WDBT: Wear Characterization, Reduction, and Leveling of DBT Systems for Non-Volatile Memory

Jin Wu
Harbin Institute of Technology

Jian Dong
Harbin Institute of Technology

Ruili Fang
University of Georgia

Wen Zhang
University of Georgia

Wenwen Wang
University of Georgia

Decheng Zuo
Harbin Institute of Technology

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. Therefore, many wear reduction and leveling mechanisms have been proposed for NVM. 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. To address this problem, 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 benchmarks show that WDBT can effectively reduce writes by 52.09% and 34.48% when emulating x86-64 and RISC-V, respectively. Moreover, the performance overhead of WDBT is negligible.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; **Memory management**; **Dynamic compilers**.

KEYWORDS

Non-volatile memory, Cross-ISA virtualization, Dynamic binary translation, QEMU

ACM Reference Format:
Jin Wu, Jian Dong, Ruili Fang, Wen Zhang, Wenwen Wang, and Decheng Zuo. 2021. WDBT: Wear Characterization, Reduction, and Leveling of DBT Systems for Non-Volatile Memory. In *The International Symposium on Memory Systems (MEMSYS 2021), September 27–30, 2021, Washington DC, DC, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3488423.3488457

MEMSYS 2021, September 27–30, 2021, Washington DC, DC, USA

ACM ISBN 978-1-4503-8570-1/21/09…$15.00
https://doi.org/10.1145/3488423.3488457

1 INTRODUCTION

Non-volatile memory (NVM) technologies, such as phase-change memory [26], spin-transfer torque magnetic random-access memory [46], and Memristor [42], 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 [24, 43], NVM-based databases [17, 31], language runtimes [20, 30], and persistent data structures [14].

Unfortunately, compared to DRAM, NVM has much more limited write endurance, which may hinder the adoption of NVM to construct 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 is expected to endure $10^7 - 10^9$ writes [25] and resistive memories may sustain over 10^{10} writes [42]. Without any wear management, the lifetime of an NVM device can be as short as several months [16]. In contrast, traditional DRAM can usually be used for many years. Therefore, it is of paramount importance to develop effective wear management mechanisms to prolong the life time of NVM.

To this end, plenty of wear reduction and leveling mechanisms have been proposed, ranging from hardware-based [7, 25, 39] to pure software systems [1, 16, 44]. However, these techniques usually suffer from one or more limitations. For example, hardware-based wear leveling techniques often stop at the research prototype stage and it is not clear whether they will be integrated 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. To give an example, many software-based wear leveling techniques are implemented at the operating system level. They use a memory page (typically 4 KB on x86-64 platforms) as the granularity of wear leveling, and therefore, it is hard for these techniques to address the intra-page wear issue caused by unbalanced writes to different memory locations in the same page without remapping the page.

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

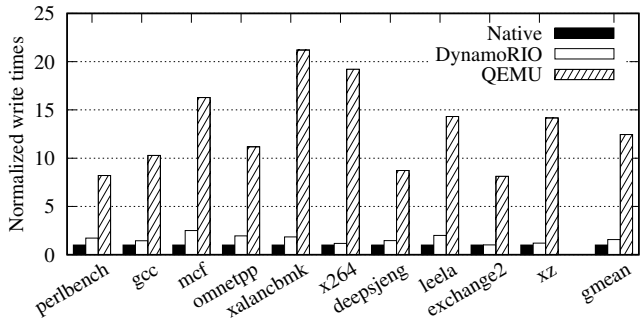

Figure 1: Applications running with DBT (i.e., QEMU and DynamoRIO) have more memory writes than native execution without DBT. For DynamoRIO, no instrumentation is applied. For QEMU, an x86-64 instruction set is emulated.

scheme to exploit application-specific potentials. More specifically, we focus on applications running with dynamic binary translation (DBT), which include a wide range of important applications, such as runtime code optimizations and analyses [5, 9, 12, 18], workload migration [3, 10], cross-architecture system emulation [2, 22, 37, 47, 48], and mobile computation offloading [38, 45]. In essence, 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 be "upgraded" to utilize NVM resources in the foreseeable future. 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. Particularly, applications running with DBT often exhibit significantly more memory writes, which can wear out NVM hardware rapidly if no memory management is applied.

