NV-process: A Fault-Tolerance Process Model Based on Non-Volatile Memory

Xu Li Kai Lu Xiaoping Wang Xu Zhou
National University of Defense Technology, China
{lixu, kailu, xiaopingwang, zhouxu}@nudt.edu.cn

ABSTRACT
Reliability wall is one of the most challenging problems for next generation High Performance Computing (HPC) systems. Traditional system design adopts extra fault tolerance mechanism. However, the cost of fault tolerance mechanism itself may incur huge cost, so as to decrease the utilization ratio of the HPC system. To address this problem, we present NV-process, a fault-tolerance process model based on NVRAM. NV-process instances run in a self-contained way in NVRAM, thus to survive across operating system reboot. NV-process provides an elegant way for the applications to tolerate system crashes. We implement a prototype system of NV-process based on Linux and analyze the advantages over traditional fault tolerant mechanism for future HPC applications.

Keywords
Fault tolerance, HPC, Process model, NV-Process

1. INTRODUCTION
With the increase of the system scale, next generation High Performance Computing (HPC) is suffering reliability wall [1]. That is, under the decreased Mean Time To Failure (MTTF), the cost of the fault tolerance mechanism may become the dominating part of the computing, resulting in low utilization of HPC systems. Theoretical research [2] also shows that the utilization ratio can be decreased to zero, when the fault tolerance mechanism cost is beyond the MTTF. Hence, it requires new research effort on the fault tolerance mechanism for future HPC systems.

Recently, the new Non-volatile Random Access Memory technologies (NVRAM), such as phase-change memory (PCM) [3] and memristors [4], promise large, fine-grained, fast, and non-volatile memory device for computer designers. NVRAM enables researchers to design new fault tolerance mechanism. Existing NVRAM-based fault tolerance techniques [5] take NVRAM as a fast external storage device for checkpoints. The inherent limit of these systems is that they still follow the same architecture of traditional fault tolerance model, i.e. computation with extra fault tolerance mechanism. As a result, they cannot break the reliability wall.

In this paper, we propose a fault-tolerance process abstraction based on NVRAM, called NV-process. NV-process provides the native support for fault tolerance. Traditional process is tightly coupled with the operating system (OS). In contrast, NV-process decouples processes from the OS, and processes are stand-alone instances running in a self-contained way in NVRAM. When the system is power off (no matter it is intended or not), NV-process instances reside in the NVRAM and can continue running where they left off as soon as the OS reboots. In the view of NV-process, the OS is no longer the container of processes, but the service provider for processes. NV-process enhances the traditional process model with fault tolerance support. Moreover, there is no explicit cost for fault tolerance in NV-process, so that NV-process has great potentials for solving reliability wall problem in future HPC system design.

We implement the prototype of NV-process in Linux 2.6.15 and evaluate the performance of NV-process. Our experiment results show that NV-process could make a process persistent within average runtime overhead of 18%. Our analytical results also show that NV-process has great advantages over traditional fault tolerant mechanism on the utilization of future HPC systems.

2. FAULT TOLERANCE IN HPC
Different from general-purpose computers, HPC systems have their own characteristics. Take our TianHe-1A supercomputer [6] as an example. The major applications are the scientific computing tasks. They are data-centric, and the typical operations of the processes are data load, data exchange, data processing, and

data store. Once the task is running, few interactions are required between the application process and other system components.

The fault tolerance of HPC is also data-centric. A fault tolerance mechanism must recover the task data and the corresponding system state, such as the processor state, to resume the task execution. To this end, the design goal of NV-process focuses on fault tolerance of data processing applications in HPC.

3. DESIGN

3.1 Overview

The design of NV-process is to decouple the process from the OS, thus to survive process instances across OS reboot. Traditional process model has several difficulties on this purpose, listed as follows. 1) The process management is tightly coupled with the OS, and the OS cannot access the process across reboot. 2) The physical memory space of processes and the OS is not separate and the OS re-initialization will destruct the process state. 3) A process could not persist its processor context during execution, which will induce the process in an inconsistent state if the OS crashes suddenly. Then, we address each of the problems in the follow sub-sections.

3.2 Self-contained management

Self-contained management of NV-process is to manage the process instances in a process-centric way, instead of the OS-centric way. Traditionally, the OS manages the information of processes through the process control blocks (PCB), such as process state, scheduling information, memory management information. The OS also maintains several data structures of PCBs to manage processes conveniently and efficiently, such as a double-linked list to traverse all processes, a hash table to look up a special process. These data structures cannot be accessed when the OS reboots.

