Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Versionized process based on non-volatile random-access memory for fine-grained fault tolerance*

Wen-zhe ZHANG, Kai LU†‡, Xiao-ping WANG

Science and Technology on Parallel and Distributed Processing Laboratory, College of Computer,
National University of Defense Technology, Changsha 410073, China
†E-mail: lukainudt@163.com

Received Aug. 16, 2016; Revision accepted Nov. 7, 2016; Crosschecked Feb. 15, 2018

Abstract: Non-volatile random-access memory (NVRAM) technology is maturing rapidly and its byte-persistence feature allows the design of new and efficient fault tolerance mechanisms. In this paper we propose the versionized process (VerP), a new process model based on NVRAM that is natively non-volatile and fault tolerant. We introduce an intermediate software layer that allows us to run a process directly on NVRAM and to put all the process states into NVRAM, and then propose a mechanism to versionize all the process data. Each piece of the process data is given a special version number, which increases with the modification of that piece of data. The version number can effectively help us trace the modification of any data and recover it to a consistent state after a system crash. Compared with traditional checkpoint methods, our work can achieve fine-grained fault tolerance at very little cost.

Key words: Non-volatile memory; Byte-persistence; Versionized process; Version number
https://doi.org/10.1631/FITEE.1601477 **CLC number:** TP316

1 Introduction

Due to the increasing scale and complexity, computer systems are becoming more prone to error. Current high-performance computing (HPC) systems normally have a mean-time-to-failure (MTTF) ranging from a couple of hours to hundreds of hours, e.g., 50–158 h for IBM's BlueGene/L (Adiga et al., 2002; Liang et al., 2006, 2007), 7–72 h for the Cray XT3/XT4 (Larkin and Fahey, 2007), and 6 h for TianHe-1. Fault tolerance mechanisms are introduced to guarantee the correctness of computing. For example, checkpointing is a widely adopted fault tolerance mechanism in these systems. It periodically dumps all the system states into non-volatile medium and recovers the states after a system failure or power loss. It is simple and widely adopted but introduces high overhead, especially for systems with low MTTF. In some extreme cases, all the system resources may be dedicated to making checkpoint, leaving the real system utility at zero, which is called the 'reliability wall' (Yang et al., 2012; Lu et al., 2013). Given this trend, new fault tolerance mechanisms are needed for future systems.

A recent fast-developing memory technology, non-volatile random-access memory (NVRAM), can open up some new possibilities for fault tolerance (Badam, 2013). NVRAM, such as phase change memory (PCM) (Wong et al., 2010) and memristors (Venkataraman et al., 2011), represents a kind of memory technology that shares the following features: fast access, non-volatility, byte-addressability,

‡ Corresponding author

* Project supported by the National High-Tech R&D Program (863) of China (Nos. 2012AA01A301, 2012AA010901, 2012AA010303, and 2015AA01A301), the Program for New Century Excellent Talents in University, the National Natural Science Foundation of China (Nos. 61272142, 61402492, 61402486, 61379146, and 61272483), the Laboratory Pre-research Fund (No. 9140C810106150C81001), and the Open Project of the State Key Laboratory of High-End Server & Storage Technology (No. 2014HSSA01)

ORCID: Wen-zhe ZHANG, http://orcid.org/0000-0002-8798-2195

© Zhejiang University and Springer-Verlag GmbH Germany, part of Springer Nature 2018

and large capacity (Badam, 2013). It allows the creation of data structures that are persistent directly in memory and accessible by the central processing unit (CPU) via normal load and store instructions. This feature, called 'byte-persistence', allows application and system programmers to design new and efficient fault tolerance mechanisms based on NVRAM.

Previous fault tolerance tools based on NVRAM include mainly checkpointing (Dong et al., 2011; Kannan et al., 2013) and durable transaction (Volos et al., 2011). Checkpoint tools (Kannan et al., 2013) achieve better performance by using memcpy to do checkpoint and thus to avoid the software overhead introduced by the traditional file system (Volos et al., 2014). However, these tools simply regard NVRAM as a traditional back-up storage. This type of design still cannot break through the reliability wall (Yang et al., 2012) when the checkpoint interval is small or the amount of data is large. Durable transaction tools (Volos et al., 2011) offer programming interfaces for upper applications to update persistent data in NVRAM. This calls for extra programming efforts and is not compatible with legacy code. Most importantly, all the previous checkpoint and durable transaction tools based on NVRAM are focused on only a part of the user-defined data (the heap data). There has been no design to date that covers the whole process state so that the process is fault tolerant natively on NVRAM.

In this study we propose the versionized process (VerP), a new process model based on NVRAM, which is natively non-volatile and fault tolerant. Our idea is to run a process directly on NVRAM and versionize all of its data (we give each part of the data a special version number). Every access to the data is versionized: (1) each update of the data is given a new version number; (2) each access to the data refers to the correct version number. Based on the version numbers, we can always find a consistent state of the whole process and thus recover the process in the event of a system crash or power outage. As we will show in this study, with VerP, achieving fault tolerance and recovery is extremely lightweight and introduces very little overhead (within 9% for the Scheme language (D'Amorim and Rosu, 2005)). Moreover, unlike traditional checkpoints, the VerP introduces a fixed ratio (within 9% for the Scheme language (D'Amorim and Rosu, 2005)) of overhead that will not increase with an increase in the program scale or checkpoint frequency, showing great potential for breaking through the reliability wall and achieving very fine-grained fault tolerance.

The traditional process runs directly on hardware and is limited by the architecture of hardware, which means that the traditional execution model is shaped by hardware and the major limitation is that, at any time, part of all the process states are in CPU registers which are volatile and cannot survive a system crash or power outage. To use the non-volatility of NVRAM, the VerP process model introduces an intermediate software layer between the traditional process and the underlying hardware, which enables us to put all the process states directly into memory and thus make all process states non-volatile natively. This intermediate software layer is similar to traditional dynamic interpreters or language virtual machines (Vallée-Rai et al., 2000). A Scheme language interpreter is introduced to demonstrate our idea. We argue that with the intermediate software layer, we can decouple the traditional process from the underlying hardware (see details in Section 3) and redesign the execution model of any program written in any language. Overall, the contributions of this work are listed as follows:

1. We design an operating system (OS) memory management system to manage NVRAM and offer support for upper applications to run on NVRAM directly.

2. We introduce a new process model that introduces an intermediate software layer, through which we can decouple the traditional process from the underlying hardware, run the whole process in memory (NVRAM), and versionize the whole process states for fault tolerance.

3. We propose a prototype system based on a Scheme language interpreter. Our versionized process can achieve fault tolerance with very little overhead (within 9%) and very frequent consistent updates compared with traditional checkpoint tools (more than 100x overhead with the same checkpoint interval).