Figure 1 shows the normalized number of writes for the integer benchmarks in SPEC CPU 2017 running with two representative DBT systems, DynamoRIO [6] and QEMU [4], respectively. The baseline is the native execution of the benchmarks without DBT. The results are obtained through dynamic instrumentation of memory write instructions using Intel Pin [21]. More details about the instrumentation tool can be found in Section 3. We also tried to collect results for another popular DBT system Valgrind [23] as well as Pin itself, but failed to conduct the instrumentation for them. From the figure, we can clearly see that more writes are issued when applications are running with DBT systems. In particular, QEMU introduces as high as 22x 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. Although not all of these writes will reach the main memory because of the existence of CPU caches, there is no doubt that these extra writes will put further pressure on the limited write endurance of NVM when running applications with DBT.

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. The techniques are designed based on the common practices of DBT systems. Specifically, we conduct a comprehensive characterization study to understand the semantics and behaviors of 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 multiple mainstream ISAs, i.e., x86-64 and RISC-V as the guest ISAs and AArch64 as the host ISA. QEMU employs the tiny code generator (TCG) to translate guest binary code into host binary code. Our implementation carefully redesigns the translation process of TCG to incorporate the proposed emulation scheme in WDBT. We plan to open source the implementation after paper acceptance to benefit our community. To evaluate the effectiveness of WDBT, we use the SPEC CPU 2017 benchmark suite. Experimental results show that WDBT can reduce writes introduced by DBT for all evaluated benchmarks. On average, the number of writes can be reduced by 52.09% and 34.48% for x86-64 and RISC-V guest ISAs, respectively. Moreover, the performance overhead of WDBT is negligible.

In summary, this paper makes the following contributions:

- 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.

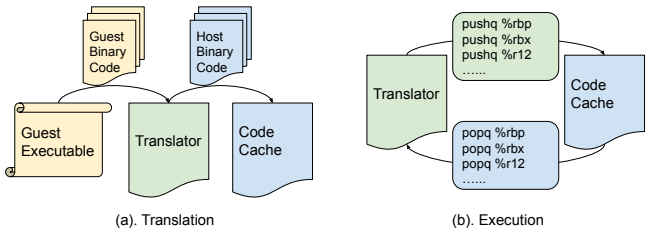

Figure 2: The work flow of a typical DBT system.

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 support from a file system. With such advantages, NVM is becoming the major solution to construct the next-generation memory system. However, NVM suffers from a fundamental limitation, i.e., the limited write endurance. Recent research has shown that a phase-change memory (PCM) cell will permanently wear out after 10^7 to 10^8 writes [28]. In an extreme case, an application can reach such a limitation within several minutes [28]. As a result, it is necessary and urgent to develop effective wear management schemes to prolong the life time of NVM.

2.1.2 Dynamic Binary Translation (DBT). DBT is a key enabling technology for many important applications. 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 [19, 29, 33]. Figure 2 shows the high-level workflow of a DBT system. The translation is conducted at the granularity of *basic blocks* [40]. Each basic block consists of a sequence of guest instructions with at most one branch instruction at the end of the block. That is to say, a basic block has only one entry and one exit, and is executed sequentially from the first instruction to the last instruction. The translated host binary code is saved to a software-managed *code cache* and reused in the following execution to mitigate the translation overhead [35, 36], 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 referred to as 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 achieve this by leveraging host memory resources due to the architectural differences between the guest and host architectures [41]. For example, the guest and host architectures may provide different numbers of registers. Therefore, to hide such differences, each component in the guest CPU that is visible to a user-level application needs to be emulated using a host memory location. Therefore, an update to a guest register or condition code is translated into an update to the corresponding memory location. 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 A WEAR CHARACTERIZATION STUDY OF DBT SYSTEMS

In this section, we conduct a wear characterization study of DBT. The purpose of this study is to understand the memory write behaviors of DBT systems and discover the reason behind the additional writes introduced by DBT. Our study employs QEMU [4] as the example DBT system, because it is one of the most popular and currently available cross-ISA DBT systems. Since the translation/emulation mechanisms in other DBT systems are very similar to QEMU, we believe the findings uncovered in this study can also apply to them.

