The Journal of Systems & Software 187 (2022) 111247

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

WDBT: Non-volatile memory wear characterization and mitigation for DBT systems☆

Jin Wu [a], Jian Dong [a,*], Ruili Fang [b], Wen Zhang [b], Wenwen Wang [b], Decheng Zuo [a]

[a] Harbin Institute of Technology, China
[b] University of Georgia, USA

ARTICLE INFO

Article history:
Received 11 October 2021
Received in revised form 22 December 2021
Accepted 23 January 2022
Available online 3 February 2022

Keywords:
Non-volatile memory
NVM Wear leveling
NVM Wear reduction
Cross-ISA virtualization
Dynamic binary translation
QEMU

ABSTRACT

Emerging high-capacity and byte-addressable non-volatile memory (NVM) is promising for the next-generation memory system. However, NVM suffers from limited write endurance, as an NVM cell will wear out very soon after a certain number of writes, making NVM undependable. To address this issue, many wear reduction and leveling mechanisms have been proposed. Nevertheless, most of these mechanisms are developed without the knowledge of application semantics and behaviors. In this paper, we advocate *application-level* wear management, which allows us to create effective and flexible wear reduction and leveling techniques for specific application domains. Particularly, we find that applications running with dynamic binary translation (DBT) exhibit significantly more writes. This is because DBT systems need to handle architectural differences when translating instructions across different architectures. In this paper, we present WDBT, which focuses on wear reduction and leveling for DBT systems on NVM. WDBT is designed based on common practices of DBT systems to reduce the majority of writes introduced by DBT. We also implement a prototype of WDBT using a real-world DBT system, QEMU, for multiple popular instruction sets. Experimental results on SPEC CPU 2017 show that WDBT can effectively reduce writes by 52.09% and 34.48% for x86-64 and RISC-V, respectively. Moreover, the performance overhead of WDBT is negligible.

1. Introduction

Non-volatile memory (NVM) technologies, such as phase-change memory (Qureshi et al., 2009b), spin-transfer torque magnetic random-access memory (Yu et al., 2013), and Memristor (Xu et al., 2015), promise to revolutionize the memory/storage hierarchies with many appealing features, e.g., non-volatility, byte addressability, high capacity, low latency, and low energy consumption. NVM is projected to be used as part of main memory along with traditional dynamic random-access memory (DRAM). Many software systems have been developed to take advantage of NVM, including persistent memory file systems (Xu and Swanson, 2016; Ou et al., 2016), NVM-based databases (Wang and Johnson, 2014; Hu et al., 2021), language runtimes (Wang et al., 2016a; Li et al., 2021), and persistent data structures (Friedman et al., 2020).

However, compared to DRAM, NVM has much more limited write endurance, which makes NVM undependable and hinder its adoption in the main memory system. In particular, an NVM cell may wear out after a certain number of write operations. For example, a phase-change memory can endure 10^7–10^9 writes (Qureshi et al., 2009a) and resistive memories may sustain over 10^{10} writes (Xu et al., 2015). If there is no wear management, the lifetime of an NVM device can be as short as several months (Gogte et al., 2019). In contrast, a traditional DRAM device can usually be used for many years. Therefore, it is important and imperative to design effective wear management mechanisms to make NVM dependable by prolonging its life time.

To achieve this, researchers have created many wear reduction and leveling mechanisms, ranging from hardware-based (Qureshi et al., 2009a; Chen et al., 2012; Wen et al., 2018) to pure software systems (Gogte et al., 2019; Yang et al., 2020; Akram et al., 2018). However, these techniques usually suffer from one or more limitations. For example, hardware-based wear leveling techniques usually need new hardware support and it is not clear whether they will be included into future commercial NVM products. On the other hand, software-only approaches are mostly developed without the awareness of *application semantics and behaviors*, which may lead to inflexibility and unexpected performance overhead.

In this paper, we advocate *application-level* wear reduction and leveling techniques for NVM. This can not only provide more flexibility for wear management, but also allow the management

☆ Editor: W. Eric Wong.
* Corresponding author.
E-mail address: dan@hit.edu.cn (J. Dong).

https://doi.org/10.1016/j.jss.2022.111247
0164-1212/

Table 1
Applications running with DBT (i.e., QEMU and DynamoRIO) have more memory writes than native execution without DBT. For DynamoRIO, no instrumentation is applied. For QEMU, an x86-64 instruction set is emulated.

	Native	DynamoRIO		QEMU	
	Writes	Writes	Increase (%)	Writes	Increase (%)
perlbench	4.67e+11	8.01e+11	71.4	3.83e+12	719.9
gcc	3.41e+11	4.92e+11	44.4	3.51e+12	928.9
mcf	1.78e+11	4.45e+11	150.6	2.89e+12	1528.2
omnetpp	1.94e+11	3.78e+11	94.2	2.17e+12	1016.7
xalancbmk	8.90e+10	1.64e+11	84.2	1.89e+12	2020.5
x264	2.57e+11	3.00e+11	16.7	4.95e+12	1822.2
deepsjeng	3.68e+11	5.35e+11	45.4	3.21e+12	771.8
leela	2.14e+11	4.26e+11	99.2	3.06e+12	1331.9
exchange2	4.52e+11	4.57e+11	1.0	3.67e+12	711.7
xz	6.92e+11	8.23e+11	18.9	9.80e+12	1315.6
ave	3.25e+11	4.82e+11	62.6	3.90e+12	1216.7

scheme to exploit application-specific potentials. More specifically, we focus on applications running with dynamic binary translation (DBT). This covers a broad range of important applications, such as runtime code optimizations and analyses (Dehnert et al., 2003; Hu and Smith, 2004; Feiner et al., 2012; Bruening et al., 2003), workload migration (Barbalace et al., 2017; DeVuyst et al., 2012), cross-architecture system emulation (Microsoft, 2021; Apple, 2021; Wang et al., 2020; Zhao et al., 2020, 2021), and mobile computation offloading (Yang et al., 2019; Wang et al., 2017).

A DBT system first translates executable binary code from a *guest* instruction set architecture (ISA) to a host ISA, which can be the same as or different from the guest ISA, and then executes the translated host binary code on a host physical machine to achieve functionality emulation or enhanced capability for guest code. Given the promising features of NVM mentioned before, it is anticipated that these DBT-enabled applications will also need to be "upgraded" in order to utilize NVM resources. However, existing DBT systems are not designed with the special characteristics and requirements of NVM in mind and thus may lead to unsustainable utilization of NVM.

To show the reason why it is necessary to conduct wear management for DBT system, Table 1 shows the numbers of writes for SPEC CPU 2017 benchmarks running with two representative DBT systems: DynamoRIO (Bruening and Amarasinghe, 2004) and QEMU (Bellard, 2005). The baseline is the native execution of the benchmarks without DBT. We can observe from the figure that more writes are issued when applications are running with DBT systems. In particular, QEMU introduces as high as 20x more writes compared to the native execution. The reason why DynamoRIO introduces less writes than QEMU is that it only supports same-ISA binary translation. In contrast, QEMU needs to tackle architectural differences between guest and host architectures during cross-ISA translation.

To address the above problem, we present WDBT in this paper. WDBT aims to facilitate the adoption of NVM in DBT systems by creating effective application-level wear reduction and leveling techniques to reduce extra writes introduced by DBT. Specifically, we conduct a comprehensive characterization study to understand the reasons behind the additional writes introduced by DBT systems. The study uncovers several interesting findings. For instance, we find that most of the writes are introduced by DBT to emulate guest machine states, e.g., general-purpose registers, due to the inherent differences between the guest and host architectures. Moreover, we observe that existing DBT systems deal with such differences by using a straightforward but NVM-unfriendly approach, which inevitably induces a significant amount of memory write operations. According to these findings, we design the wear reduction technique in WDBT, which leverages hardware resources provided by host architectures to realize the emulation of guest machine states. Furthermore, WDBT also periodically reallocates the host memory space that is used for the emulation to distribute writes evenly across different memory regions. Through these techniques, WDBT can effectively reduce the number of writes introduced by DBT systems and alleviate the pressure on the limited write endurance of NVM.

We have implemented a prototype of WDBT based on QEMU, which is a popular and widely-used cross-ISA DBT system. Our prototype supports x86-64 and RISC-V as the guest ISAs and AArch64 as the host ISA. To evaluate the effectiveness of WDBT, we use the SPEC CPU 2017 benchmark suite. Experimental results show that WDBT can effectively reduce writes introduced by DBT for the evaluated benchmarks. On average, the writes can be reduced by 52.09% and 34.48% for x86-64 and RISC-V guest ISAs, respectively.

We make the following contributions in this paper:

- We identify and characterize the problem of additional writes introduced by DBT systems, which can cause extra pressure on the limited write endurance of NVM.
- We present WDBT, which employs effective application-level wear reduction and leveling techniques to reduce writes in DBT systems and prolong the life time of NVM.
- We implement a prototype of WDBT based on a real-world DBT system QEMU for multiple guest and host ISAs and address several practical implementation issues.
- We evaluate WDBT using the standard benchmark suite SPEC CPU 2017. The results demonstrate the effectiveness of WDBT on reducing the writes in DBT systems.

The rest of this paper is organized as follows. Section 2 presents the background knowledge and motivates this work. Section 3 conducts the characterization study of memory writes in DBT systems. Section 4 describes the design details of WDBT. Section 5 elaborates the implementation of the prototype. Section 6 shows the experimental results. Section 7 discusses related work. And Section 8 concludes the paper.