Our current VerP is proposed for Scheme language and we argue that it can be generalized to any other language. Applying VerP to native C or binary code may introduce more overhead, but the overhead ratio is fixed and will not increase with an increase in the program scale or update (or checkpoint) frequency (see details in Section 3), showing great potential in breaking through the reliability wall (Yang et al., 2012).

2 Background and assumptions

2.1 Non-volatile memory and the proposed architecture

NVRAM or persistent memory (Badam, 2013) represents a kind of memory technology that has the features of byte-addressability, non-volatility, fast access, and large capacity. These features allow NVRAM to be connected with the memory bus and share the whole physical address space with dynamic random access memory (DRAM) (Bailey et al., 2011). From the view of CPU, part of the physical address is DRAM and part is NVRAM. CPU can access both memories with traditional load and store instructions. This architecture is proposed and accepted by most studies and industrial plans. Our work is based on this architecture (Fig. 1). The benefits of connecting NVRAM to the memory bus are faster memory access with the CPU cache and directly persistent data structure designed in NVRAM (Badam, 2013).

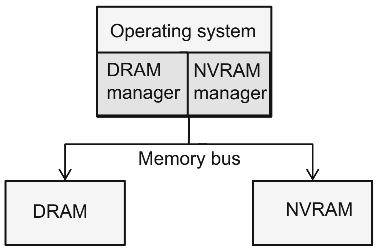

Fig. 1 NVRAM integrated architecture

PCM is a representative NVRAM technology that is rapidly maturing recently and is very likely to be released for industrial use in the near future (Wong et al., 2010). The main advantages of PCM are its high density and high read speed. The read speed of PCM is almost as high as that of DRAM. However, the write speed of PCM is lower (100–1000 ns (Coburn et al., 2011; Volos et al., 2011)). In this study, we mainly emulate PCM's write latency to show the overall performance of our proposed process model (see details in Section 5).

2.2 Assumptions

As PCM is not widely available currently, similar to previous studies (Volos et al., 2011), we make two assumptions in this study: (1) PCM should support atomic writes of 64 bits; (2) NVRAM-integrated hardware should offer a mechanism to pause execution until all the previous PCM writes have reached PCM (such as Intel's PCOMMIT). Other mechanisms to order writings include cache flushing instructions (such as clflush) and memory fence (such as mfence), which are currently available for DRAM and thus we can use them directly.

3 Design

In this section, we first describe our new process model that decouples the upper processes and the underlying hardware, which enables us to put all of the process states in NVRAM. Then we present our design to versionize all the process states.

3.1 Process model

Traditional process code runs directly on hardware (Fig. 2a), which means that the process running model is defined and limited by the hardware. For example, in the current x86 architecture, a process has stack, heap, and other CPU states (CPU registers). The stack and heap are in the main memory and can survive system crash and power cut if they are put into NVRAM. However, the stack pointer (SP, bp) and other CPU registers (program counter (PC), etc.) are in CPU and will get lost after power cut. Although we have NVRAM integrated into the underlying hardware, the traditional process model cannot support fault tolerance natively. Previous checkpoint tools (Kannan et al., 2013) dump all of the CPU states into non-volatile medium at the checkpoint time. We argue that CPU dumping is an operation imposed by the checkpoint tools rather than an operation native to the processes, which may introduce high overhead in some extreme cases where fine-grained fault tolerance is needed (e.g., doing checkpoint at every or several instructions) and thus they cannot break through the reliability wall. We have designed a new fault tolerance process model that could introduce a fixed ratio of overhead no matter how large the program scale is or how large the checkpoint frequency is.

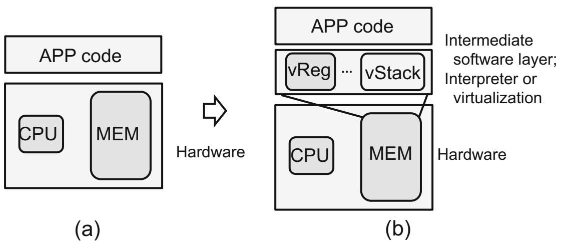

Fig. 2 VerP process model: (a) traditional process; (b) our new process model

Our new process model is shown in Fig. 2b. Between the upper processes and the underlying hardware, we introduce an intermediate software layer to decouple them. Thus, we can define freely that the VerP process model does not have to be limited by the underlying hardware. This intermediate software layer is similar to the traditional interpreters, such as the Python interpreter (Oliphant, 2007) or language virtual machines (VM), such as the Java VM (Liang and Bracha, 2000). We let the upper program be executed on our intermediate software layer and can offer some virtual resources to run the upper applications, such as the virtual CPU (vCPU) and virtual Stack (vStack) shown in Fig. 2b. Most importantly, all of these virtual resources are stored in the main memory (or NVRAM for non-volatility). For example, the Java VM keeps an PC counter for the Java byte code, and the PC counter is in memory instead of in the real CPU PC register. Above all, the intermediate software layer offers two key properties:

1. Run upper programs in an interpretive execution way. Thus, the upper programs are fully decoupled from the underlying hardware and we can redesign all the process states and the process running model. In this study, we put all the process states into NVRAM for fault tolerance. We now define the new PC (actually the line number) and the new stack pointer in the Scheme control block in memory (NVRAM). After executing each operation, the PC increases in memory, not in CPU. The stack pointer behaves in a similar way and changes only in memory. In the new process, all the volatile CPU registers are similar to fast caches for the memory states.

2. It is easy to trap every memory access to guarantee the consistency of each update or read access.

Interpretive execution is an old topic and has several advantages and disadvantages: (1) It can be adopted to execute any language (e.g., Java byte code, Python, C (Luk et al., 2005), or even binary code) and offer great flexibility in redesigning the process model. Other virtualization tools that trap-and-emulate each instruction fall within the same scope, and help any language code run on any hardware. (2) Although interpretive execution introduces more overhead than native execution, its ratio of overhead is fixed. For example, if we execute a binary program with interpretive execution, say averagely each binary instruction is executed with three more instructions due to interpretive execution, then with interpretive execution we introduce 4x overhead. Most importantly, we will always introduce 4x overhead no matter how large the program scale is. This is a very important feature that we can use to break through the reliability wall. As we will discuss in this study, to execute Scheme programs, we introduce the fixed overhead (within 9%) with increasing update (checkpoint) frequency while traditional checkpoint tools introduce over 100x overhead increasingly.