NV-process provides a mechanism to loosen the coupling between the process management and the OS. In NV-process, we adopt non-volatile process system (NVPS) to make the process management self-contained. NVPS maintains a set of management structures of PCB in process-associated, but process-inaccessible memory and the OS is granted to access these data structures. Because the data structures of NVPS are stored in process-associated memory and they could survive across the OS re-initializations. With the help of NVPS, the OS could put processes in new schedule queues to restore the execution of processes after reboot.

3.3 Independent running space

Independent running space of NV-process is to provide separate running space for the OS and the NV-process instances. Current memory organization mechanism separates virtual memory into system and process space, and relies on page tables to distinguish the physical memory of user process space and system space. On the one hand, the OS re-initializes its page table during rebooting; on the other hand, it cannot fetch page tables of user processes. Thus, the OS could not recognize the physical memory region of processes and reserve process data during rebooting. After the OS reboots, the states of processes are destructed and processes cannot be resumed. In order to resume a process successfully, we must ensure the completeness of process state across the OS reboots.

We propose independent virtual and physical (IVP) memory organization mechanism for the OS. On the one hand, IVP separates the physical memory space of the OS and that of user processes. On the other hand, IVP guarantees the mapping between physical and virtual address of a process to be the same across the OS reboots. The separation of physical space endows independent running space for processes and guarantees that the OS will not destruct the states of processes during rebooting. The persistence of memory mappings guarantees the state of a process could be accessed through the same logical address after the OS reboots.

3.4 Consistency-preserved execution

Consistency-preserved execution of NV-process is to make consistent process state persistent across OS reboot. The state of a process is composed of application, system and processor context. If the OS reboots suddenly, the processor context is clearly lost, thus the context consistency is broken. In order to resume a process, we must maintain the consistent contexts of a process in NVRAM. Current works mainly adopt process snapshot to achieve state consistency, but this technique introduces the cost of copying the whole process data.

To achieve higher efficiency, we propose a consistent execution model to make a process persistent. First, the execution model guarantees that the contexts of a process are consistent in NVRAM during the process executing. Second, the execution model allows a process to make its consistent contexts persistent in place.

3.5 In-place restart

In-place restart is to resume NV-process instances when the OS boots up. Currently, there are two approaches to start a process. The first is the creation approach, which creates a new process by loading an executable file. The second is the restart approach, which restarts a process from a checkpoint file. NV-process is very different from both the executable file and checkpoint file. We propose in-place restart to resume the execution of NV-process instances using the same memory space. In-place restart only needs to recognize the process instances and continue the execution of them. Therefore, there is no additional cost of copying process state for NV-process restart.

4. IMPLEMENTATION

Our prototype system is implemented based on Linux 2.6.15. This section describes the implementation details of NV-process.

4.1 IVP

IVP provides NV-region, a continuous physical memory space in NVRAM, as the independent running space of processes. In Linux, physical memory pages are managed through the zone mechanism and every zone is composed of page frames. There are three memory zones in original Linux: 1) ZONE_DMA, 2) ZONE_NORMAL, and 3) ZONE_HIGHMEM. A page frame is allocated from a particular memory zone through a special flag. We add a new zone “ZONE_NV” into the kernel as NV-region and a new flag “GFP_NV” to indicate that pages should be allocated from “ZONE_NV”. Data structures of NV-process instances are all allocated from NV-region.

In Figure 1, we show the layout of both virtual memory and physical memory space of NV-region. The data of NVPS include process metadata (NVPS management structures and PCBs) and application state. The process metadata and NV-region’s management data are mapped into the virtual address space of the OS, which enables the OS to manage NV-region and processes transparently. The application state of a process is mapped into the process user space.

4.2 NVPS

NVPS implements the self-contained management of processes and ensures the OS could access processes across reboot. NVPS maintains all management structures and processes in NV-region that IVP provides. The structure of NVPS consists of three parts, as illustrated in Figure 2. The first part is the NVPS entry, which is a predefined address in the OS. The second part is the management data structure of PCBs, which maintains all PCBs of processes in a list. The third part

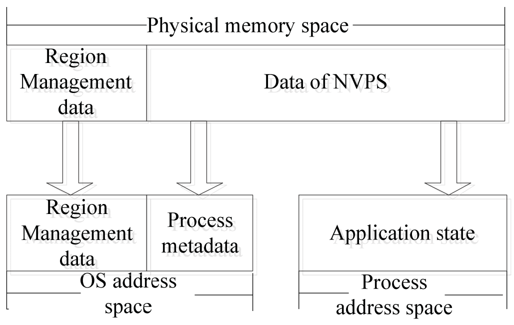

Figure 1: **NV-region memory space layout.**

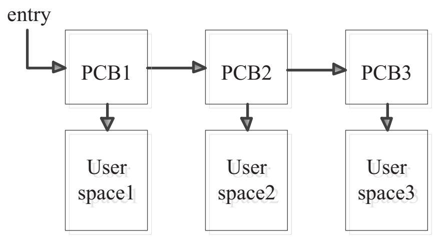

Figure 2: **The structure of NVPS.**

is the process memory space, which could be accessed through the memory map information in PCB. NVPS provides interfaces of *Register* and *Delete* for the OS to access NVPS. *Register* adds a new process in NVPS, and *Delete* removes a process from the NVPS.