Different from regular applications, there are two execution environments in a DBT system, i.e., the translator and the code cache. The translator environment is responsible to translate guest instructions into semantically-equivalent host instructions, while the code cache is used to store the translated host instructions for the execution. Since both of these two execution environments may introduce additional writes, our first study is to figure out which environment introduces more writes. To this end, we develop a dynamic binary instrumentation tool based on Intel Pin [21]. The tool collects profiling information about memory writes in DBT systems, e.g., how many and where write instructions are executed during the emulation of a guest application. Figure 3 shows the results for integer benchmarks in the SPEC CPU 2017 benchmark

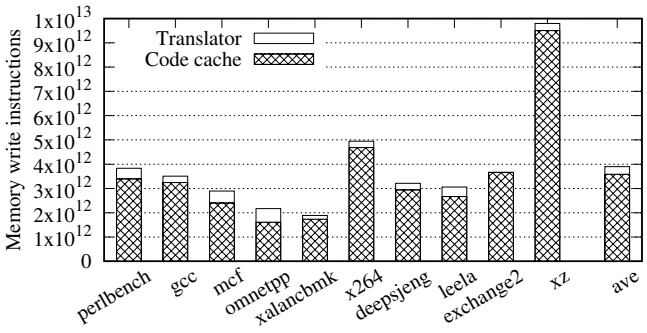

Figure 3: Memory writes issued from code cache account for the majority of the writes caused by DBT.

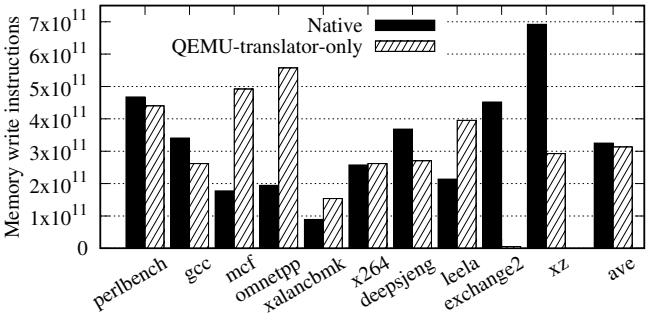

Figure 4: The number of memory writes issued by the translator, compared against the number of writes in the native execution.

suite. From the figure, we can observe that an average of 92% of memory writes are issued by the host code in code cache. This means that wear reduction and leveling techniques for DBT systems need to pay special attention to the translated host binary code, as most of the writes are caused by them.

Next, to further reason about the additional writes in DBT systems, we will present more experimental results about the writes executed in each of the two execution environments, as well as context switches between them.

3.1 Instruction Translation

The translator is the core component of a DBT system. The translation process typically consists of several steps: disassembling guest binary code, translating guest instructions into host instructions, and assembling the translated host instructions into host binary code. In QEMU, the tiny code generator (TCG) is employed to conduct the translation. TCG is designed with a platform-independent intermediate representation (IR) to achieve *retargetability*. That is, guest instructions are first translated into TCG IRs and then translated to host instructions. This IR-based translation process mitigates the engineering efforts required to support multiple guest and host ISAs in QEMU. Besides, complex guest instructions, such as system calls, and floating-point instructions, are translated to helper function calls to simplify the translation process.