In this study, we demonstrate our idea of new fault tolerant process model based on a Scheme language interpreter (TinyScheme (Surhone et al., 2010)). Scheme is a functional programming language. We choose Scheme language because: (1) It has a simple syntax and is easy to interpret; (2) Its data management mechanism is natively suitable for fault tolerance (see details in Section 3.2). We argue that our idea can be generalized to any other language. For example, applying our idea to C or binary code may introduce more overhead, but many studies have overcome this, e.g., dynamic binary translation tools (Luk et al., 2005) and hardware support for virtualization (Uhlig et al., 2005). Applying our idea to native binary code with the help of dynamic binary translation (Luk et al., 2005) will be our future work.

3.2 Versionlized process of the Scheme language

We design our fault tolerance process model on the Scheme language. First we give a brief introduction of Scheme language and its basic interpreter framework (TinyScheme (Surhone et al., 2010)), and then present our design of the versionized Scheme process which can survive power loss natively.

3.2.1 Scheme language and basic interpreter framework

Scheme (Surhone et al., 2010) is a functional programming language and a dialect of the Lisp language (Rhodes et al., 2007). Its syntax is simple and is based on blocks. All Scheme codes have a uniform style (Fig. 3a): all codes are enclosed in blocks (the '(' and ')') and all operations are prefix operation. Fig. 3b shows a simple example of the adding operation. Like other languages, we can define variables (Fig. 3c) and functions (Fig. 3d). The blocks can be nested to achieve complex computation (Fig. 3e).

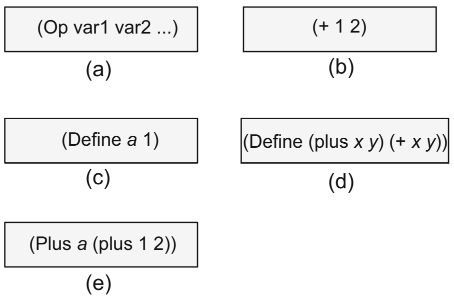

Fig. 3 Scheme code sample: (a) Scheme codes with a uniform style; (b) example of adding operation; (c) definition of variables; (d) definition of functions; (e) nesting blocks

Interpreting Scheme is simple. We read the source code and create a new stack frame (to execute a new code block) every time we encounter a '(', and when we encounter a ')', we evaluate the block and return the value.

3.2.2 Scheme data management and versionization

Automatic garbage collection and data management style: TinyScheme adopts a simple mark-and-sweep garbage collection method, which determines its data management style. As shown in Fig. 4, each piece of data is assigned a handle in a handle pool. Any access to the data should first refer to the corresponding handles and then access to the data. The handles act as proxies for all of the data. The garbage collection happens in two steps: (1) traversal accesses all data from a root point and marks all reachable data; (2) the garbage collector searches all handles and frees all unmarked data. This mark-and-sweep method is adopted widely in other languages such as some Java implementations. The point is that, the indirect data access style is completely based on pointers and could greatly help us in trapping and redirecting every memory access. We can store some meta-data information for each piece of data in its proxy (handle) to guide data access and achieve fault tolerance natively, which will be described later.

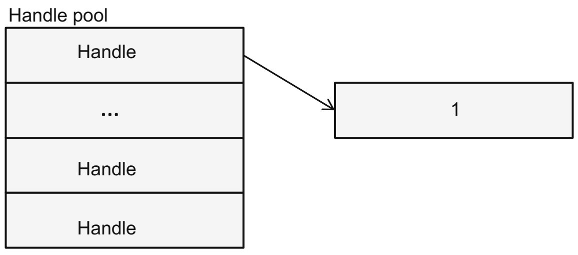

Fig. 4 Scheme garbage collection

Environment and variables: Define the variables in Scheme in two steps: define a symbol in the environment table and bind the symbol with a certain value. For example, as shown in Fig. 3c, the (define a 1) code in Scheme is organized as shown in Fig. 5a. The environment is a hash table and in the hash table we can find the symbol 'a' binded with a value '1'. The cells are the handles introduced before (Fig. 5). Thus, any reference to symbol 'a' will return the value '1'. Changing the variable's value from '1' to '2' will first create a value '2' and its handle (proxy cell), and then update the link in the environment hash table to its handle (similar to Figs. 5b and 5e).

Versionized variable updates: We introduce a mechanism to versionize all data when we execute a process on NVRAM. As shown in Fig. 5b, we introduce a global version number '1' and give each piece of data a version number in its proxy (handle). Then the update process of a variable is shown in Figs. 5c–5f. If we want to change the variable 'a' from '1' to '2', first we allocate a new piece of memory to store '2' and allocate a new corresponding handle. We give the new value '2' a version number 2, which is the current global version number plus 1 (Fig. 5c). Then before updating the variable 'a', we maintain a link to the old-version data (the red link in Fig. 5d). Thus, the old-version data will not be lost after the updating or system crash. The next step is to update the variable 'a' from '1' to '2', which is done by linking the value '2' to the variable 'a' (Fig. 5e). Finally, to concrete the update, we increase the global version number by 1 (Fig. 5f). The

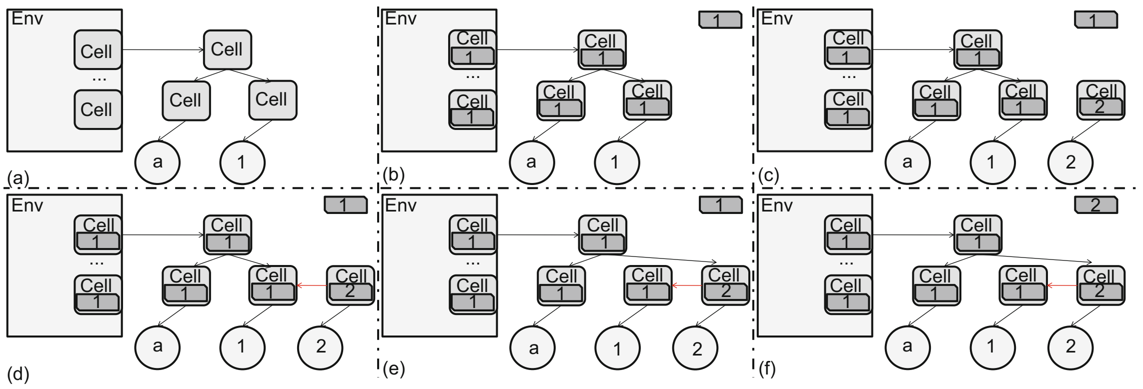

Fig. 5 Versionlized variable update: (a) organization of code; (b) introduction of a version number; (c) changing the variable 'a' from '1' to '2'; (d) maintaining a link to the old-version data; (e) updating the variable 'a' from '1' to '2'; (f) increasing the global version number by 1 (References to color refer to the online version of this figure)

entire version-number-based mechanism guarantees the consistency (atomicity) when updating any data. Under this version-based mechanism, for each access to any data the version number should be first checked to obtain the correct data. For example, if the system crashes before step (e) in Fig. 5, which increases the global version number, then after a system reboot the process state is like that in Fig. 5e. Any access to variable 'a' will first obtain the value '2' with version number 2. The version number is not equal to the global version number (which is '1', and all valid data should have a version number that is not greater than the current global version number). Thus, we know '$a = 2$' is not valid. Then we will go through the old-version link (the red link in Fig. 5e) to obtain most recent valid data, which is '1' in this case, and thus we can recover the process into a consistent state easily. As all data in Scheme is organized based on pointers and handles, we can versionize it in a similar way to achieve fault tolerance natively. The overhead is that, we need to add a link field in each handle (the cell in Fig. 5) for each piece of data to keep track of the old-version data.