2. Background and motivation

2.1. Background

2.1.1. Non-Volatile Memory (NVM)

Compared to the traditional main memory, such as DRAM, NVM offers the appealing persistence feature while maintaining a reasonable access latency and low power consumption. Moreover, NVM is byte addressable, which means programs can access data stored in NVM in a way similar to DRAM, without the need of a file system. Because of such advantages, NVM is becoming the major solution to construct the next-generation memory

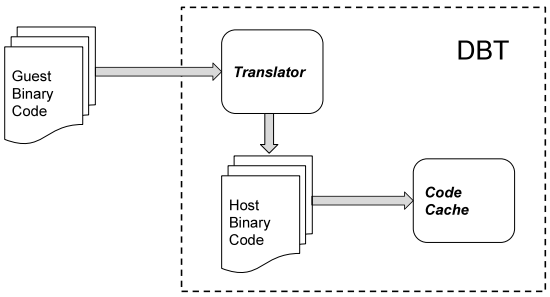

Fig. 1. The work flow of a typical DBT system.

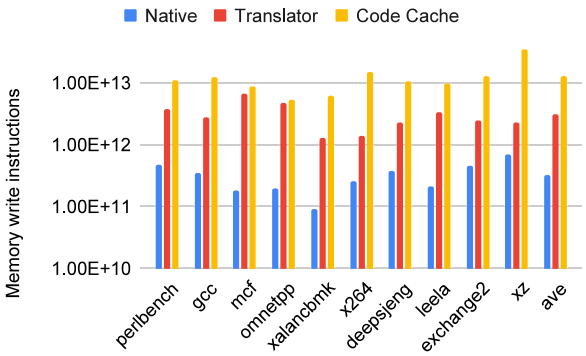

Fig. 2. Memory writes issued from the DBT translator and code cache, compared to the native execution.

system. However, NVM suffers from a fundamental limitation, i.e., the limited write endurance. Recent researches have shown that a phase-change memory (PCM) cell will permanently wear out after 10^7 to 10^8 writes (Seong et al., 2010). In an extreme case, an application can reach such a limitation within several minutes (Seong et al., 2010). As the reliability is very important to computer systems (Zhang et al., 2020; Wang et al., 2021), it is necessary and urgent to develop effective wear management schemes to prolong the life time of NVM and make it dependable.

2.1.2. Dynamic Binary Translation (DBT)

In general, a DBT system translates an executable binary from a guest ISA to a host ISA and preserves the semantics of the guest binary (Wang et al., 2018a; Song et al., 2019; Jiang et al., 2020). Fig. 1 shows the high-level workflow of a DBT system. The translation is conducted at the granularity of *basic blocks* (Wu et al., 2020). Each basic block contains a sequence of guest instructions with at most one branch instruction at the end of the block. The translated host binary code is saved to a software-managed *code cache* and reused in the following execution to mitigate the translation overhead (Wang et al., 2016b, 2018b), as a basic bock may be executed multiple times in the same run. By executing the translated host binary code, a DBT system can emulate the semantics of the input guest binary code. To this end, the execution context of a DBT system needs to be switched from the translator to the code cache. This process is often called a *context switch*. During the execution of the translated host binary code, if an untranslated basic block is encountered, the execution is then switched back to the translator to restart the translation. After all basic blocks are translated, the execution will mainly stay in the code cache until the end of the execution.

2.2. Motivation

To emulate a guest architecture that is different from a host architecture, a key technical challenge for a DBT system is how to emulate the machine states of the guest CPU, including the general-purpose registers and various status registers, e.g., condition codes. Most existing DBT systems use host memory locations to accomplish the emulation because of the differences between the guest and host architectures (Wu et al., 2021). Since the CPU state is updated frequently during the execution of a program, the host memory region that is used to emulate the guest CPU state undergoes a large amount of load and store operations.

Obviously, the above memory-based emulation scheme introduces intensive writes to a specific range of main memory on the host platform. Therefore, when porting such a DBT system to a host machine equipped with NVM as main memory, the intensive writes may cause severe and uneven wear of the NVM device. Though there are many wear reduction and leveling approaches, it is hard to apply them directly to DBT systems due to the lack of the special application semantics of DBT systems. In fact, as we will see in the next section, the writes introduced by DBT mainly stem from the translated host binary code and are executed in the code cache. Hence, it is necessary to modify the translation mechanism in existing DBT systems to mitigate the intensive and uneven wear of NVM. To this end, we propose WDBT, an application-level wear leveling and reduction solution for DBT systems to prolong the lifetime of NVM.

3. Wear characterizations of DBT systems

In this section, we conduct a study on the memory write characterizations of DBT systems. Our study is conducted on a representative DBT system, i.e., QEMU (Bellard, 2005), which is one of the most popular cross-ISA DBT system supporting multiple guest and host ISAs. Given the similarity between different translation and emulation strategies, we believe our findings uncovered in this study can also apply to other DBT systems.

3.1. Overview

There are two execution environments in a DBT system: the translator and the code cache. The translator generates semantically-equivalent host code according to the input guest code. The generated host code are stored in a software managed code cache, as it may execute multiple times at run time. During the execution, the control flow transfers between the translator and the code cache. Thus, the increased memory writes may be from the translator, the code cache, and the process of context switch.

We develop an analysis tool based on Intel Pin (Luk et al., 2005) to discover the causes of NVM wear for DBT. We instrument memory write instructions in the target DBT system, i.e., QEMU. Both the amount of the executed writes and their origins are collected by such instrumentation. In particular, we record the destination of each memory write instruction for NVM wear analysis. Fig. 2 shows the results. We present the memory writes issued from the translator and the code cache in the figure, while the writes in the native execution is provided as a reference. The result shows that an average of 92% of memory writes are issued by the host code in code cache. In particular, further study reveals that 70% of the writes in the code cache are for CPU state emulation, including emulating the general-purpose registers and condition code, as shown in Fig. 3. The study indicates that CPU state emulation consumes the majority of the memory writes. It is worth pointing out that even though the translator contributes less writes than the code cache, it still issues more writes than the native execution.

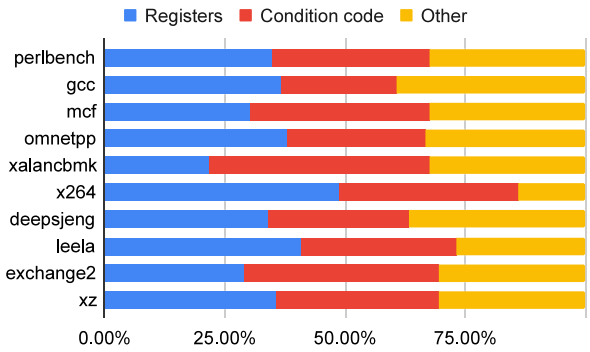

Fig. 3. The distribution of memory writes in the code cache.

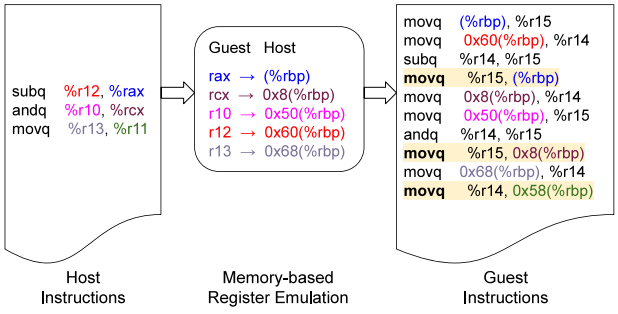

Fig. 4. An example of the translation in QEMU. The guest and host ISAs are both x86-64. Each guest register is emulated by a host memory location in the same color. Three host write instructions (highlighted) are introduced during the translation process.

3.2. Guest register emulation

To bridge the architectural gaps between the guest and host platforms, QEMU emulates guest registers in host memory locations. Fig. 4 shows a code translation example. The guest registers are stored in the main memory and indexed by the host register `%rbp`. Therefore, an access to a guest register is translated to the access to the corresponding memory location: `offset(%rbp)`. For example, the guest register `%rcx` corresponds to the host memory location 0×8(`%rbp`). It is worth pointing out that even though both the guest and host ISAs are x86-64, the memory-based translation approach still uses host memory locations to emulate guest registers. As shown in the host code, three memory write instructions (highlighted) are generated during the translation process to update the emulated guest registers. In contrast, there is no memory write in the original guest instructions.

3.3. Condition code emulation

Condition code (CC) (Wang et al., 2014) is another CPU state, which is a set of flag bits that indicate the status of the execution result of an instruction. For example, the Z flag is set if the result of an instruction is zero, while V flag indicates an overflow. CC is very important because the destination of a conditional branch relies on the status of the flags, which determines the control flow of a program. However, the architectural implementations of CC are different on popular platforms, e.g., ARM and x86 use specific registers to store the CC while RISC-V does not provide such a register. On the native platforms, the CCs are produced by the processor along with the execution of an instruction, but the DBT system needs to calculate the CCs explicitly to bridge such gaps between the guest and host architectures.

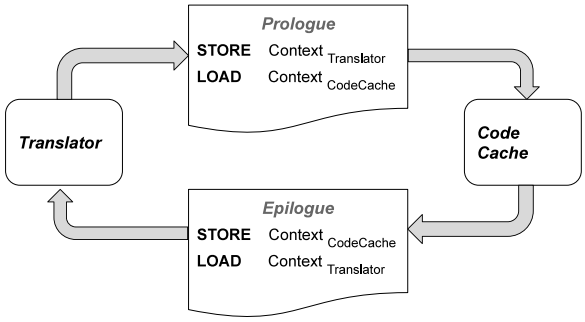

Fig. 5. Context switch between the translator and code cache.

It is worth pointing out that although CC is produced along with an instruction, it is not always used and usually overwritten by the ones generated from the following instructions. Thus, DBT system leverages the optimization of lazy evaluation for calculating CCs. It simply stores the necessary information to the memory, i.e., the opcode, source values and the result of an instruction, and only performs the evaluation of CCs when is required by a conditional instruction. For example, such optimization is implemented in QEMU for x86, m68k, cris and Sparc guests. Although this strategy improves the performance of the DBT system, it introduces additional memory write operations to the NVM, as multiple stores are generated when translating a guest instruction which affects the CCs.