Though most of the memory writes are not contributed by the translator, we still collect the number of memory writes issued by the translator and compare it against the native execution. This will help us to understand the writes in the translator. Figure 4 shows the results for the SPEC CPU 2017 benchmarks. As shown in the figure, the translator introduces an average of 0.96x memory writes, compared to the native execution. Note that all writes introduced by the translator happen during the translation process and thus will disappear after all basic blocks have been translated. Next, we will study the writes caused by the emulation of guest CPU states, which lasts until the end of the execution.

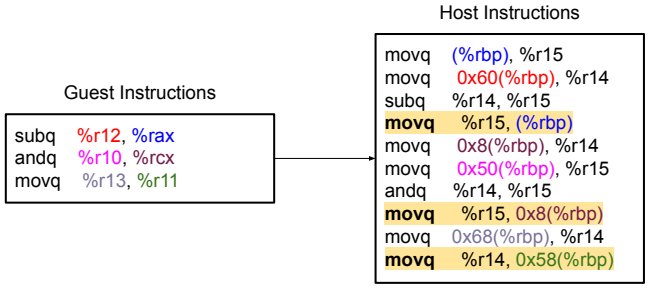

Figure 5: An example of the translation in QEMU. The guest and host ISAs are both x86-64. Each guest register is emulated by a host memory location in the same color. Three host write instructions (highlighted) are introduced during the translation process.

3.2 Guest CPU State Emulation

Typically, during the execution of an application, one of the frequently updated components in the CPU is the set of general-purpose registers. Hence, the host memory location used for emulating guest registers are also accessed intensively. Figure 5 shows a code translation example in QEMU. 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 `0x8(%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.

In addition to writes caused by emulating guest CPU states, there are also other writes, e.g., used to emulate guest memory access instructions. Figure 6 shows the distribution of memory writes in the code cache. As shown in the figure, on average, 79% memory

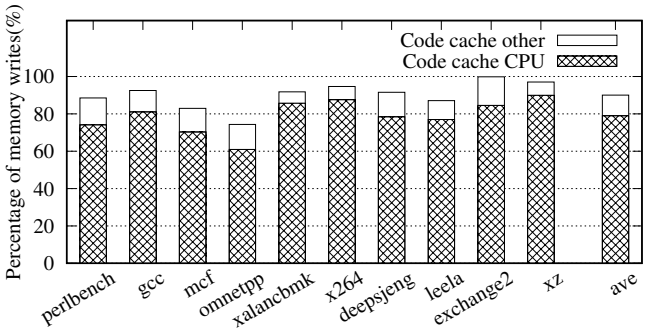

Figure 6: The distribution of memory writes in the code cache.

Table 1: 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

writes in code cache are introduced because of the emulation of guest CPU states.

3.3 Context Switch

As mentioned before, there are two execution environments in a DBT system, i.e., the translator and the code cache. The two environments use the same host machine resources in a time-sharing manner. Thus, when the execution flow is transferred from one environment to another, a context switch is required to save and restore the host machine states, such as host general-purpose registers. Since this process also includes memory writes, it is necessary to conduct a study to understand the quantity of memory writes in this process. Table 1 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. This is mainly because the execution of a DBT system is rarely transferred to the translator after all guest basic blocks are translated.

3.4 Summary

To summarize, Figure 7 shows the distribution of memory writes in the evaluated DBT system, QEMU. We can see from the figure

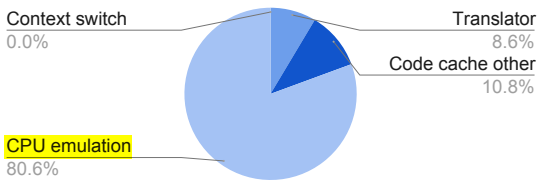

Figure 7: The distribution of memory writes in a real DBT system, QEMU.

that the emulation of the guest CPU states introduces the largest portion of writes in the DBT system, which accounts for more than 80% of total writes. The writes caused by other instructions in the code cache and the translator are 10.8% and 8.6%, respectively. In addition, the writes introduced by context switches are close to 0%. According to this observation, we design WDBT to conduct effective wear reduction and leveling for DBT systems on NVM.