Stack: The stack organization in TinyScheme (Surhone et al., 2010) is also based on pointers (Fig. 6a). The stack pointer points to the root of a binary tree. Each left child of a node is a pointer to any other data and each right child of a node points to the next subtree. Stack pushing and popping are straight forward. The popping needs to move the stack pointer to the right child (right subtree) and the pushing needs to add a new root to the tree and move the stack pointer to the new root.

Versionized stack pushing and popping: The versionized stack pushing process (Fig. 6) is similar to variable updating. First, we give each handle (the cell) a version number (Fig. 6b). The data that needs to be pushed into stack is given a new version number, which is the current global version number plus 1 (Fig. 6c). Then the new data is pushed into stack as usual (Figs. 6d and 6e). Finally we increase the global version number by plusing 1 to concrete this stack push operation. For stack popping we just need to check the version number. For example, if the system crashes before step (f) in Fig. 6, then the system state is like Fig. 6e. In this case, for stack popping the version number will be checked and we find that the first piece of data is not valid (its version number is greater than the current global version number). We can simply skip this invalid node and move the stack pointer to the next subtree and then perform popping again. The invalid node will be freed by the automatic garbage collection mechanism. Thus, it is simple to recover the stack state to a consistent state after a system crash. The versionized stack popping is a bit complex (Fig. 7). Normally, a stack pop needs to move the stack pointer to the next subtree. However, we need to track all the popped data in case of a system crash or recovery requirement. Here we introduce a special handle to achieve this (the skip node is shown in Fig. 7b). Each skip node acts as a handle in the Scheme data organization and has two pointers. One is for tracking the old-version data (the red link in Fig. 7b) and the other for linking with the rest of the sub-tree. When doing a stack pop, we first create a skip node and give it a version

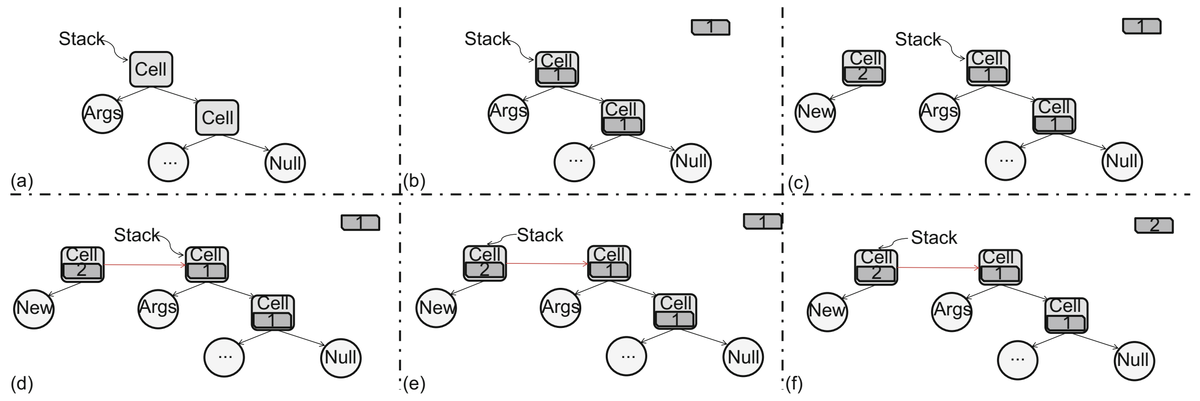

Fig. 6 Versionlized stack push: (a) stack organization based on pointers; (b) giving each handle a version number; (c) giving a new version number to data; (d) and (e): pushing new data into stack; (f) increasing the global version number by 1 (References to color refer to the online version of this figure)

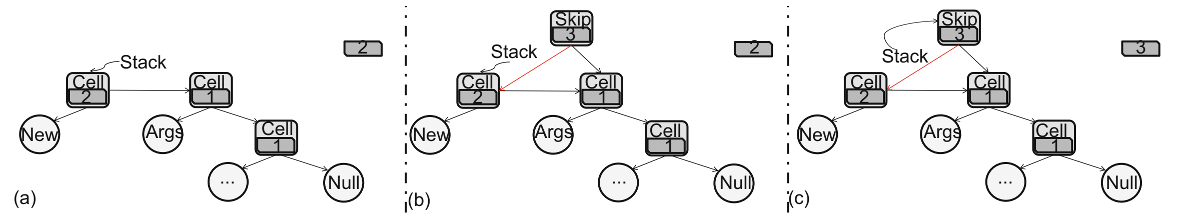

Fig. 7 Versionlized stack pop: (a) skip node with two pointers acted as a handle; (b) setting the two pointers in the skip node; (c) increasing the global version number (References to color refer to the online version of this figure)

number. The version number is the current global version number plus 1. Then we set the two pointers in the skip node properly (Fig. 7b). Finally we set the stack pointer to the skip node and increase the global version number to finish the popping operation (Fig. 7c). With the skip node in the stack, the subsequent stack pop or access is ruled as follows: check whether the current stack top is a skip node, or whether the skip node is valid (its version number should be not greater than the current global version number). If the current skip node is valid, we jump to the next subtree to perform the stack pop. Otherwise, we need to find and recover the old-version data through the skip node and access the old-version data with its version number. The overhead for this mechanism is more space for the skip node and more access time to skip the skip node. However, we will show that this does not increase the access time much in the experiments.