Our study also shows that the memory writes introduced by such a strategy are quite common and thus take a large portion of memory writes, up to 46% and with an average of 34%, as shown in Fig. 3.

3.4. Context switch

As mentioned before, there are two execution environments in a DBT system, i.e., the translator for translating guest instructions to host instructions, and the code cache for executing the translated host instructions. During the execution, the control flow transfers between the translator and code cache as shown in Fig. 5. To ensure the correctness of each environment, context switches have to be performed to store and recover the corresponding CPU state. To this end, prologue and epilogue are generated for the context switch. Thus, additional memory writes are introduced during the process.

Table 2 shows the absolute numbers of context switches during the execution of the benchmarks in QEMU and the numbers of writes in context switches. We also include the numbers of executed basic blocks in the table as a reference. From the table, we can conclude that context switches are not triggered very frequently.

3.5. Summary

To summarize, DBT introduces a significant amount of additional memory writes, and the emulation of guest CPU state takes the largest portion of writes in the DBT system, which accounts for more than 70% of total writes. It is worth noting that the CPU state emulation uses a very small range of memory space, e.g., 128 bytes for registers and 32 bytes for condition code on the x86-64 guest platform. According to this observation, we design WDBT to conduct effective wear reduction and leveling for DBT systems on NVM.

Table 2
Statistics of context switches, memory writes in context switches, and executed blocks.

	#Context Switch	#Memory Write	#Executed BB
perlbench	1.64e6	9.83e6	5.20e11
gcc	7.90e5	4.74e6	7.04e11
mcf	3.68e3	2.21e4	4.26e11
omnetpp	3.75e4	2.25e5	2.44e11
xalancbmk	6.05e4	3.63e5	3.56e11
x264	1.12e8	6.71e8	1.74e11
deepsjeng	9.57e3	5.74e4	4.41e11
leela	9.93e3	5.96e4	3.63e11
exchange2	2.63e5	1.58e6	4.69e11
xz	1.07e4	6.41e4	1.16e12

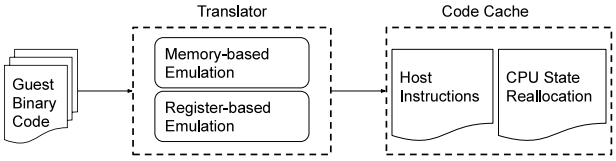

Fig. 6. WDBT enables register-based emulation and CPU state reallocation.

4. System design of WDBT

This section introduces the system design of WDBT, which creates effective wear leveling and reduction techniques to reduce the number of writes caused by DBT. It is worth pointing out that WDBT is designed as a general approach that is not limited to any specific DBT system.

4.1. Overview

Fig. 6 shows an overview of WDBT. WDBT integrates register-based emulation in the translator. So the generated host code exploits the host registers in the code cache, to emulate the guest CPU state. Therefore, the memory write pressure is mitigated compared to the memory-based emulation. In addition, dynamic reallocation of CPU state emulation is provided for wear leveling, which distributes the memory writes to more different cells rather a small set of fixed cells. The host code for the reallocation is placed in the code cache to reduce the context switch overhead. Next, we describe these two techniques in detail.

4.2. Dynamic reallocation of host memory for wear leveling

Based on our study, the additional memory writes are mostly introduced by the CPU state emulation, which causes uneven wear to the NVM. Therefore, the wear leveling technique in WDBT is designed to avoid intensive writes to a *fixed* host memory region by moving the emulated guest CPU states around through dynamic host memory reallocation.

4.2.1. Enabling dynamic allocation for guest CPU emulation

When emulating a guest CPU, DBT systems allocate the memory space for guest CPU states at the initialization stage and the location will not change during the execution. But, when porting such a DBT system to NVM, accesses to the emulated CPU state inevitably put extra pressure on the limited write endurance of NVM. Therefore, it is necessary to develop a *dynamic* memory allocation mechanism for memory-emulated guest registers. In addition, it can also collaborate with other NVM wear leveling techniques, for example, by requesting a least worn memory region through an efficient and wear-aware NVM memory allocator (Gao et al., 2013; Chen et al., 2019).

To achieve this, WDBT redesigns the translator in the DBT system and revises the translation process. Specifically, the host memory locations used for guest register emulation are allocated dynamically and accessed through a *pointer* variable, instead of a global variable. We next provide more details.

4.2.2. Reallocation of host memory regions

The aforementioned reallocation is triggered at certain points at runtime, e.g., by a dynamic counter that counts the number of executed basic blocks. WDBT generates additional host code at the end of each basic block to update and check the counter. If a preset threshold is reached, the reallocation is invoked. The counter sits in a dedicated host general-purpose register to eliminate the potential wear pressure on NVM.

The reallocation process includes three steps. First, a new host memory space is allocated for guest register emulation and the initial values of the memory space are populated by the values in the old memory space. Second, all data structures related to the emulated guest registers in the translator are updated to point to the newly allocated memory space. The host register that is used to access the emulated guest registers in code cache is also updated. Finally, the threshold of the basic block counter is reset for the next reallocation.

4.3. Leveraging host registers for wear reduction

As one of the most frequently accessed component of the CPU, the aforementioned memory-based emulation scheme for guest CPU state leads to a large amount of writes to NVM. Even if with the wear leveling technique applied, the absolute number of total memory writes is still not reduced. As an application-level approach, WDBT is able to identify such write operations in a DBT system and provide an application-specific strategy to reduce the number of writes. Specifically, WDBT leverages host general-purpose registers to emulate the guest CPU state. Fig. 7 shows the differences between the emulations using memory locations and host registers. A guest register is mapped to a dedicated host register for the emulation. Therefore, in the translated host code, write operations to the memory locations for guest register emulation are replaced with write operations to the corresponding host registers. This way, memory writes to the NVM can be reduced.

Fig. 8 uses an example to show the differences between the translation results of the memory-based and the register-based emulation schemes. With the register-based emulation scheme, an update to a guest register is translated into a write to the mapped host register. Therefore, the memory writes introduced by the memory-based emulation scheme are eliminated. In this example, the three store instructions are removed in the translated host binary code. Next, we explain how WDBT overcomes technical challenges when putting this register-based emulation scheme into practice.

4.3.1. Collaborating with memory-based emulation

There are not always sufficient host general-purpose registers available for emulating *all* guest registers. For example, x86-64 has 16 general-purpose registers while AArch64 has 32 general-purpose registers. Therefore, some guest registers may have to be emulated using host memory locations. Second, even if there are sufficient host general-purpose registers, some of them may not be used flexibly. An example is the AArch64 register 31, which has various purposes in different contexts, e.g., zero register and stack pointer.

To facilitate the collaboration between the two different emulation schemes, WDBT introduces a *flag* to indicate where the guest register is emulated. As shown in Fig. 9, in most cases, if the value is in a host register, the operation of the guest instruction can be emulated directly using the register, i.e., without the necessity to read/write host memory.

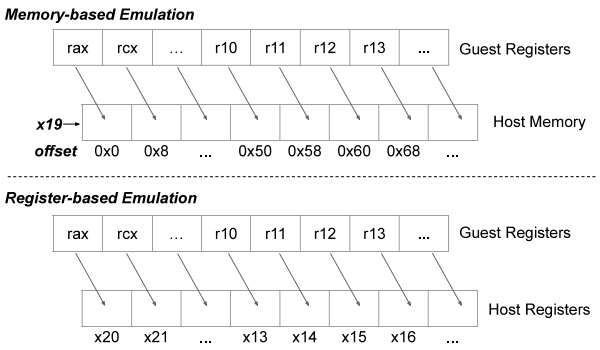

Fig. 7. The register-based emulation scheme of guest registers compared to the memory-based emulation scheme.

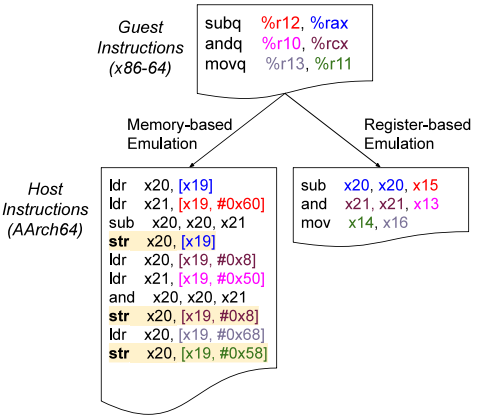

Fig. 8. The register-based emulation scheme of guest registers have less writes in the translated host binary code than the memory-based emulation scheme.

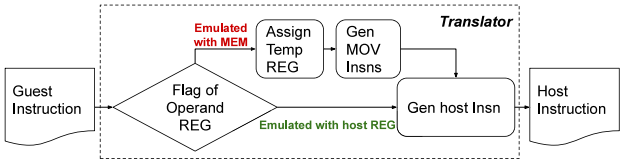

Fig. 9. WDBT collaborates with memory-based emulation.