4.3 State-consistent execution

Currently, we adopt transactional execution mode to maintain the consistent state during running. The transactional execution mode refers to that the execution of a process is composed of a series of transaction segments. Every transaction has ACID properties and the state of a process in NVRAM is consistent when each segment finishes. The transactional execution consists of a transactional programming model and transaction runtime. The programming model is based on C language and transaction runtime is based on TL2. The detail programming rules are shown as follow:

1) A function could be put into a transaction or out of a transaction. If a function is in a transaction, there should be no transaction in the function. If a function is out of transactions, the function is composed of transactions and control-flow statements.

2) A segment of sequential code could be put into a transaction or several transactions freely.

3) There are two choices for a segment of branch code: (a) to put the branch statement out of a transaction, then the body of the branch is a segment of sequence code, and (b) to put the whole segment of a branch in a transaction.

4) We deal with loop in the same way with the branch code.

4.4 Process resumption

Process resumption implements in-place restart for the NV-process instances. It consists of two phases: (1) the restart phase, which enables the OS to schedule a persistent process to run, and (2) the resumption phase, which resumes the execution of a process from the unsuccessful transaction.

In order to restart a process, NV-process 1) registers the process in the newly started OS; 2) insures that the process could switch from kernel context to user context correctly. To insure the process switch context correctly, NV-process builds a temporary stack to protect the user stack from being destructed. After the process restarts successfully, NV-process aborts the unfinished transaction and continues to run.

4.5 Hardware support

Phase change memory (PCM) is the most viable of emerging cell technologies for NVRAM [3, 7, 8], so we prototype NV-process with the characteristics of PCM. In order to eliminate the data inconsistency caused by the hardware, we have to make some common assumptions [9, 10]. First, NV-process requires the atomicity of a write operation. That is to say, the hardware should ensure that there is no update left in an intermediate state in the case of a power failure. Second, NV-process requires that updates reach NVRAM without reordering. Commonly, the cache may reorder updates to DRAM in order to improve performance. If a process accesses NVRAM in the same way like DRAM, updates may not be consistent.

5. EVALUATION

In this section, we evaluate the runtime cost of NV-process, and study performance of NV-process on the fault tolerance.

5.1 Methodology

As no hardware platform with a PCM-based memory is commercially available yet, we use DRAM-based platform to evaluate the runtime performance of NV-process, as many studies [6, 9-11]. This is reasonable, because several researches [12, 13] show that future PCM technology will close this gap and PCM may replace DRAM as the main memory.

Because there are no off-the-shelf benchmarks to evaluate NV-process, we build a set of benchmarks from stamp-0.9.10 [14] (bayes, genome, intruder, kmeans, ssca2, vacation) and splash-2 [15] (fft, lu-c and radix). These benchmarks could be compiled into transactional and non-traditional executable files.

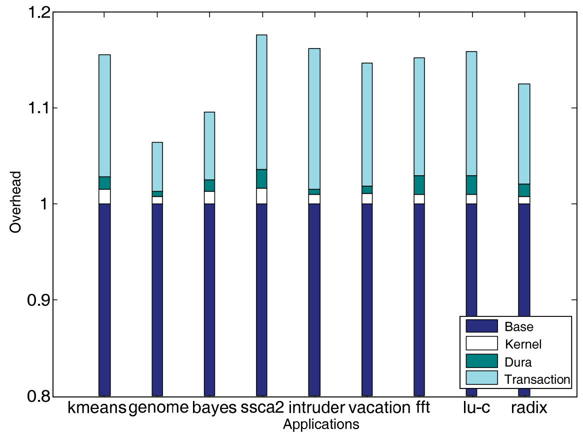

Figure 3: The detail overhead of NV-process: *For all applications, NV-process introduces a time overhead no more than 18%.*

5.2 Runtime Performance

The time overhead of NV-process consists of: (1) transactional execution, (2) durability mechanism (the atomicity of memory operations and enforcing order of updates), and (3) memory and process management mechanism of the OS kernel. To fully understand the cost of NV-process, we run our benchmarks with different configurations. In Figure 3, we show the detail overhead of NV-process. We can observe that:

(1) The overhead of kernel and durability mechanism is very small. The kernel overhead is within 1%, and the durability overhead is within 2%.

