10.1145/1367829.1367831

S. Y. Wang, C. L. Chou, and C. C. Lin. 2007. The Design and Implementation of the NCTUns Network Simulation Engine. *Simul. Model. Pract. Theory*, 15 (2007), 57–81. DOI:http://dx.doi.org/10.1016/j.simpat.2006.09.013

Emmett Witchel and Mendel Rosenblum. 1996. Embra: Fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.* 24, 1 (May 1996), 68–79. DOI:http://doi.acm.org/10.1145/233008.233025

Charles P. Wright, Nikolai Joukov, Devaki Kulkarni, Yevgeniy Miretskiy, and Erez Zadok. 2005. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'05).* USENIX Association, Berkeley, CA, 53.

Charles P. Wright and Erez Zadok. 2007. The Auto-pilot Benchmarking Suite Version 2.4. Retrieved from http://www.filesystems.org/project-autopilot.html.

Chin-Hsien Wu. 2010. A self-adjusting flash translation layer for resource-limited embedded systems. *ACM Trans. Embed. Comput. Syst.* 9, 4 (April 2010), Article 31, 26 pages. DOI:http://doi.acm.org/10.1145/1721695.1721697

Wikipedia. 2014a. Apple A7. Retrieved from http://en.wikipedia.org/wiki/Apple_A7.

Wikipedia. 2014b. Noop scheduler. Retrieved from http://en.wikipedia.org/wiki/Noop_scheduler.

Xilinx. 2013. Zynq-7000 All Programmable SoC Technical Reference Manual UG585 (v1.6.1). Xilinx.

Xilinx. 2014. The official Linux kernel from Xilinx. Retrieved from https://github.com/xilinx.

Jinsoo Yoo, Youjip Won, and Joongwoo Hwang. 2013. VSSIM: Virtual machine based SSD simulator. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13).*

ZedBoard. 2014. ZedBoard Documentations. Retrieved from http://www.zedboard.org/product/zedboard.

Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. 2013. Phase-change memory: An architectural perspective. *ACM Comput. Surv.* 45, 3 (July 2013), Article 29, 33 pages. DOI:http://doi.acm.org/10.1145/2480741.2480746

Received February 2014; revised October 2014; accepted May 2015

QUERY

Q1: AU: Please provide complete mailing and email addresses for both authors.