4.3.2. Resolving host register conflicts

Host registers are limited resources of the system, and shared by both the translator and the code cache of the system. WDBT may introduce potential conflicts when using host registers for guest CPU state emulation. First, the code cache accesses host registers not only for CPU state emulation, but also for instruction semantic emulation. Therefore, registers might be accessed for different purposes, and thus data corruption may occur to the guest registers which are emulated by the host registers. To resolve this conflict, WDBT marks the host registers for guest CPU state emulation as *reserved*, which means such registers are occupied exclusively and should not be assigned for other uses. Similarly, the translator also accesses the registers. Different from the code cache, the translator is written in a high level language such as C/C++, and the instructions are generated by the compiler, so we are not able to reserve registers from the translator without noticeable performance loss. To address this issue, WDBT strengthens context switches between the translator and the code cache to store and recover the host registers used for guest register emulation. There are two cases that requires switching from the code cache to the translator. The first one is for translating the following instructions. In this case, all the guest registers have to be protected due to the large code base of the translation process. The second one is calling helper functions (Wang, 2021) from the code cache. DBT uses helper functions to emulate complex guest instructions instead of generating host code directly, to reduce the engineering efforts. So helper functions are in the translator's environment. Therefore, context switch is required when the code cache calls a helper function. This also provides us the opportunity to reduce the registers involved in the context switch.

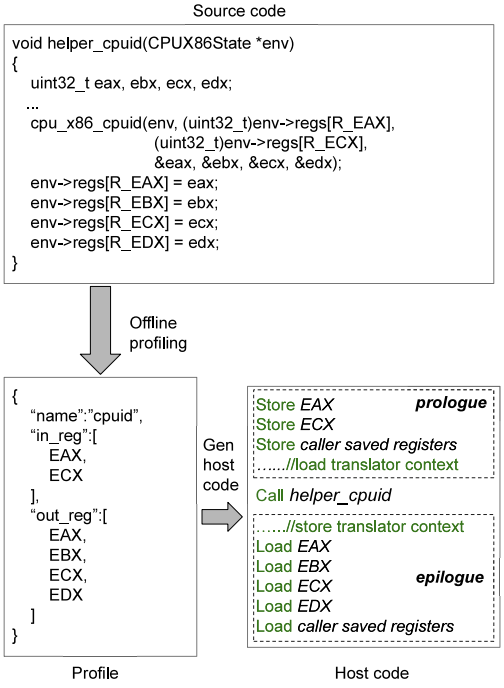

Fig. 10. The work flow of the helper function analysis tool.

To further reduce the memory writes, we develop an offline analysis tool to customize the prologue and epilogue for each helper function. Fig. 10 shows an example to demonstrate the work flow of the offline helper function analyzer. Through an offline profiling, the input and output registers are extracted from the source code. This information is leveraged by the translator to generate the prologue and epilogue for the helper function call. As shown in the figure, even with 16 guest registers mapped to the host registers, there are only two of them need to be stored and 4 of them need to be recovered during the context switch of calling *helper_cpuid*, which further mitigates the NVM wear of WDBT. This way, WDBT can resolve the conflicts and guarantee the correctness of the emulation.

4.3.3. Addressing limited host registers

As discussed before, general-purpose registers are usually limited resources of a processor. To minimize the number of writes caused by emulating guest registers, it is necessary for WDBT to further exploit hardware resources available on the host processor. To this end, WDBT leverages host vector registers or floating-point registers to hold updated values of emulated guest registers, instead of writing them back to memory. This also requires the aforementioned approaches to resolve the conflicts. By utilizing such register storage resources, WDBT can further reduce the number of write instructions in the translated host binary code.

4.3.4. Adaptive code translation

Algorithm 1: Adaptive Code Translation of WDBT

```
Input: InBB - Guest basic block to be translated
Output: OutBB - Translated host basic block
1  OutBB ← [ ];
2  foreach Guest instruction GuestInsn in InBB do
3      Operand list OpndList ← [ ];
4      Host instruction HostInsn ← NULL;
5      foreach Operand Opnd in GuestInsn do
6          if Opnd.LocationType ≠ GPRegister then
7              Opnd.Reg ←AllocTempReg();
8              if Opnd.Type = Source then
9                  HostInsn ←
10                   GenMov(Opnd.Reg, Opnd.Location);
11                 OutBB ← OutBB.Append(HostInsn);
12             end
13         else
14             Opnd.Reg ← Opnd.Location;
15         end
16         OpndList ← OpndList.Append(Opnd.Reg);
17     end
18     HostInsn ←GenInsn(Opnd.Opcode, OpndList);
19     OutBB ← OutBB.Append(HostInsn);
20     foreach Operand Opnd in GuestInsn do
21         if Opnd.Type = Destination then
22             HostInsn ←
23               GenMov(Opnd.Location, Opnd.Reg);
24             OutBB ← OutBB.Append(HostInsn);
25         end
26     end
27 end
28 return OutBB;
```

We now describe the translation process in WDBT. As discussed before, WDBT supports memory, host general-purpose register, host vector and floating point register for guest CPU state emulation. Thus, WDBT needs to translate guest binary code adaptively based on the emulation schemes of the guest registers involved in each guest instruction. Algorithm 1 shows the details of the adaptive translation process in WDBT. For guest registers emulated by host memory locations, floating-point registers, or vector registers, temporary host registers need to be allocated to load and compute the values of the emulated guest registers. Here, the function `GenMov` is used to generate data movement instructions accordingly, e.g., between temporary host registers and host memory locations (i.e., memory load and store instructions), or between temporary host registers and vector registers. For guest registers emulated by host registers, the computation can be conducted directly on the host registers without data movement. Through this adaptive translation mechanism, WDBT is able to generate correct host binary code with minimum write instructions for guest register emulation.

4.3.5. Threat to validity

WDBT may not receive considerable NVM wear mitigation on platforms with large cache configuration, when running applications with high cache hit rate. This is because the cache could absorb the majority of memory writes, and the emulated guest CPU is held in the cache most of the time. However, the NVM does not experience severe worn issue with this situation.

5. Implementation

We have implemented a prototype of WDBT based on QEMU (Bellard, 2005) (version 6.0.0), which has been widely used in many systems. Our implementation currently supports RISC-V and x86-64 as the guest ISAs and AArch64 as the host ISA. x86-64 and RISC-V have 16 and 32 general-purpose registers, respectively. In our implementation, we reserve 16 AArch64 general-purpose registers for WDBT. That is, our prototype can cover all x86-64 general-purpose registers but half of the RISC-V registers. In addition, two SIMD registers are reserved to store the condition code information for the x86-64 guest.

As a cross-ISA DBT system, QEMU is able to emulate various guest processor states. This is realized by using host memory locations to emulate guest processor states. Thus, QEMU may bring significant pressure on the limited write endurance of NVM when running on a machine with NVM.

Next, we describe the issues we encountered during the implementation of WDBT and our solutions.

5.1. Reallocation of emulated guest CPU states

In QEMU, both the translator and the generated host binary code need to access the components in the emulated guest machine state, such as general-purpose registers, stack pointer, program counter, and etc. When the host memory that is used for the emulation is reallocated, it is critical to update all related pointers in the translator and the generated host binary code. Otherwise, the execution will crash due to accessing incorrect memory locations. To avoid unnecessary writes caused by context switches, we manually implement the reallocation function in code cache in the form of *assembly code*, This allows WDBT to flexibly use host registers that are not occupied by other generated code. Moreover, this removes the requirement of context switches when reallocating host memory locations for emulated guest registers.

Once the host memory space for guest register emulation is reallocated in the code cache, it is necessary to update the pointers when the execution is switched back to the translator because the translator also needs to access the emulated guest CPU state, e.g., for interrupt handling. To this end, we implement a function in WDBT, which can be invoked by the translator to update the pointer. In addition, some helper functions in QEMU also access the emulated guest CPU state through the pointers in the translator. To guarantee the correctness of these helper functions, our implementation also updates the pointers on demand before such helper functions are invoked.

5.2. Emulating guest registers with host registers

To implement the register-based emulation scheme for guest registers, WDBT needs to cooperate with the TCG translator in QEMU. To this end, we modify the translation flow in TCG to incorporate the new emulation scheme. Besides, the implementation of context switch in QEMU is also revised to store and recover host registers used by WDBT. We next present more implementation details.

5.2.1. Modifying TCG code translation for register-based emulation

In TCG, guest registers are emulated using host memory locations, and host registers are mainly used as temporary registers to emulate the semantics of guest instructions. Since some host registers are reserved by QEMU for special purposes, e.g., passing function arguments and return values, it may introduce performance overhead if we reserve such host registers for WDBT. Hence, our implementation only touches host registers that are used by TCG as temporary registers. This allows WDBT to minimize the potential performance overhead introduced by occupying registers.

To this end, our implementation first finds a subset of host registers and marks them unavailable for TCG so that they will

not be used by TCG as temporary registers for code translation. Each of these reserved registers is then used to emulate a guest register in the translated host code. A flag variable is introduced for each guest register to indicate whether it is emulated by a host register. By checking the flag variable, the revised TCG translator is able to generate correct host code to access/update the emulated guest register. With these techniques, WDBT can be integrated into the TCG translator to generate host binary code with reduced memory writes.

5.2.2. Context switch

In QEMU, context switch is implemented as a prologue before entering the code cache and an epilogue before exiting from the code cache. The prologue is used to store the context for the translator and recover the context of the code cache. In contrast, the epilogue stores the context of code cache and recovers the context of the translator. The context mainly includes host general-purpose registers. But, not all host general-purpose registers are included because some host registers are used as temporary registers in code cache and thus can be excluded from the context.

Our implementation enhances the prologue and epilogue in QEMU to include host registers that are used by WDBT for guest register emulation. Specifically, it loads the values of the emulated guest registers from the corresponding host memory locations in the prologue and writes the values back to the memory locations in the epilogue. This guarantees the correctness of the emulation in QEMU. In addition, the memory writes caused by context switch are further reduced with the helper function analysis tool, which customizes the prologue and epilogue for each helper function.

6. Evaluation

In this section, we conduct experiments to evaluate the proposed NVM wear reduction and leveling techniques. We first provide the experimental setups and followed by the detailed evaluation results of WDBT.

6.1. Experimental setup

Table 3 shows the detailed configuration of the experimental environment.