(2) The performance overhead mainly comes from transactional execution, because a transaction performs two writes to memory for every update. Additionally, transaction system must trace the allocation and free operations of every transaction to prevent memory leaks.

5.3 Fault tolerance performance

Currently, HPC systems use checkpoint-restart approach to deal with system failures. We adopt a utilization model for NV-process following the approach of Bianca Schroeder et al. [2] and study the fault tolerance performance of NV-process and CR theoretically.

5.3.1 Overhead model of transaction

The overhead of a transaction is related to the implementation of the transaction system and the amount of memory operations in a transaction. We assume that memory operations distribute uniformly for most applications. Then we build the transaction overhead model between the transaction overhead and transaction length (execution time). Our transaction model is

$$T_{overhead} = a*T + b \quad (1)$$

In equation (1), a and b depend on the implementation of a transaction system and application character-

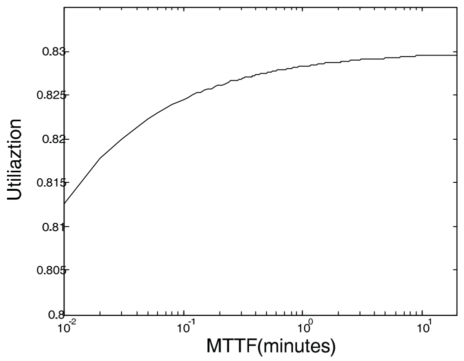

Figure 4: MTTF < 20 minutes.

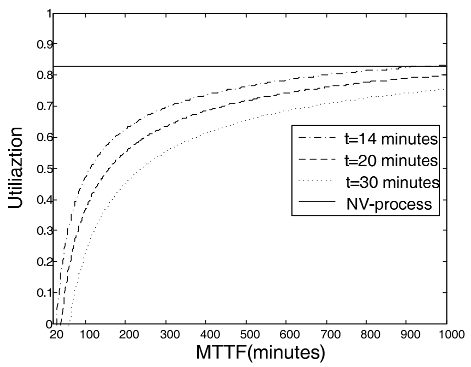

Figure 5: MTTF >= 20 minutes.

istics. We calculate the average value from our experimental applications and get that a = 0.17 and b = 0.2 μs.

5.3.2 Utilization model

Bianca Schroeder et al. [2] have proposed an approach to describe the utilization of HPC systems under different MTTF. When using checkpoint-restart techniques for fault tolerance, Bianca Schroeder et al. [2] present the utilization model is

$$U_{CR}^{o} = 1 - \sqrt{2*t} / \sqrt{MTTF} \quad (2)$$

In equation (2), t is the time of writing a checkpoint.

According to the approach of Bianca Schroeder and our transaction overhead model, we obtain the utilization model of NV-process is

$$U_{NV}^{o} = 1 - a - \sqrt{2*b} / \sqrt{MTTF} \quad (3)$$

5.3.3 Utilization comparison

We compare the fault tolerance performance of NV-process with checkpoint approaches under different MTTF. According to the study of checkpoint techniques [2], we select 14, 20 and 30 minutes as the typical checkpoint time.

In Figure 4-5, we compare the utilization of a system between checkpoint approach and NV-process. We can observe that: (1) for NV-process, when MTTF is very small, the utilization of a system increases dramatically with the increase of MTTF; when MTTF is more than 1 minute, the system utilization will keep steady, (2) for checkpoint approach, when MTTF is less than half an hour, the utilization of a system will drop to zero, and (3) the utilization of checkpoint approach could be better than NV-process when MTTF is more than 900 minutes, however, researches [16] show that the MTTF of future exascale systems will be no more than 30 minutes.

6. RELATED WORK

NVRAM. Phase change memory (PCM) is the most promising technology for building NVRAM in practice [7]. On one hand, many commercial companies launch projects to support PCM research, for example, Samsung recently announced a prototype of 8Gb PCM chip in ISSCC 2012 [17]. On the other hand, architecture researches have done a lot of efforts to close the performance gap between PCM and DRAM [12, 13, 18]. Based on these facts, we believe that NVRAM can be commercially available in the near future.

Checkpoint-restart technique. Checkpoint-restart (CR) techniques are widely used for fault tolerance in HPC [19]. CR periodically saves the computation state in stable storage, and the computation is restarted from one of these previously saved states when a failure occurs. Comparing to checkpoint-restart technique, NV-process could maintain the consistent state of process in NVRAM during runtime, and could resume a process in place. For future HPC systems, NV-process has great advantage over CR on the system utilization ratio.