4 SYSTEM DESIGN OF WDBT

In this section, we describe the design of WDBT, which aims to prolong the lifetime of NVM for applications running with DBT. WDBT 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 tied to any specific DBT system.

As concluded by the above study, most of the additional writes in a DBT system are introduced by the emulation of the guest CPU states, as the updates to guest CPU state are translated into accesses to host memory locations. To reduce the intensive writes to a specific host memory region, WDBT incorporates two novel and effective techniques. First, a wear leveling technique to periodically move the emulated guest CPU state from one memory region to another. This allows WDBT to distribute the intensive writes to different host memory regions. Second, a wear reduction technique to utilize available host CPU resources, e.g., registers, to emulate guest CPU states. This way, WDBT can reduce the number of host memory writes by replacing them with write operations to host registers. Next, we describe these two techniques in detail.

4.1 Dynamic Reallocation of Host Memory for Wear Leveling

In general, general-purpose registers in a CPU are one of the most frequently accessed components. Typically, an instruction takes one or more registers as operands. Depending on the semantic of the instruction, the value in one of the registers, which is usually considered as a *destination* register, may be modified based on the computation result of the instruction. In a DBT system, with guest registers emulated using host memory locations, the update to a destination register of a guest instruction is emulated by writing to the corresponding host memory location after the emulation of the instruction operation. This leads to a large volume of writes to the host memory region that is used to emulate guest CPU states. 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.

4.1.1 Dynamically allocating host memory locations for guest CPU emulation. When emulating a guest CPU, a DBT system often creates the emulated processor, including the general-purpose registers and other user-visible registers, at the initialization stage when the system is launched. That is, the host memory locations allocated for emulating guest registers are *static* and fixed. These memory locations will not change during the entire execution process. But, when porting such a DBT system to NVM, the memory region on NVM for register emulation will be accessed intensively and these accesses 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. Such a mechanism allows the memory locations for register emulation to be allocated at runtime. 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 [8, 15]. To this end, both the translator and the generated host binary code need to support such dynamically allocated host memory locations for guest register emulation.

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. At the same time, to access the dynamically allocated host memory locations in code cache, WDBT passes the value of the pointer variable to the generated host binary code.

4.1.2 Reallocation of host memory regions. To further mitigate the wear pressure on a specific host memory region, WDBT also periodically reallocates the host memory locations for guest registers to different host memory regions. The reallocation is triggered 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. Note that if there is no available general-purpose register, WDBT can alternatively reserve a vector register or floating-point register, e.g., `%xmm` on x86-64 platforms and `v` on AArch64 platforms, to store the counter. These registers are typically available and abundant on modern architectures.

Algorithm 1 shows the reallocation of the memory space for guest register emulation. It firstly sets up the initial location for the emulated guest CPU state and places the threshold of the basic block counter into the reserved host register. Each time after a basic block is executed, the value in the register is decreased by one. Once the value reaches 0, a reallocation will be executed. The reallocation process consists of 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 new 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. Note that the threshold value can be tuned individually for different guest applications in practice to balance the performance efficiency and NVM wear pressure, given that different guest applications may execute completely different numbers of basic blocks.

Algorithm 1: Dynamic Reallocation of Guest CPU States

```
1  BbDec ← THRESHOLD;
2  CPUState ← malloc(sizeof(CPUStateEmu) * CPUStateLen);
3  CurIndex ← 0;
4  CurState ← CPUState[CurIndex];
5  foreach Basic block Bb do
6      Excute(Bb);
7      BbDec ← BbDec − 1;
8      if BbDec = 0 then
9          NextIndex ← (CurIndex + 1)%CPUStateLen;
10         CPUState[NextIndex] ← CPUState[CurIndex];
11         CurState ← CPUState[CurIndex];
12     else
13         continue;
14     end
15 end
```

4.2 Leveraging Host Registers for Wear Reduction

For long-running and computation-intensive guest applications, the memory-based emulation scheme for guest registers can result in a large amount of writes to NVM. Even with the aforementioned write leveling technique, it still puts significant wear pressure on the memory space allocated for each time interval. 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 guest registers, which means the accesses to guest registers are translated into accesses to corresponding host registers, rather than host memory locations. This way, the number of writes, as well as the pressure on the limited write endurance of NVM, can be reduced.