Hardware platform. Our evaluation platform is equipped with an AArch64 RK3399 big.LITTLE processor, which includes dual-core Cortex-A72 at 2.0 GHz and quad-core Cortex-A53 at 1.5 GHz. The main memory is 4 GB LPDDR4. All the tests are performed on big cores, which have 32 KB private L1d cache for each core and 1MB unified L2 cache. As the test applications are single-threaded, only one big core is working during the evaluation. We use the main memory to simulate NVM writes, and there is no involved DRAM on the simulated platform. Though the actual access latency of NVM is longer, the total number of writes should be the same.

Software stack. The operating system is the Manjaro AArch64 version with the Linux kernel 5.7.19. The evaluation platform is exclusively occupied during our experiments to reduce the potential influence of random factors. We use integer benchmarks in the SPEC CPU 2017 benchmark suite for our evaluation. SPEC CPU 2017 is an industry-standard benchmark suite and has been widely used by many commercial companies and research projects to evaluate the performance of various computer systems. The benchmarks in the suite cover a broad range of problem domains, such as compiler, video processing, data compression, artificial intelligence, discrete event simulation, and etc. SPEC CPU 2017 is shipped with three inputs: *test*, *train*, and *reference*. Our experiments use the *reference* input, as it is the standard input for performance evaluation.

Table 3
Configurations of our evaluation platform.

	Configuration	
CPU	Rockchip RK3399	
Big cluster	ARM Cortex-A72	
	# of Cores	2
	Frequency	2 GHz
	L1d cache	32 KB
	L2 cache	1 MB
Little cluster	ARM Cortex-A53	
	# of Cores	4
	Frequency	1.5 GHz
	L1d cache	32 KB
	L2 cache	512 KB
Memory	4GB LPDDR4	
Operating system	Manjaro AArch64 with Linux-5.7.19	
Benchmark suite	SPEC CPU 2017	

Methodology. We first evaluate the benefits of wear leveling and reduction techniques in WDBT individually and then present the combined results of reduced writes for NVM. Hardware performance counter is leveraged to calculate the actual memory writes to the physical memory, while the executed memory write instructions are also collected to evaluate the memory write pressure of WDBT. On the x86-64 guest, all the 16 general-purpose registers are mapped to host general-purpose registers, and the memory locations for storing the information of condition code are mapped to two SIMD registers, with 128 bits of each. This demonstrates the effectiveness of WDBT with the highest capability of exploiting host registers for wear reduction. While on the RISC-V guest, 16 out of 32 general-purpose registers are mapped to host registers, while the remaining ones are still emulated in the main memory, as the original QEMU does. This also covers the hybrid mode of WDBT when the guest general-purpose registers are emulated in the memory and host registers. In addition, each benchmark is evaluated 5 times and their average is used as the final result.

6.2. Wear leveling

We first evaluate the wear leveling technique in WDBT. As introduced in the previous section, the memory area for CPU state emulation suffers from intensive store operations, which lead to severe uneven wear to the dedicated memory space of NVM. With the dynamic reallocation of the emulated guest CPU state during the execution of WDBT, the memory writes for CPU state emulation are distributed to more memory regions. Table 4 shows the maximum writes to a single memory location and the total number of writes to an emulated guest CPU state. Note that it takes a memory space of 160 bytes to emulate the general-purpose registers and store the condition code information on x86-64 guest, while 256 bytes to emulate the general-purpose registers on RISC-V guest.

As shown in the table, WDBT reduces both the maximum and total numbers of writes by reallocating the host memory locations used for emulating guest CPU state. With the memory space for reallocation configured as 160MB on x86-64 guest and 512MB on RISC-V guest, the total number of writes is reduced to the order of magnitude of $10^5 \sim 10^6$ from $10^{11} \sim 10^{12}$. Similarly, the maximum number of writes is reduced to the order of magnitude of $10^4 \sim 10^5$. This demonstrates the effectiveness of the wear leveling technique in WDBT to balance the wear of different NVM regions.

Table 4
Memory write reduction with guest CPU state reallocation.

	x86-64						RISC-V					
	160B		160KB		160MB		256B		512KB		512MB	
	total	max	total	max	total	max	total	max	total	max	total	max
perlbench	4.1e12	5.4e11	4.0e9	5.3e8	4.0e6	5.1e5	1.9e12	5.1e11	9.4e8	2.5e8	9.2e5	2.4e5
gcc	3.5e12	4.1e11	3.4e9	4.0e8	3.4e6	3.9e5	7.4e11	1.5e11	3.6e8	7.3e7	3.5e5	7.2e4
mcf	2.5e12	3.2e11	2.5e9	3.1e8	2.4e6	3.1e5	9.2e11	1.4e11	4.5e8	7.0e7	4.4e5	6.8e4
omnetpp	1.9e12	2.1e11	1.8e9	2.1e8	1.8e6	2.0e5	7.5e11	1.4e11	3.6e8	6.6e7	3.6e5	6.5e4
xalancbmk	2.4e12	4.0e11	2.3e9	3.9e8	2.3e6	3.8e5	6.9e11	1.7e11	3.4e8	8.5e7	3.3e5	8.3e4
x264	4.9e12	9.1e11	4.8e9	8.9e8	4.7e6	8.7e5	3.0e12	3.2e11	1.5e9	1.6e8	1.4e6	1.5e5
deepsjeng	2.0e12	3.8e11	2.0e9	3.7e8	1.9e6	3.6e5	1.5e12	3.2e11	7.3e8	1.6e8	7.1e5	1.5e5
leela	3.1e12	4.3e11	3.0e9	4.2e8	3.0e6	4.1e5	1.7e12	2.3e11	8.2e8	1.1e8	8.0e5	1.1e5
exchange2	4.5e12	7.1e11	4.4e9	7.0e8	4.3e6	6.8e5	2.0e12	4.4e11	9.9e8	2.1e8	9.7e5	2.1e5
xz	2.4e12	4.2e11	2.4e9	4.1e8	2.3e6	4.0e5	2.0e12	2.8e11	9.9e8	1.4e8	9.6e5	1.3e5

Table 5
Statistics of memory writes. "Ori" represents original QEMU, "W" represents WDBT, "wb" represents cache write back, "mr" represents memory write, and "R" represents reduction rate.

	x86-64						RISC-V					
	Ori-wb	W-wb	R (%)	Ori-mr	W-mr	R (%)	Ori-wb	W-wb	R (%)	Ori-mr	W-mr	R (%)
perlbench	1.1e10	9.5e9	17.33	6.3e12	3.2e12	48.65	1.0e10	9.0e9	13.54	3.1e12	2.4e12	22.27
gcc	1.1e10	9.8e9	14.91	2.9e12	1.4e12	51.36	1.2e10	1.1e10	5.83	1.3e12	1.1e12	18.13
mcf	2.3e10	2.1e10	9.17	7.3e12	2.3e12	68.23	2.0e10	2.0e10	1.51	1.8e12	1.3e12	32.23
omnetpp	3.8e10	2.8e10	27.76	4.7e12	2.1e12	55.23	2.9e10	2.9e10	0.66	1.5e12	1.3e12	10.40
xalancbmk	9.5e9	8.1e9	14.62	2.7e12	1.6e12	39.54	8.4e9	8.0e9	4.42	1.0e12	6.8e11	34.76
x264	4.6e9	4.3e9	6.91	6.0e12	2.6e12	55.57	4.0e9	3.7e9	8.46	3.6e12	1.5e12	58.26
deepsjeng	4.6e10	4.6e10	7.68	5.1e12	2.4e12	53.39	4.7e10	4.7e10	0.50	2.5e12	1.3e12	46.04
leela	7.1e10	4.8e9	93.26	5.6e12	2.5e12	54.67	9.4e9	6.0e9	35.97	2.5e12	1.5e12	38.65
exchange2	9.0e9	6.5e8	92.74	5.6e12	2.8e12	50.90	9.6e7	8.2e7	14.41	2.5e12	7.2e11	71.84
xz	3.8e10	3.6e10	4.54	4.7e12	2.4e12	47.81	3.5e10	3.5e10	1.83	2.5e12	8.7e11	65.51
gmean			**16.90**			**52.09**			**5.01**			**34.48**

6.3. Wear reduction

Hardware performance counter is employed to evaluate the amount of memory writes of WDBT. In general, memory writes are determined by the amount of executed memory store instructions. On cacheless systems, the actual memory writes are close to the store instructions. However, on architectures with CPU cashes, some of the writes are absorbed by the caches, and thus the memory writes performed to the physical memory is less than the executed memory store instructions. Therefore, our evaluation covers both *cacheless* architectures and architectures with caches. Specifically, we use the cache write back event to count the number of writes to main memory on systems with cache equipped, and memory access write event to count the memory writes for cacheless systems.

Table 5 shows the results of cache write backs and memory writes for both x86-64 and RISC-V guest ISAs. As shown in the table, the executed memory write instructions are reduced by 52.09% and 34.48% on average for guest x86-64 and RISC-V ISAs, respectively. This corresponds to the actual memory writes on a cacheless system. The reduction rates are less than the percentage of memory writes for CPU emulation shown in Fig. 3, because besides general-purpose registers, updates to other emulated guest CPU components, such as floating point registers still cause memory writes, which are not eliminated by WDBT. In addition, our implementation only maps 16 out of 32 guest general-purpose registers to host registers for RISC-V. Therefore, the emulation of remaining guest registers can still result in memory writes.

The cache write back data in Table 5 indicates the actual number of writes to main memory on our evaluation platform. As most write operations are absorbed by the cache, our experimental result shows less than 1% of the write instructions are actually issued to main memory. So, with 32 KB L1d cache and 1MB unified L2 cache, the memory write reduction rate is 16.90% with x86-64 as the guest ISA and 5.01% with RISC-V as the guest ISA. This shows the effectiveness of WDBT on wear reduction for NVM on platforms with caches.