Other data management: Most data structures in TinyScheme are organized based on pointers and handles. Based on the indirect memory access model, it is simple to manage all the data using our versionized method to achieve fault tolerance. We need to add a version number to each piece of data and a link to keep the old-version data. For other data structures that are not based on pointers, we need to transform their data layout into a pointer-based model. A typical example is the Scheme control block shown in Fig. 8a. It is just a normal data structure that is accessed and updated normally without relying on any pointer. Our transform for the data structure is shown in Fig. 8b. We add an indirect layer for the data structure and have to rely on pointers or handles to access the data structure. We can do the same to other similar data structures and achieve fault tolerance easily. The point is that, at the intermediate software layer (the language interpreter), we have full control of the data layout and can instrument any memory access easily. Thus, we can handle any data structure in any language in this way. The main contributions are our notion of the fault tolerance process model and the idea of versionizing each piece of data. In this study, we use Scheme language to demonstrate this method. For Scheme language and other languages that rely on mark-and-sweep garbage collection (which is popular), the mechanism introduces little overhead because most of the data structures are already based on pointers. We may introduce more overhead for other languages, but as we will show in the experiment, our fault tolerance process introduces a constant ratio of overhead, which does not increase with the increase in the program scale or system complexity, and this helps break through the reliability wall.

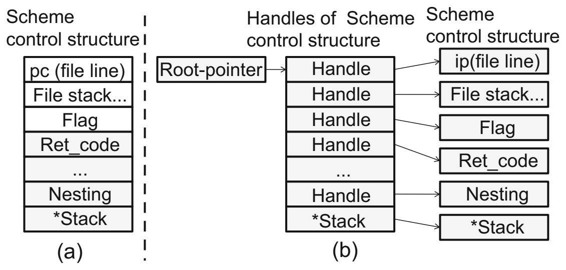

Fig. 8 Transform for other data structures: (a) Scheme control block; (b) our transform for data structure

3.3 Garbage collection

Garbage collection for old-version data is simple. Original garbage collection works by walking through all data and finding the unreachable pieces to collect (see details in Section 3.2.2). Our new garbage collection for old-version data can be easily integrated into the original mechanism. For each piece of data, it may have an old link with the old-version data (in this case it means this piece of data has been updated before). If it has an old link to the old-version data, we need to walk through the old link and collect all the old-version data except the most recent old one. We keep the most recent ones to guarantee that we can recover the state successfully after a power loss.

3.4 Recovery

As all the new process states are in memory, we give each piece of data a version number and use the version number to guarantee a consistent update of any data. This makes it easy to recover from a power loss. We need only to scan all the data and check the data's version number. If the version number is invalid, we regard this piece of data as garbage and restore the old data. The old data is always kept through a link as shown in Figs. 5–7.

3.5 Discussions

Above all, to run a process on NVRAM and to achieve fault tolerance natively, we first introduce a process running model to run all the process states in memory. Thus, we have the flexibility to organize all the process's data beyond the limitation of the underlying hardware. Then we introduce an example of interpreting the Scheme language and versionizing all its data to achieve fault tolerance with very little overhead. We argue that the fault tolerance process model or our idea can be generalized to any other language or any other hardware platform, which sheds some light on breaking through the reliability wall for future systems.

For supporting concurrency, we make the following discussions:

1. With regard to multithreading, our mechanism of versionized data can support multithreading natively: we give each thread a private buffer. Before a thread modifies any shared data, it first fetches the global version number and performs the modification in its private buffer. After the modification, it increases the global version number to validate its modification. Any further access to the updated data would go to the most recent version (either in the thread's private buffer or in the shared space). The mechanism is similar to previous work read-log-update and OPTIK (Guerraoui and Trigonakis, 2016).

2. With regard to distributing computing, it is actually 'many nodes executing many VMs'. To support fault tolerance, we build a higher level of protocol in which there is a single global version number. Consider the whole distributed system as a single process, where all the single process states are in NVRAM and there is one global version number. The situation is similar to what we have discussed in this study. However, the implementation may be complex and we leave it to our future work.

4 Implementation

In this section, we highlight the implementation details of managing NVRAM in the operating system kernel and emulating NVRAM using DRAM for experiments.

4.1 Memory management of NVRAM

The operating system should offer a mechanism for upper applications to access NVRAM directly and thus we can organize non-volatile data freely. As shown in Fig. 9, we provide a new system call in the kernel, nvmap, which is similar to the traditional system called 'mmap'. Upper applications can use nvmap to allocate a region of virtual memory (we call this the nv region) and the operating system will map this nv region to the NVRAM physical pages (Fig. 9). Thus, the upper applications can access NVRAM and run on NVRAM directly. We keep all the mapping information and the nv virtual region information in NVRAM as metadata, and this information is all non-volatile. The metadata part in NVRAM is kept in a fixed address, which is similar to some previous file system designs. Thus, after each system crash, we can use this metadata to recover all the nv regions and the related virtual-physical mapping relationship. The same mechanism is applied to previous file system designs where we adopt a journaling mechanism to avoid corruption of the metadata.

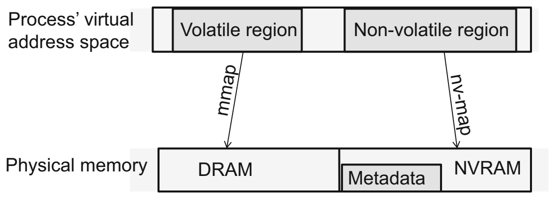

Fig. 9 NVRAM management

4.2 Emulating NVRAM

As NVRAM is not commercially available now, we use DRAM to emulate it during the experiments. As shown in Fig. 9, we set apart several DRAM pages to emulate NVRAM. At boot-up time of the operating system, when the kernel is probing the physical memory, we set the NVRAM pages to be reserved and then manage them using a free list. Then, every time the upper applications access one virtual page in the nv region and trigger a page fault, we will allocate one NVRAM physical page and construct the mapping. The metadata for this mapping is then updated accordingly. We emulate the non-volatility by dumping all the NVRAM pages to disk before a system shuts down and then copy them back after the system reboot. The read speed of NVRAM is almost the same as that of DRAM, but the write speed is slower. We emulate the latency by adding latency after every update of the non-volatile data (see details in Section 5).

5 Experiments

In this study we introduce a new process model based on NVRAM to achieve fault tolerance natively and efficiently. In this section we mainly test and show two aspects: (1) the overhead of OS management of NVRAM compared with previous NVRAM management systems; (2) the runtime overhead of our Scheme language prototype. Delivering support for other languages will be a part of our future work.

5.1 Methodology and benchmarks

1. To test and show the efficiency of our NVRAM management system, we have performed three stress tests and compared our management system to two previous NVRAM management tools: Mnemosyne (Volos et al., 2011) and PMFS. Mnemosyne and PMFS both suggest using a file system to help manage NVRAM. Mnemosyne backs each virtual memory region of NVRAM with a file (in a memory mapped way). PMFS builds a file system directly on NVRAM and optimizes the block layer of the file system. We argue that in case of making a process run directly on NVRAM, our NVRAM management is a more lightweight way for managing the virtual regions than file systems, which typically introduce much software overhead. There are three stress tests: (1) region creation—we create different virtual memory regions using nvmap and show the overhead; (2) page fault—we create an NVRAM virtual region and trigger all the page faults in this region to force allocation of physical NVRAM pages and construct all the virtual-physical mappings; (3) normal access—after an NVRAM virtual region is created and the mapping relationship has been constructed, we test the normal access overhead.