Figure 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 two 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.

4.2.1 Collaborating with memory-based emulation. There are at least two reasons why the register-based emulation scheme needs to collaborate with existing memory-based emulation. First, 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

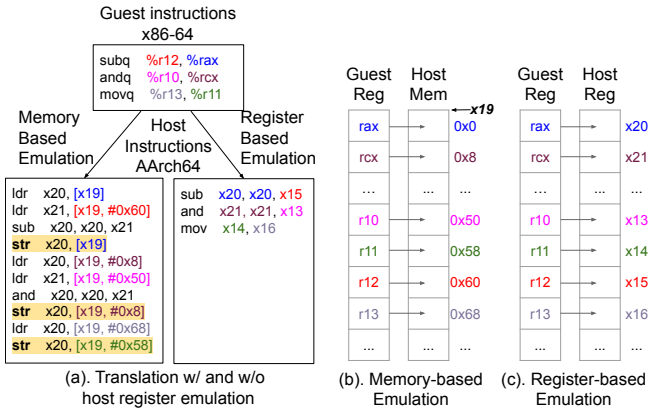

Figure 8: The register-based emulation scheme of guest registers have less writes in the translated host binary code than the memory-based emulation scheme.

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. Hence, it is not easy to utilize it for guest register emulation.

To facilitate the collaboration between the two different emulation schemes, WDBT introduces a *flag* to indicate where the latest value of an emulated guest register is placed. If the flag is zero, it means the value is in host memory, otherwise, the value is in a host register. 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.

4.2.2 Resolving host register conflicts. Different from memory locations, host registers are valuable resources shared among all modules of a DBT system, including the translator and the code cache. Therefore, WDBT needs to resolve the potential conflicts between different usage scenarios of host registers. This is realized by reserving host registers that are used for guest register emulation. This way, the host registers will not be used by the generated host instructions for other purposes. However, this is not sufficient, as the translator may also use the same host registers. That is, when the execution is transferred to the translator, the values in the reserved host registers may be corrupted. Since the translator is compiled from the source code, it is also hard to avoid the usage of specific host registers 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. This way, WDBT can resolve the conflicts and guarantee the correctness of the emulation.

4.2.3 Addressing limited host registers. As discussed before, general-purpose registers are usually limited resources of a processor. Although host memory locations can be used to emulate guest registers when there are no sufficient host registers, this may not be helpful to reducing the pressure on the limited write endurance of NVM. Therefore, 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,