In-depth studies are conducted to reveal the memory writes reduction of WDBT. The maximum write times to an individual memory location and total writes to the emulated CPU state are evaluated. In addition, we also look into the internal of the emulated CPU state to show the reduction of memory writes of each register, and the locations for condition codes on x86-64 as well.

In general, the lifetime of an NVM device is determined by the most worn cells. Therefore, we monitor the maximum number of writes to the memory locations for guest register emulation. Besides, the numbers of total memory writes are included for reference. As it is difficult to monitor the destinations of cache write back on real platforms even with hardware performance counters, we use the number of executed memory write instructions as the metric. Fig. 11 shows the results for x86-64 and RISC-V. As shown in the figure, for all evaluated benchmarks, WDBT can reduce both the maximum number of writes and the total number of writes. For the maximum number of writes of RISC-V, the reduction rate ranges from 62.31% to 90.88%, with an average of 75.41%, and for the total number of writes, the reduction rate is up to 84.79%, with an average of 75.15%. Similarly, for x86-64, WDBT reduces the maximum number of writes by 99.32% and the total number of writes by 98.54% on average. The reason why WDBT achieves higher reduction rates for the x86-64 guest than the RISC-V guest is that x86-64 has less general-purpose registers than RISC-V. Therefore, all x86-64 registers and the condition code can be emulated using AArch64 registers while only part of RISC-V registers are emulated. Overall, for both x86-64 and RISC-V guests, the reduction rates demonstrate the effectiveness of WDBT on reducing writes in DBT systems.

Next, Fig. 12 shows the number of writes caused by emulating each individual guest register in WDBT and original QEMU. In particular, the three locations for storing condition code information, i.e., `cc_dst`, `cc_src`, `cc_op`, are included on x86-64, as they

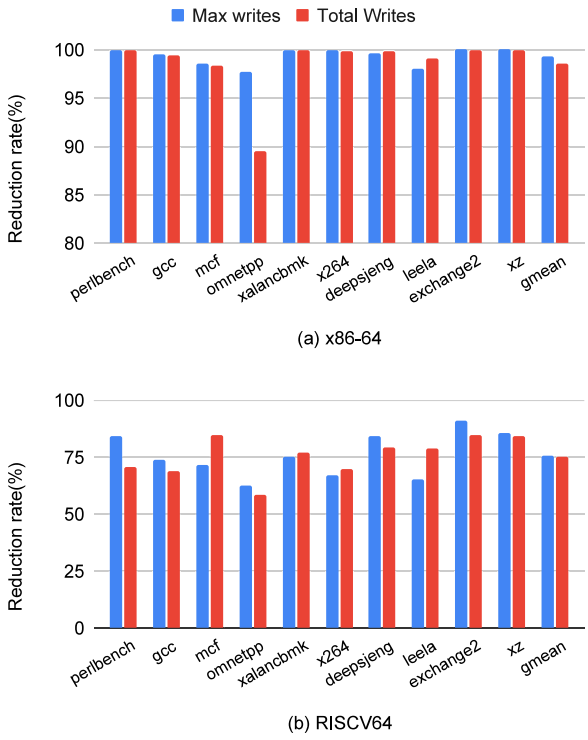

Fig. 11. WDBT reduces the maximum number of writes and the total number of writes for prolonging the lifetime of NVM. Here, RISC-V is the guest ISA.

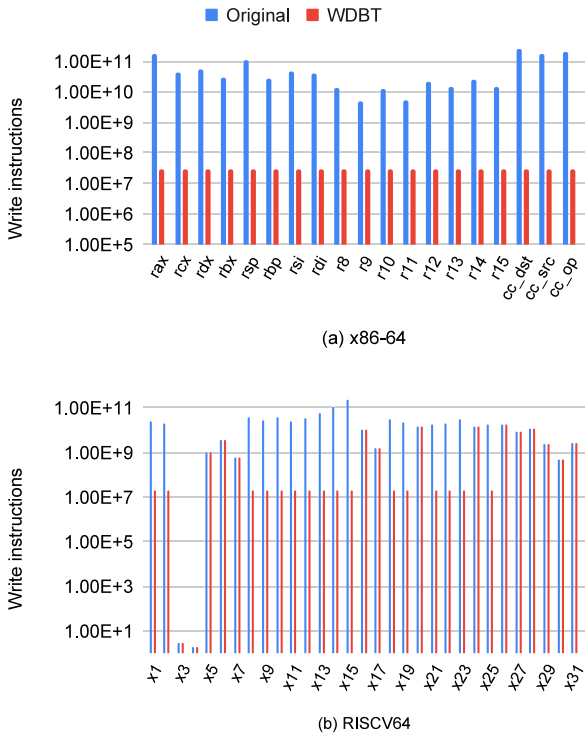

Fig. 12. The number of writes caused by emulating each individual guest register. The benchmark is *perlbench* and the guest ISA is RISC-V.

are mapped to two SIMD registers on the host AArch64 platform. Note that on the RISC-V architecture, the `x0` register is a zero register and thus excluded from the figure. Also, the `x3` and `x4` registers are global pointer and thread pointer, respectively, so

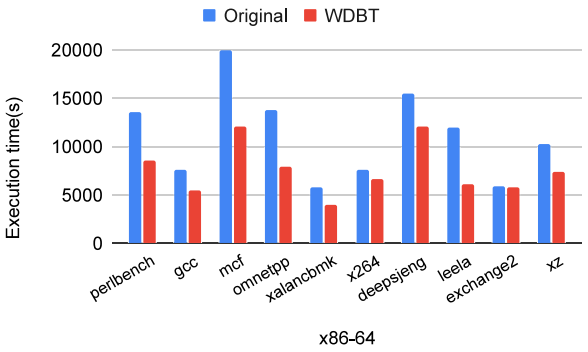

Fig. 13. Performance of WDBT.

they are rarely accessed in guest applications. Here, the benchmark is *perlbench*. As shown in the figure, WDBT successfully reduces writes for guest register emulation by mapping guest registers to host registers. The number of memory writes to guest registers which are mapped to host registers equals to the number of context switches, because the values in registers need to be written back to the memory for data synchronization. From the figure, the number of reduced writes with RISC-V guest can be as high as 11787x for a single guest register, i.e., `x15`, while 8480x of `cc_dst` with x86-64 guest. Though some registers of RISC-V guest are still emulated using host memory locations due to the limited availability of host registers, the maximum number of writes to the host memory locations for emulating guest registers is significantly reduced, from 2.3e11 on `x15` to 1.78e10 on `x26`. This shows that WDBT is helpful for reducing wear of NVM when running applications with DBT.

6.4. Overall effectiveness

The effectiveness of the combination of wear leveling and reduction is evaluated in this subsection. Table 6 shows the total number of memory write instructions executed with different configurations on x86-64 and RISC-V, respectively. The configurations of `Full` indicate the NVM wear with WDBT with both leveling and reduction approaches enabled. Compared to WDBT with wear leveling enabled only, the numbers of memory writes on x86-64 and RISC-V are further reduced by 56.78% and 23.65% on average, respectively.

6.5. Performance

With emulating CPU state in host registers, a large amount of memory load and store instructions are eliminated. Thus, the performance of WDBT benefits from the high efficiency of accessing registers. However, WDBT also introduces overhead. That is because of the additional instructions for saving and recovering contexts during the switch between the translator and the code cache. In addition, as a subset of host registers are reserved for CPU state emulation, there is a shortage of the available registers for the code cache. Thus, there are more instructions in the code cache to spill some registers to memory. These all introduce additional instructions to increase the execution time. So, it is necessary to conduct experiments to evaluate the performance of WDBT. Fig. 13 shows the results of x86-64 as the guest. The execution times of all the tested benchmark applications on WDBT are less than on the original QEMU. In particular, the speedup is up to 1.9x of `leela`, and the geometric mean of all the tested applications is 1.4x. As this evaluation is conducted on DRAM, which is faster than NVM, we believe the performance could be further improved on real NVM devices. On the other hand, for the

Table 6
Memory writes reduced by both the wear leveling and reduction techniques in WDBT, with different memory sizes. Ori represents the original QEMU, RegEmu represents host register emulated guest registers, Full represents with both host register based emulation and CPU state reallocation enabled.

	x86-64			RISC-V		
	Ori	w/ RegEmu	w/ Full	Ori	w/ RegEmu	w/ Full
	160B	160B	320MB	256B	256B	512MB
perlbench	4.1e12	1.4e12	6.6e5	1.9e12	5.7e11	2.7e5
gcc	3.5e12	1.3e12	6.4e5	7.4e11	2.3e11	1.1e5
mcf	2.5e12	8.2e11	3.9e5	9.2e11	1.4e11	6.8e4
omnetpp	1.9e12	6.2e11	3.0e5	7.5e11	3.1e11	1.5e5
xalancbmk	2.4e12	7.7e11	3.7e5	6.9e11	1.6e11	7.6e4
x264	4.9e12	6.0e11	2.9e5	3.0e12	9.1e11	4.3e5
deepsjeng	1.5e12	5.6e11	2.6e5	1.5e12	3.1e11	1.5e5
leela	3.1e12	8.4e11	4.0e5	1.7e12	3.6e11	1.7e5
exchange2	4.5e12	1.4e12	6.5e5	2.0e12	3.1e11	1.5e5
xz	1.8e12	5.4e11	2.6e5	2.0e12	3.2e11	1.5e5

RISC-V guest, the performance is close to the original QEMU. The details are omitted due to the high similarity. This demonstrates the efficiency of WDBT.

7. Related work

In this section, we discuss wear reduction and leveling techniques for NVM that are related to WDBT.