2. To test our new fault tolerance process model, we choose some popular Scheme benchmarks (Table 1). In the experiments, we increase the version number after every update operation of any process data. Thus, the process is highly fault tolerant. Any system crash could entail only a very small data loss.

Our experimental platform is an AMD server (2.2 GHz, 12-core CPU) equipped with 16 GB

Table 1 Benchmark

Benchmark	Recoverable	Version number
Tak	Y	25 881 732
Sort	Y	22 736 172
Nboyer	Y	129 483 838
Sboyer	Y	143 723 421
Conform	Y	138 472 617
Graphs	Y	142 873 822
Nfa	Y	23 727 162
Nucleic2	Y	112 392 384

DRAM running Linux kernel 3.11. We set apart 8 GB of DRAM to emulate NVRAM and emulate the write latency of NVRAM to be 100 ns (Volos et al., 2011). When we need to update the version number to make the process jump from one consistent state into another, we first flush the corresponding cache line using clflush instructions and then pause the execution for a certain interval to emulate the write latency of NVRAM (Volos et al., 2011).

5.2 Results

1. The results of the stress tests are shown in Fig. 10. We can see that our NVRAM management system (FP-Heap) performs much better than Mnemosyne and PMFS in Region_creation and Page_fault_test. The main overhead of Mnemosyne and PMFS when creating a persistent region is setting up some mapping structures with a backed file. With respect to directly managing NVRAM, we find that our work achieves better performance. Moreover, each page fault will result in complex searching for a proper physical NVRAM page and setting up the mapping relationship. PMFS optimizes the file system for non-volatile memory and performs better than Mnemosyne in Page_fault_test, while in our work (FP-Heap), the NVRAM region based on organization of non-volatile memory makes it much easier to obtain a new physical page and map the new physical page. In Access_test, we can see there is no obvious difference between FP-Heap, PMFS, and Mnemosyne. This shows that the main overhead in supporting direct access is setting up persistent regions and handling page faults, and our management system works well for this.

2. The overhead in running our fault tolerance process model with the Scheme language is shown in Figs. 11 and 12. Fig. 11 shows the runtime overhead of our versionized process (vprocess in the figure) compared to normal execution (base in the figure). In the experiments we use DRAM to emulate NVRAM and do not emulate the write latency of NVRAM. Thus, the results show the pure software overhead of our versionized process. We can see that our versionized process introduces only 2%–9% overhead by introducing the versionized mechanism, showing great potential in achieving fault tolerance with little overhead. Note that we update the global version number after each single instruction in the Scheme language to make the process jump from one consistent state to another, showing an intensive case. We also perform a checkpoint during this very short time interval for comparison. The checkpoint introduces more than 300x overhead and thus we do not show the detailed results here.

3. Fig. 12 shows the execution overhead when emulating the write latency of NVRAM. In emulating the write latency, when we need to update the version number to make the process jump from one consistent state to another, we first flush the corresponding cache line using clflush instructions and then pause the execution for a certain interval to emulate the write latency of NVRAM. In the experiments we set the write latency to be 100 ns. The results show that our mechanism introduces less than

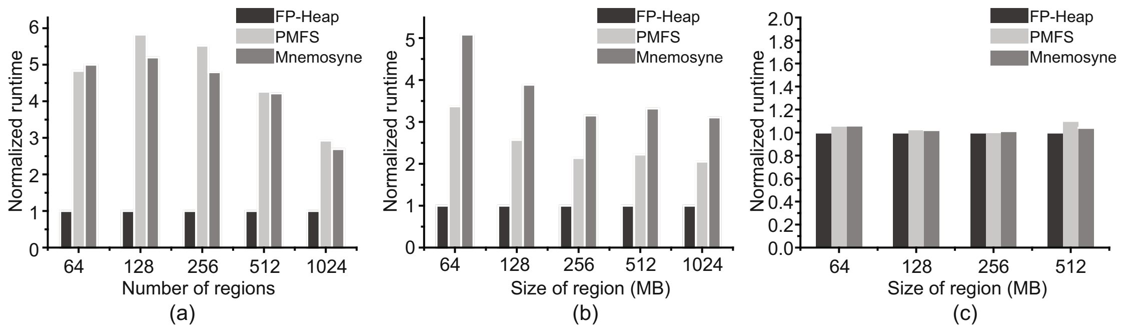

Fig. 10 Results of the stress tests: (a) region_creation; (b) page_fault_test; (c) access_test

20% overhead at this intensive update frequency. Comparing Figs. 11 and 12, we can see that the hardware overhead of NVRAM is the main source of overhead and could be further reduced by other architectural level work (Qureshi et al., 2012), and that the software overhead of our work is small and completely acceptable.

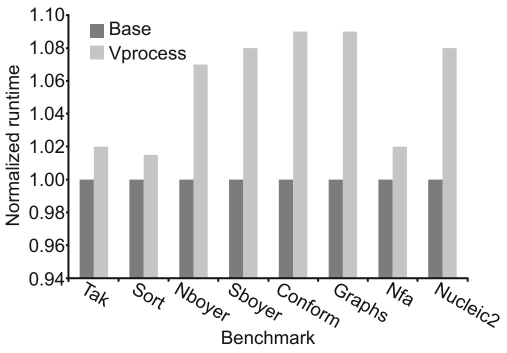

Fig. 11 Runtime overhead without write latency

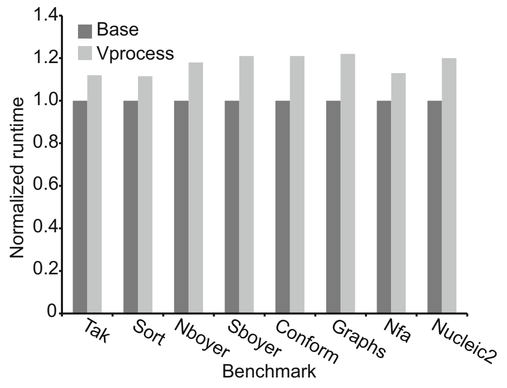

Fig. 12 Runtime overhead with write latency of 100 ns

We have done two experiments: one is without writing latency (Fig. 11) and the other is with emulating the latency of NVRAM (Fig. 12). In one experiment where we emulate the latency of NVRAM, every time after increasing the global version number, we first use the clflush instruction to flush the updated data's cache line and then emulate the latency. Thus, Fig. 12 shows two parts of the overhead: cache and latency. For the impact on cache, we argue that it is not too high because we flush only the updated data, and all other data that is read but not written does not need to be flushed. During execution, we still use the write-back cache protocol.