Algorithm 2: 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;
```

WDBT leverages host vector registers or floating-point registers to hold updated values of emulated guest registers, instead of writing them back to memory. For example, an x86-64 machine typically provides up to 32 vector registers and an AArch64 processor offers both floating-point registers and vector registers. By utilizing such register storage resources, WDBT can further reduce the number of write instructions in the translated host binary code.

4.2.4 Adaptive code translation. We now describe the translation process in WDBT. Since there are two emulation mechanisms, WDBT needs to translate guest binary code adaptively based on the emulation schemes of the guest registers involved in each guest instruction. Algorithm 2 shows the details of the adaptive translation process in WDBT. For guest registers emulated by host memory locations, floating-point registers, or vector registers, the original translation mechanism can still work, as 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.

5 IMPLEMENTATION

We have implemented a prototype of WDBT based on QEMU [4] (version 6.0.0), which is a widely-used cross-ISA DBT system. Our implementation currently supports RISC-V and x86-64 as the guest ISAs and AArch64 as the host ISA. These ISAs are among the most popular ISAs in the market. 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.

QEMU employs the tiny code generator (TCG) to perform the code translation. To support multiple guest and host architectures, QEMU first translates guest binary code into TCG IR, which is a platform-independent intermediate representation, and then generates host binary code from TCG IR. This approach effectively mitigates the engineering efforts required to support multiple guest and host ISAs. As a cross-ISA DBT system, QEMU is able to emulate various guest processor states. This is realized by using host memory locations to maintain the states of possible guest processor states. Thus, QEMU may bring significant pressure on the limited write endurance of NVM when ported to 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.

A straightforward approach is to update the host memory locations of the emulated guest registers in the translator. This approach simplifies the update process as the translators in existing DBT systems are typically implemented in high-level programming languages, e.g., C/C++. Another advantage of this approach is that it can pass the address of the new memory space to the code cache so that the translated host code can access the emulated registers via the address. However, this approach may force the DBT system to exit from code cache periodically, and thus unavoidably introduces extra context switches between the translator and code cache. Since the emulated guest registers need to be synchronized with the corresponding host memory locations during context switches, this will lead to memory writes. Similarly, implementing the reallocation as a function call invoked from the code cache can also result in unexpected memory writes.

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. To further reduce the performance overhead of memory allocation, WDBT reserves a large memory space for reallocation at the initialization stage when the DBT system is launched. In this way, WDBT can use the available memory region when reallocation is triggered, instead of allocating the space every time.

Once the host memory space for guest register emulation is reallocated in the code cache, the corresponding pointers in the translator still point to the old location. Hence, 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. Note that the function is expected to be called immediately after the execution exits from the code cache. 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. We will describe more details in the next section.

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 the 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 bound with a guest register and 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. For example, if a guest register is emulated by a host memory location, the translator needs to generate host code to load the value of the guest register from the memory location, conduct the computation, and then store the updated value to the memory location. Conversely, if the guest register is emulated by a host register, the translator can generate host code to conduct the computation directly with the host register without the necessity to access the memory. 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 code cache and an epilogue before exiting from 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. A potential concern about the enhanced context switch is that more memory writes will be introduced and this may put extra pressure on the limited write endurance of NVM. But this is not true, as compared to the memory writes reduced by the register-based emulation, the memory writes added in context switches are practically negligible. This is because the total number of context switches is very small during the emulation of a guest application, as shown in Table 1. We will present more results in Section 6.

5.2.3 Helper function. To mitigate the engineering effort, QEMU employs helper functions to emulate complex guest instructions [32] They are developed using the same high-level language as QEMU, i.e., C, and compiled to host binary code by a native host compiler. Since the compiled host code of helper functions can also use host registers, it is necessary to save the values in the host registers used for guest register emulation before invoking helper functions from code cache and restore the values after returning.