7.1. DRAM-based wear mitigation

DRAM is widely adopted by researches to mitigate the wear to NVM (Dhiman et al., 2009; Akram et al., 2018; Ramos et al., 2011; Qureshi et al., 2009b; Ferreira et al., 2010). The basic idea is to place write-intensive data in DRAM while keeping only persistent data in NVM. For example, PDRAM (Dhiman et al., 2009) proposes a hybrid DRAM and NVM memory system, which allocates pages according to a hardware-based write frequency strategy. Kingsguard (Akram et al., 2018) places read-intensive objects in NVM while write-intensive ones in DRAM for garbage collectors. RaPP (Ramos et al., 2011) ranks memory pages by a sophisticated memory controller and the top-ranked pages are migrated to DRAM. LLWB (Qureshi et al., 2009b) proposes an architectural approach to reduce write backs to NVM and combine DRAM as a layer of cache. Ferreira et al. (2010) developed a novel cache replacement policy and a swap-based wear leveling scheme to alleviate the high pressure on NVM caused by intensive writes.

Although these DRAM-based approaches can effectively reduce memory writes to NVM, they generally do not exploit application semantics and behaviors. Also, data synchronizations between DRAM and NVM may complicate the design of the systems. In contrast, WDBT solves this problem based on the characterization study of real-world DBT systems. This allows WDBT to develop application-level wear reduction and leveling techniques to maximize the potential of the techniques. Moreover, WDBT uses registers rather than DRAM as the cache layer to reduce write operations for DBT systems. This can also provide high performance efficiency as the access speed of registers is much faster than DRAM.

7.2. Reallocation-based wear leveling

The general idea of wear leveling for NVM is to distribute writes to more memory cells. Chen et al. (2012) use an age-based mechanism to guide the migration and re-mapping of frequently written pages. WAFA (Chen et al., 2019) allocates memory resources in a rotational manner for fine-grained wear leveling of NVM. Kevlar (Gogte et al., 2019) shuffles pages in NVM to avoid intensive writes to fixed cells. Security Refresh (Seong et al., 2010) is a hardware-based approach with runtime randomization to mitigate the wear-out vulnerability. Start-Gap (Qureshi et al., 2009a) moves the writes from one line to another with two reserved registers as the indicator. NVM file systems can benefit from such strategy as well, e.g., LMWM (Yang et al., 2020) reallocates `inodes` which are frequently updated in NVM. Also, both wear reduction and leveling can be combined for further prolonging the lifetime of NVM (Gogte et al., 2019; Qureshi et al., 2009b).

Compared to existing NVM wear leveling techniques, WDBT is able to relocate target memory regions in a fine-grained granularity that is designed specifically for DBT systems. The relocation frequency is also tunable to minimize the performance overhead. This is benefited from the behavior knowledge of DBT systems.

8. Conclusion

In this paper, we identify the critical wear problem of NVM when running applications with DBT. This is caused by the significant amount of additional memory writes introduced by DBT. To understand the reason for the introduced writes, we conduct a comprehensive characterization study of writes in DBT systems. The study finds that most of the writes are caused by emulating guest registers. This is because of the architectural differences between guest and host architectures. Based on this finding, we design WDBT, which creates effective application-level wear reduction and leveling techniques for DBT systems. WDBT dynamically reallocates the host memory locations that are used for emulating guest CPU states. This can distribute writes evenly to different host memory regions. In addition, WDBT utilizes hardware resources available on the host architecture, such as general-purpose registers, to emulate guest registers. This allows WDBT to reduce the absolute number of writes caused by guest register emulation. We also implement a prototype of WDBT based on a real-world cross-ISA DBT system QEMU. Experimental results on SPEC CPU 2017 benchmarks show that WDBT can effectively reduce writes by 52.09% and 34.48% for x86-64 and RISC-V guests, respectively. We believe WDBT provides some insight for our community on application-specific wear reduction and leveling techniques for NVM.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We are very grateful to the anonymous reviewers for their valuable feedback and comments. This work is supported in part by the M. G. Michael Award of the Franklin College of Arts and Sciences at the University of Georgia and a faculty startup funding of the University of Georgia.

References

Akram, S., Sartor, J.B., McKinley, K.S., Eeckhout, L., 2018. Write-rationing garbage collection for hybrid memories. In: Proceedings Of The 39th ACM SIGPLAN Conference On Programming Language Design And Implementation. In: PLDI 2018, Association for Computing Machinery, New York, NY, USA, pp. 62–77. http://dx.doi.org/10.1145/3192366.3192392.

Apple, 2021. About the rosetta translation environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment.

Barbalace, A., Lyerly, R., Jelesnianski, C., Carno, A., Chuang, H.-R., Legout, V., Ravindran, B., 2017. Breaking the boundaries in heterogeneous-ISA datacenters. In: Proceedings Of The Twenty-Second International Conference On Architectural Support For Programming Languages And Operating Systems. In: ASPLOS '17, Association for Computing Machinery, New York, NY, USA, pp. 645–659. http://dx.doi.org/10.1145/3037697.3037738.

Bellard, F., 2005. QEMU, A fast and portable dynamic translator. In: Proceedings Of The Annual Conference On USENIX Annual Technical Conference. In: USENIX ATC '05, USENIX Association, USA, p. 41.

Bruening, D.L., Amarasinghe, S., 2004. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. (Ph.D. thesis). Massachusetts Institute of Technology, USA, AAI0807735.

Bruening, D., Garnett, T., Amarasinghe, S., 2003. An infrastructure for adaptive dynamic optimization. In: Proceedings Of The International Symposium On Code Generation And Optimization: Feedback-Directed And Runtime Optimization. In: CGO '03, IEEE Computer Society, USA, pp. 265–275.

Chen, C.-H., Hsiu, P.-C., Kuo, T.-W., Yang, C.-L., Wang, C.-Y.M., 2012. Age-based PCM wear leveling with nearly zero search cost. In: Proceedings Of The 49th Annual Design Automation Conference. In: DAC '12, Association for Computing Machinery, New York, NY, USA, pp. 453–458. http://dx.doi.org/10.1145/2228360.2228439.

Chen, X., Qingfeng, Z., Sun, Q., Sha, E.H.-M., Gu, S., Yang, C., Xue, C.J., 2019. A wear-leveling-aware fine-grained allocator for non-volatile memory. In: Proceedings Of The 56th Annual Design Automation Conference 2019. In: DAC '19, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3316781.3317752.

Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J., 2003. The transmeta code morphing™ software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings Of The International Symposium On Code Generation And Optimization: Feedback-Directed And Runtime Optimization. In: CGO '03, IEEE Computer Society, USA, pp. 15–24.

DeVuyst, M., Venkat, A., Tullsen, D.M., 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In: Proceedings Of The Seventeenth International Conference On Architectural Support For Programming Languages And Operating Systems. In: ASPLOS XVII, Association for Computing Machinery, New York, NY, USA, pp. 261–272. http://dx.doi.org/10.1145/2150976.2151004.

Dhiman, G., Ayoub, R., Rosing, T., 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In: 2009 46th ACM/IEEE Design Automation Conference. IEEE, pp. 664–669.

Feiner, P., Brown, A.D., Goel, A., 2012. Comprehensive kernel instrumentation via dynamic binary translation. In: Proceedings Of The Seventeenth International Conference On Architectural Support For Programming Languages And Operating Systems. In: ASPLOS XVII, Association for Computing Machinery, New York, NY, USA, pp. 135–146. http://dx.doi.org/10.1145/2150976.2150992.

Ferreira, A.P., Zhou, M., Bock, S., Childers, B., Melhem, R., Mossé, D., 2010. Increasing PCM main memory lifetime. In: 2010 Design, Automation & Test In Europe Conference & Exhibition (DATE 2010). IEEE, pp. 914–919.

Friedman, M., Ben-David, N., Wei, Y., Blelloch, G.E., Petrank, E., 2020. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In: Proceedings Of The 41st ACM SIGPLAN Conference On Programming Language Design And Implementation. In: PLDI 2020, Association for Computing Machinery, New York, NY, USA, pp. 377–392. http://dx.doi.org/10.1145/3385412.3386031.

Gao, T., Strauss, K., Blackburn, S.M., McKinley, K.S., Burger, D., Larus, J., 2013. Using managed runtime systems to tolerate holes in wearable memories. In: Proceedings Of The 34th ACM SIGPLAN Conference On Programming Language Design And Implementation. In: PLDI '13, Association for Computing Machinery, New York, NY, USA, pp. 297–308. http://dx.doi.org/10.1145/2491956.2462171.

Gogte, V., Wang, W., Diestelhorst, S., Kolli, A., Chen, P.M., Narayanasamy, S., Wenisch, T.F., 2019. Software wear management for persistent memories. In: Proceedings Of The 17th USENIX Conference On File And Storage Technologies. In: FAST'19, USENIX Association, USA, pp. 45–63.

Hu, D., Chen, Z., Wu, J., Sun, J., Chen, H., 2021. Persistent memory hash indexes: An experimental evaluation. Proc. VLDB Endow. 14, 785–798. http://dx.doi.org/10.14778/3446095.3446101.

Hu, S., Smith, J.E., 2004. Using dynamic binary translation to fuse dependent instructions. In: Proceedings Of The International Symposium On Code Generation And Optimization: Feedback-Directed And Runtime Optimization. In: CGO '04, IEEE Computer Society, USA, p. 213.

Jiang, J., Dong, R., Zhou, Z., Song, C., Wang, W., Yew, P.-C., Zhang, W., 2020. More with less – deriving more translation rules with less training data for DBTs using parameterization. In: 2020 53rd Annual IEEE/ACM International Symposium On Microarchitecture (MICRO), pp. 415–426, http://dx.doi.org/10.1109/MICRO50266.2020.00043.