Above all, we show that our OS management of NVRAM is efficient and based on that, our versionized fault tolerance process model presents real advantages over traditional fault tolerance mechanisms, and our model can be generalized to other languages, which show the light to break through the reliability wall.

5.3 Comparison with C and discussions

In this study, we use Scheme to demonstrate our idea of fault tolerance process model and argue that our idea can be generalized to all other languages. To better demonstrate that our idea can be used to achieve fault tolerance under some intense scenarios, we design a simple experiment to show the advantages of our system over traditional C processes on making checkpoint to achieve fault tolerance.

We have designed and implemented a simple in-memory database system in both C and Scheme. The in-memory database system is based mainly on a vector of [key, value], where the key and value are both integer for simplicity. The database system supports two operations, get (fetch) and put (insert), which represent two main basic operations: write and read. By adjusting the ratio of get and put operations, we can actually change the write/read ratio in the program. In the C-based database system, we write-protect all pages to monitor updates to any data and then at checkpoint time we back up the modified data page. We make in-memory copy to simulate the checkpoint progress and avoid involving the overhead of dumping data to secondary storage. In the Scheme-based database system, all modifications to any data are done in a copy-on-write way with the version number, and thus at checkpoint time we need to increase the global version number. To show an intense case, we do checkpoint after every put operation. Thus, in our database system, any put operation is guaranteed to be persistent after it is returned. During the experiments, we randomly do 100 000 operations with different get/put ratios. The results are shown in Fig. 13.

The baseline in Fig. 13 is the C program executed with 0% put operation and 100% get operations, which does not update data or checkpoint. The x-axis indicates the ratio of put operations. First, at ratio 0, our Scheme program is 10x slower than the baseline and C program. This overhead comes mainly from interpreting the Scheme language. At this time, neither the baseline nor the

C program writes and thus no fault tolerance overhead is introduced. Furthermore, when we increase the put ratio to 5%, we can see a significant increase in overhead (about 10x) in the C process. This comes mainly from the page fault which is used to monitor updates. In comparison, the overhead in our Scheme program increases only a little. If we continue to increase the put ratio to 10%, we can see that our Scheme process begins to show its advantage over the C process in achieving fault tolerance. At this time the C program introduces more than 20x the overhead. We can see a clear trend that with the increase of the update ratio, the C program introduces much more overhead to achieve fault tolerance, while the Scheme process performs well. Here we do the experiments only with a put ratio from 0% to 30%, because the trend is already emerging clearly and most programs have a write ratio less than 30%.

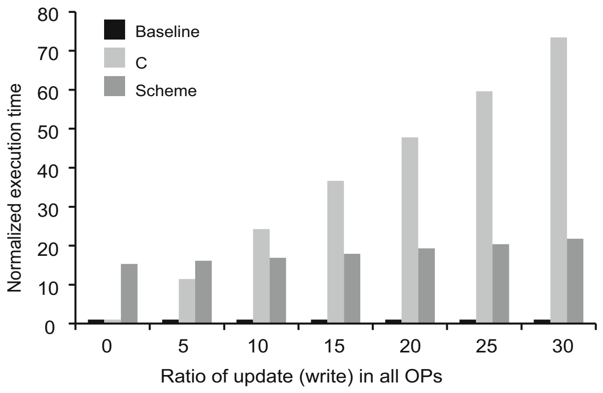

Fig. 13 Comparison with C (Baseline is the C program executed with 0% put operation and 100% get operations)

Note that we have conducted the experiment under an intense scenario but we argue that this case demonstrates well how our process model would work in the case of big data applications and database systems, or when we need high reliability. Moreover, as we will discuss further, the Scheme language can be optimized with sophisticated optimizations such as just in time compiler (JIT) in Java to close the performance gap with C.

Above all, for other languages, we make the following discussions:

1. As we have shown in the above experiments, we can compare Scheme with traditional C programs and the C programs must outperform Scheme by 10x or more. However, when it comes to achieving fine-grained fault tolerance, doing checkpoint for C programs at the same frequency as in our fault tolerance Scheme (e.g., at each update of data or at every instruction) will introduce much more overhead (more than 50x–100x). We have implemented one benchmark (the tak) in C. For the C-version tak, doing full-sized checkpoint introduces more than 200x overhead while doing incremental checkpoint introduces more than 50x . Moreover, the checkpoint for C is very sensitive to the program scale or system complexity, while our fault tolerance Scheme constantly introduces a fixed ratio of overhead. This discussion shows that even compared with C, our fault tolerance model could be adopted in some fields to achieve high reliability.

2. Our fault tolerance process model can be used in other languages with more sophisticated optimizations, such as Java. Java also designs a new process model that is different from C and runs on a Java VM. We can also alter the Java VM to change the data management and instrument every data access instruction to achieve a similar goal. Again, using Scheme is just a way to demonstrate our idea. Adopting our idea to Java or other languages is more complex but we can obtain more benefits, which will be the focus of our future work. Moreover, whether Java or Scheme, the VM-based languages represent a new fast-developing trend called 'management runtime systems', which could greatly facilitate both programming and resource management. Management runtime systems are being considered for adoption in many fields, including high-performance computing and data base. Based on this, our fault tolerance process model works perfectly for fault tolerance.

3. Scheme is not as fast as some traditional languages such as C/C++ because of its data management and mostly its interpreting running model. In other words, the Scheme process model is not directly supported by current hardware architectures, while the C/C++ process model fits the current hardware architecture well. As we have discussed in this study, the current process model does not support fault tolerance well, and thus we need a new process model that is based on an intermediate software layer. We argue that maybe in the future or in some fields where high reliability is called for, certain hardware could be modified to offer support directly. In this study, we simply demonstrate the idea of the fault tolerance process model.

6 Related work

Using NVRAM to accelerate traditional checkpoint has been proposed in many studies (Dong et al., 2011; Kannan et al., 2013; Zhang et al., 2017). They usually regard NVRAM as fast memory and use the memcpy interface to do checkpoint instead of using the file system interfaces. However, checkpoint is a traditional data-based fault tolerant mechanism and none of the previous work has shown considerations at altering the whole process model. Our work introduces a new model that can achieve fault tolerance with lower overhead compared with checkpointing.