NVRAM for fault tolerance. Dong et al. propose a local/global PCRAM-based incremental checkpoint scheme for HPC systems [5] and their approach could greatly improve the checkpoint performance. Comparing to their approach, NV-process does not need to copy the application state for fault tolerance. WSP [20] proposes to use flush-on-fail technique, which leverage the residual energy of the system, to flush registers and caches to NVRAM in the presence of power failure. However, WSP could not recover applications from system reboots which are induced by the OS kernel crashes.

Persistent systems. Persistent systems, such as Key-

KOS [21] and Grasshopper [22], provide recovery support for processes in kernel. Comparing to NV-process, they use snapshot approach, and could not save and resume a process in place.

7. CONLUSTION AND FUTURE WORK

In this paper, we present NV-process, a fault-tolerance process model based on NVRAM. NV-process decouples a process from the OS, and enables a process to survive across OS reboot. Our analysis shows that NV-process has great advantages over traditional fault tolerant mechanism on improving the utilization of future HPC systems.

For future work, we are interested in extending our experiments to larger platforms. Also, we would like to investigate new programming models for NV-process to reduce the cost. Finally, it would be useful to further optimize the performance of our design.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work is partially supported by National Science Foundation (NSF) China under grant numbered 61170261, 61103082, 61103193 and 61003075, and a grant from the National High-tech R&D Program of China (863 Program) numbered 2012AA01A301 and 2012AA010901.

8. REFERENCES

[1] Z. W. Xuejun Yang, Jingling Xue, Yun Zhou, "The Reliability Wall for Exascale Supercomputing," *IEEE Transactions on Computers,* 2011.

[2] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics Conference Series: SciDAC,* vol. 78, pp. 12-22, 2007.

[3] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, and H. L. Lung, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development,* vol. 52, pp. 465-479, 2008.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature,* vol. 453, pp. 80-83, 2008.

[5] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *2009 High Performance Computing Networking, Storage and Analysis (SC'09)*, 2009.

[6] X. J. Yang, X. K. Liao, K. Lu, Q. F. Hu, J. Q. Song, and J. S. Su, "The TianHe-1A Supercomputer: Its Hardware and Software," *Journal of Computer Science and Technology,* vol. 26, pp. 344-351, 2011.

[7] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *13th Workshop on Hot topics in operating systems (HotOS'11)* 2011.

[8] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development,* vol. 52, pp. 439-447, 2008.

[9] J. C. E. B. Nightingale, C. Frost, E. I. B. Lee, and D. B. D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *22nd symposium on Operating systems principles (SOSP'09)*, 2009, pp. 133-146.

[10] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *16th Architectural support for programming languages and operating systems (ASPLOS'11)*, 2011, pp. 91-104.

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *16th Architectural support for programming languages and operating systems (ASPLOS'11)*, 2011, pp. 105-118.

[12] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *36th Annual International Symposium on Computer Architecture (ISCA'09)*, 2009, pp. 14-23.

[13] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *36th Annual International Symposium on Computer Architecture (ISCA'09)*, 2009, pp. 24-33.

[14] C. C. Minh, J. W. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization (IISWC 2008)*, 2008, pp. 35-46.

[15] M. O. Steven Cameron Woo, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 Programs:Characterization and Methodological Considerations," in *22nd Annual International Symposium on Computer Architecture (ISCA'95)*, 1995, pp. 24-36.

[16] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)* 2010.

[17] I. S. Y. Choi, M-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo,, Y. R. J. Shin, C. Lee, M. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y-J. Lee, Q. Wang, S. Cha,, and H. H. S. Ahn, J. Lee, K. Kim, H. Joo, K. Lee, Y-T. Lee, J. Yoo, G. Jeong, "A 20nm 1.8V 8Gb PRAM with 40MB/s Program Bandwidth," in *ISSCC 2012*, 2012.

[18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," *Communications of the ACM,* vol. 53, pp. 99-106, 2010.

[19] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR),* vol. 34, pp. 375-408, 2002.

[20] D. Narayanan and O. Hodson, "Whole-system persistence," in *17th Architectural support for programming languages and operating systems (ASPLOS'12)*, 2012, pp. 401-410.

[21] N. Hardy, "KeyKOS architecture," *ACM SIGOPS Operating Systems Review,* vol. 19, pp. 8-25, 1985.

[22] A. Dearle, R. Di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan, "Grasshopper: An orthogonally persistent operating system," *Computing Systems,* vol. 7, pp. 289-312, 1994.