Li, D., Reidys, B., Sun, J., Shull, T., Torrellas, J., Huang, J., 2021. UniHeap: MAnaging persistent objects across managed runtimes for non-volatile memory. In: Proceedings Of The 14th ACM International Conference On Systems And Storage. In: SYSTOR '21, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3456727.3463775.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K., 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings Of The 2005 ACM SIGPLAN Conference On Programming Language Design And Implementation. In: PLDI '05, Association for Computing Machinery, New York, NY, USA, pp. 190–200. http://dx.doi.org/10.1145/1065010.1065034.

Microsoft, 2021. How x86 emulation works on ARM. https://docs.microsoft.com/en-us/windows/uwp/porting/apps-on-arm-x86-emulation.

Ou, J., Shu, J., Lu, Y., 2016. A high performance file system for non-volatile main memory. In: Proceedings Of The Eleventh European Conference On Computer Systems. In: EuroSys '16, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/2901318.2901324.

Qureshi, M.K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L., Abali, B., 2009a. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In: Proceedings Of The 42nd Annual IEEE/ACM International Symposium On Microarchitecture. In: MICRO 42, Association for Computing Machinery, New York, NY, USA, pp. 14–23. http://dx.doi.org/10.1145/1669112.1669117.

Qureshi, M.K., Srinivasan, V., Rivers, J.A., 2009b. Scalable high performance main memory system using phase-change memory technology. In: Proceedings Of The 36th Annual International Symposium On Computer Architecture. In: ISCA '09, Association for Computing Machinery, New York, NY, USA, pp. 24–33. http://dx.doi.org/10.1145/1555754.1555760.

Ramos, L.E., Gorbatov, E., Bianchini, R., 2011. Page placement in hybrid memory systems. In: Proceedings Of The International Conference On Supercomputing. In: ICS '11, Association for Computing Machinery, New York, NY, USA, pp. 85–95. http://dx.doi.org/10.1145/1995896.1995911.

Seong, N.H., Woo, D.H., Lee, H.-H.S., 2010. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In: Proceedings Of The 37th Annual International Symposium On Computer Architecture. In: ISCA '10, Association for Computing Machinery, New York, NY, USA, pp. 383–394. http://dx.doi.org/10.1145/1815961.1816014.

Song, C., Wang, W., Yew, P.-C., Zhai, A., Zhang, W., 2019. Unleashing the power of learning: An enhanced learning-based approach for dynamic binary translation. In: Proceedings Of The 2019 USENIX Conference On Usenix Annual Technical Conference. In: USENIX ATC '19, USENIX Association, USA, pp. 77–89.

Wang, W., 2021. Helper function inlining in dynamic binary translation. In: Proceedings Of The 30th ACM SIGPLAN International Conference On Compiler Construction. In: CC 2021, Association for Computing Machinery, New York, NY, USA, pp. 107–118. http://dx.doi.org/10.1145/3446804.3446851.

Wang, C., Cao, T., Zigman, J., Lv, F., Zhang, Y., Feng, X., 2016a. Efficient management for hybrid memory in managed language runtime. In: Gao, G.R., Qian, D., Gao, X., Chapman, B., Chen, W. (Eds.), Network And Parallel Computing. Springer International Publishing, Cham, pp. 29–42.

Wang, R., Chen, G., Liang, N., Huang, Z., 2021. Preventive maintenance optimization regarding large-scale systems based on the life-cycle cost. Int. J. Performabil. Eng. 17, 766. http://dx.doi.org/10.23940/ijpe.21.09.p3.766778, URL http://www.ijpe-online.com/EN/abstract/article_4619.shtml.

Wang, T., Johnson, R., 2014. Scalable logging through emerging non-volatile memory. Proc. VLDB Endow. 7, 865–876. http://dx.doi.org/10.14778/2732951.2732960.

Wang, W., McCamant, S., Zhai, A., Yew, P.-C., 2018a. Enhancing cross-ISA dbt through automatically learned translation rules. In: Proceedings Of The Twenty-Third International Conference On Architectural Support For Programming Languages And Operating Systems. In: ASPLOS '18, Association for Computing Machinery, New York, NY, USA, pp. 84–97. http://dx.doi.org/10.1145/3173162.3177160.

Wang, W., Wu, C., Bai, T., Wang, Z., Yuan, X., Cui, H., 2014. A pattern translation method for flags in binary translation. J. Comput. Res. Dev. 51, 2336–2347, URL http://crad.ict.ac.cn/EN/10.7544/issn1000-1239.2014.20130018.

Wang, W., Wu, J., Gong, X., Li, T., Yew, P.-C., 2018b. Improving dynamically-generated code performance on dynamic binary translators. In: Proceedings Of The 14th ACM SIGPLAN/SIGOPS International Conference On Virtual Execution Environments. In: VEE '18, Association for Computing Machinery, New York, NY, USA, pp. 17–30. http://dx.doi.org/10.1145/3186411.3186413.

Wang, W., Yew, P.-C., Zhai, A., McCamant, S., 2016b. A general persistent code caching framework for dynamic binary translation (DBT). In: Proceedings Of The 2016 USENIX Conference On Usenix Annual Technical Conference. In: USENIX ATC '16, USENIX Association, USA, pp. 591–603.

Wang, W., Yew, P.-C., Zhai, A., McCamant, S., 2020. Efficient and scalable cross-ISA virtualization of hardware transactional memory. In: Proceedings Of The 18th ACM/IEEE International Symposium On Code Generation And Optimization. In: CGO 2020, Association for Computing Machinery, New York, NY, USA, pp. 107–120. http://dx.doi.org/10.1145/3368826.3377919.

Wang, W., Yew, P.-C., Zhai, A., McCamant, S., Wu, Y., Bobba, J., 2017. Enabling cross-ISA offloading for COTS binaries. In: Proceedings Of The 15th Annual International Conference On Mobile Systems, Applications, And Services. In: MobiSys '17, Association for Computing Machinery, New York, NY, USA, pp. 319–331. http://dx.doi.org/10.1145/3081333.3081337.

Wen, W., Zhang, Y., Yang, J., 2018. Wear leveling for crossbar resistive memory. In: Proceedings Of The 55th Annual Design Automation Conference. In: DAC '18, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3195970.3196138.

Wu, J., Dong, J., Fang, R., Wang, W., Zuo, D., 2020. PerfDBT: Efficient performance regression testing of dynamic binary translation. In: 2020 IEEE 38th International Conference On Computer Design (ICCD), pp. 389–392, http://dx.doi.org/10.1109/ICCD50377.2020.00071.

Wu, J., Dong, J., Fang, R., Zhao, Z., Gong, X., Wang, W., Zuo, D., 2021. Effective exploitation of SIMD resources in cross-ISA virtualization. In: Proceedings Of The 17th ACM SIGPLAN/SIGOPS International Conference On Virtual Execution Environments. In: VEE 2021, Association for Computing Machinery, New York, NY, USA, pp. 84–97. http://dx.doi.org/10.1145/3453933.3454016.

Xu, C., Niu, D., Muralimanohar, N., Balasubramonian, R., Zhang, T., Yu, S., Xie, Y., 2015. Overcoming the challenges of crossbar resistive memory architectures. In: 2015 IEEE 21st International Symposium On High Performance Computer Architecture (HPCA), pp. 476–488, http://dx.doi.org/10.1109/HPCA.2015.7056056.

Xu, J., Swanson, S., 2016. NOVA: A Log-structured file system for hybrid volatile/non-volatile main memories. In: Proceedings Of The 14th Usenix Conference On File And Storage Technologies. In: FAST'16, USENIX Association, USA, pp. 323–338.

Yang, Q., Li, Z., Liu, Y., Long, H., Huang, Y., He, J., Xu, T., Zhai, E., 2019. Mobile gaming on personal computers with direct android emulation. In: The 25th Annual International Conference On Mobile Computing And Networking. In: MobiCom '19, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3300061.3300122.

Yang, C., Liu, D., Zhang, R., Chen, X., Nie, S., Wang, F., Zhuge, Q., Sha, E.H.-M., 2020. Efficient multi-grained wear leveling for inodes of persistent memory file systems. In: Proceedings Of The 57th ACM/EDAC/IEEE Design Automation Conference. In: DAC '20, IEEE Press.

Yu, H.-C., Lin, K.-C., Lin, K.-F., Huang, C.-Y., Chih, Y.-D., Ong, T.-C., Chang, J., Natarajan, S., Tran, L.C., 2013. Cycling endurance optimization scheme for 1Mb STT-MRAM in 40nm technology. In: 2013 IEEE International Solid-State Circuits Conference Digest Of Technical Papers, pp. 224–225, http://dx.doi.org/10.1109/ISSCC.2013.6487710.

Zhang, W., Wang, X., Cabrera, D., Bai, Y., 2020. Product quality reliability analysis based on rough Bayesian network. Int. J. Performabil. Eng. 16, 37. http://dx.doi.org/10.23940/ijpe.20.01.p5.3747, URL http://www.ijpe-online.com/EN/abstract/article_4340.shtml.

Zhao, Z., Jiang, Z., Chen, Y., Gong, X., Wang, W., Yew, P.-C., 2021. Enhancing atomic instruction emulation for cross-ISA dynamic binary translation. In: 2021 IEEE/ACM International Symposium On Code Generation And Optimization (CGO), pp. 351–362, http://dx.doi.org/10.1109/CGO51591.2021.9370312.

Zhao, Z., Jiang, Z., Liu, X., Gong, X., Wang, W., Yew, P.-C., 2020. DQEMU: A scalable emulator with retargetable DBT on distributed platforms. In: 49th International Conference On Parallel Processing - ICPP. In: ICPP '20, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3404397.3404403.