NVRAM is a new device that researchers have been studying. Most researchers proposed to use a file system to mange it (Dulloor et al., 2014; Volos et al., 2014) and tried to optimize the block layer in the file system to better use its byte-addressability. Other researchers proposed to regard NVRAM just as DRAM and supported direct access to it (Coburn et al., 2011; Volos et al., 2011).

There have also been many studies trying to tackle the slow write problem of NVRAM at the architectural level, which could be used in our work. Introducing a DRAM buffer in front of NVRAM is a popular way to speed up NVRAM writes (Zhou et al., 2009; Kannan et al., 2013). Pre-set (Qureshi et al., 2012) improved the write speed of phase change memory by pre-doing the set operation to the initialized not-currently-using memory and thus hiding the high latency of the set operation.

7 Conclusions

We proposed the versionized the process, a new process model for fault tolerance based on NVRAM. Traditional processes run directly on hardware and thus at any given time, some of the process states are in CPU, which is volatile and cannot survive a power loss. We also proposed an intermediate software layer where we can run the new process. Then, we can execute the process all in NVRAM and make the process non-volatile natively. Based on that, we proposed a versionized mechanism to versionize every modification of the process data. Each piece of the process data is given a special version number and the version number increases with the modification of that piece of data. The version number can effectively help us trace the modification of any piece of data and recover it to a consistent state after a system crash. Compared with traditional checkpoint methods, our work can achieve fine-grained fault tolerance at very little cost (less than 20% vs. 300x in very intensive situations).

References

Adiga NR, Almasi G, Bright AA, et al., 2002. An overview of the Bluegene/L supercomputer. Proc ACM/IEEE Conf on Supercomputing, p.60. https://doi.org/10.1109/SC.2002.10017

Badam A, 2013. How persistent memory will change software systems. *Computer*, 46(8):45-51. https://doi.org/10.1109/MC.2013.189

Bailey K, Ceze L, Gribble SD, et al., 2011. Operating system implications of fast, cheap, non-volatile memory. Proc 13th Usenix Conf on Hot Topics in Operating Systems, p.2.

Coburn J, Caulfield AM, Akel A, et al., 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Comput Archit News*, 39(1):105-118. https://doi.org/10.1145/1950365.1950380

D'Amorim M, Rosu G, 2005. An equational specification for the scheme language. *J Univ Comput*, 11(7):1327-1348. https://doi.org/10.3217/jucs-011-07-1327

Dong X, Xie Y, Muralimanohar N, et al., 2011. Hybrid checkpointing using emerging nonvolatile memories for future exascale system. *ACM Trans Archit Code Optim*, 8(2), Article 6. https://doi.org/10.1145/1970386.1970387

Dulloor SR, Kumar S, Keshavamurthy A, et al., 2014. System software for persistent memory. Proc 9th European Conf on Computer Systems, p.15. https://doi.org/10.1145/2592798.2592814

Guerraoui R, Trigonakis V, 2016. Optimistic concurrency with OPTIK. ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, p.197-211. https://doi.org/10.1145/2851141.2851146

Kannan S, Gavrilovska A, Schwan K, et al., 2013. Optimizing checkpoints using NVM as virtual memory. IEEE 27th Int Symp on Parallel & Distributed Processing, p.29-40.

Larkin J, Fahey M, 2007. Guidelines for efficient parallel I/O on the cray XT_3/XT_4. Proc Cray User Group.

Liang S, Bracha G, 2000. Dynamic class loading in the Java virtual machine. *ACM SIGPLAN Not*, 33(10):36-44. https://doi.org/10.1145/286942.286945

Liang Y, Zhang Y, Sivasubramaniam A, et al., 2006. Bluegene/L failure analysis and prediction models. Int Conf on Dependable Systems and Networks, p.425-434. https://doi.org/10.1109/DSN.2006.18

Liang Y, Zhang Y, Xiong H, et al., 2007. Failure prediction in IBM Bluegene/L event logs. 7th IEEE Int Conf on Data Mining, p.583-588. https://doi.org/10.1109/ICDM.2007.46

Lu X, Wang H, Wang J, et al., 2013. Internet-based virtual computing environment: beyond the data center as a computer. *Fut Gener Comput Syst*, 29(1):309-322. https://doi.org/10.1016/j.future.2011.08.005

Luk CK, Cohn R, Muth R, et al., 2005. Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Conf on Programming Language Design and Implementation, p.190-200. https://doi.org/10.1145/1064978.1065034

Oliphant TE, 2007. Python for scientific computing. *Comput Sci Eng*, 9(3):10-20. https://doi.org/10.1109/MCSE.2007.58

Qureshi MK, Franceschini MM, Jagmohan A, et al., 2012. PreSET: improving performance of phase change memories by exploiting asymmetry in write times. 39th Annual Int Symp on Computer Architecture, p.380-391.

Rhodes C, Costanza P, D'Hondt T, et al., 2007. Lisp. Conf on Object-Oriented Technology, p.1-6.

Surhone LM, Timpledon M, Marseken SF, et al., 2010. TinyScheme. Betascript Publishing.

Uhlig R, Neiger G, Rodger D, et al., 2005. Intel virtualization technology. *Computer*, 38(5):48-56.

Vallée-Rai R, Gagnon E, Hendren L, et al., 2000. Optimizing Java bytecode using the soot framework: is it feasible? Int Conf on Compiler Construction, p.18-34.

Venkataraman S, Tolia N, Ranganathan P, et al., 2011. Consistent and durable data structures for non-volatile byte-addressable memory. Usenix Conf on File and Stroage Technologies, p.61-75. https://doi.org/10.1145/2189750.2151018

Volos H, Tack AJ, Swift MM, 2011. Mnemosyne: lightweight persistent memory. *ACM SIGARCH Comput Archit News*, 39(1):91-104. https://doi.org/10.1145/1961296.1950379

Volos H, Nalli S, Panneerselvam S, et al., 2014. Aerie: flexible file-system interfaces to storage-class memory. Proc 9th European Conf on Computer Systems, p.1-14.

Wong HSP, Raoux S, Kim SB, et al., 2010. Phase change memory. *Proc IEEE*, 98(12):2201-2227. https://doi.org/10.1109/JPROC.2010.2070050

Yang X, Wang Z, Xue J, et al., 2012. The reliability wall for exascale supercomputing. *IEEE Trans Comput*, 61(6):767-779. https://doi.org/10.1109/TC.2011.106

Zhang WZ, Kai L, Luján M, et al., 2017. Fine-grained checkpoint based on non-volatile memory. *Front Inform Technol Electron Eng*, 18(2):220-234. https://doi.org/10.1631/FITEE.1500352

Zhou P, Zhao B, Yang J, et al., 2009. A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH Comput Archit News*, 37(3):14-23. https://doi.org/10.1145/1555754.1555759