A straightforward solution is to save and restore *all* related host registers. But, this will introduce a large amount of memory writes, especially for guest applications with frequent helper function calls. Our further study finds that the compiled host code of helper functions also obeys to the application binary interface (ABI) on the host platform, which specifies which registers are caller-saved and which registers are callee-saved. The value in a caller-saved register is not preserved across function calls, while a callee-saved register will keep the same value after a function call as it has before the call. That means, our implementation only needs to take care of caller-saved registers, as callee-saved registers will be saved and restored automatically by the compiled code of helper functions according to the ABI. Through this way, we can reduce many unnecessary writes caused by invoking helper functions while maintaining the correctness of the emulation.

6 EVALUATION

In this section, we evaluate the effectiveness of WDBT on wear reduction and leveling for NVM. We first describe the details of our experimental setup and then present the evaluation results.

6.1 Experimental Setup

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.

Our evaluation platform is equipped with an AArch64 RK3399 big.LITTLE processor, which includes dual-core Cortex-A72 at 2.0GHz and quad-core Cortex-A53 at 1.5GHz. The main memory is 4GB LPDDR4. All the tests are performed on big cores, which have 32KB private L1d cache for each core and 1MB unified L2 cache. We use the main memory to simulate NVM writes. Though the actual access latency of NVM is longer, the total number of writes should be the same. The operating system is the Manjaro AArch64 version with Linux kernel 5.7.19. The evaluation platform is exclusively occupied during our experiments to reduce the potential influence of random factors. In addition, each benchmark is evaluated 5 times and their average is used as the final result.

We first evaluate the benefits of wear reduction and leveling techniques in WDBT individually and then present the combined results of reduced writes for NVM.

6.2 Wear Reduction

On modern architectures, CPU caches can absorb some writes, as a write is usually cached in CPU caches until it needs to be written back to main memory. Therefore, our evaluation covers both *cacheless* architectures and architectures with caches. Specifically, on a cacheless platform, the total number of writes to the main memory is the same as the number of write instructions executed. But, on a platform with caches, a write reaches to the main memory only when there is a cache miss or a dirty cache line needs to be written back. Hence, the number of actual writes to the main memory is typically less than the number of write instructions on a platform with caches. To measure the actual number of writes to main memory, we use *hardware performance counters* available on our evaluation platform. Specifically, we use the cache write back event to count the number of writes to main memory. This allows us to eliminate the influence of CPU caches on memory writes.

Table 2 shows the results of 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 Figure 6, because besides general-purpose registers, updates to other emulated guest CPU components, such as SIMD registers and condition codes [34], 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.

Table 2: 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	1.0e10	12.76	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	8.4e9	11.90	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	5.0e10	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	93.24	5.6e12	2.5e12	54.67	9.4e9	6.0e9	35.97	2.5e12	1.5e12	38.65
exchange2	9.0e9	2.0e9	77.40	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			**16.09**			**52.09**			**5.01**			**34.48**

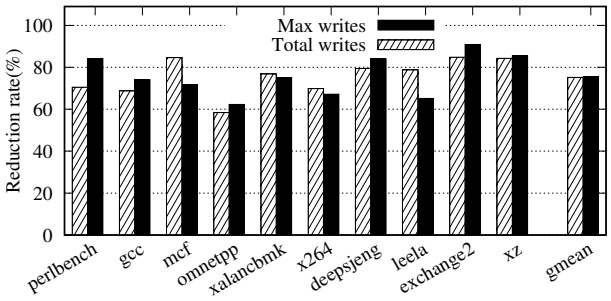

Figure 9: WDBT reduces the maximum number of writes and the total number of writes for prolonging the lifetime of NVM. Here, RISC-V is the guest ISA.

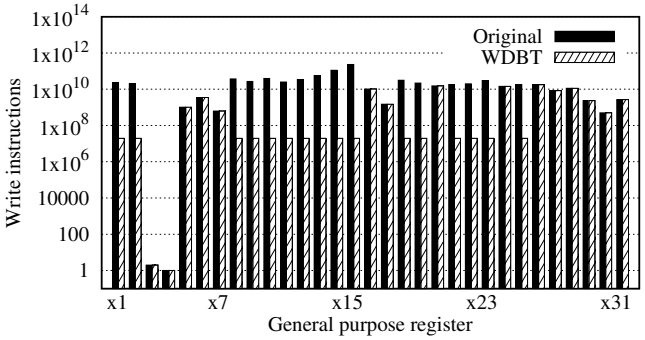

Figure 10: The number of writes caused by emulating each individual guest register. The benchmark is *perlbench* and the guest ISA is RISC-V.

The cache write back data in Table 2 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 32KB L1d cache and 1MB unified L2 cache, the memory write reduction rate is 16.09% 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.

To further understand how WDBT reduces writes through the register-based emulation scheme, we conduct more experiments. In particular, we measure the number of memory writes to each individual host memory location that is used to emulate a guest register. 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. 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. Figure 9 shows the results for 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, 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 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.

Figure 10 shows the number of writes caused by emulating each individual guest register in WDBT and original QEMU. Here, the benchmark is *perlbench* and the guest ISA is RISC-V. Note that 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 they are rarely accessed in guest applications. As shown in the figure, WDBT successfully reduces writes for guest register emulation by mapping guest registers host registers. The number of reduced writes can be as high as 10000x for a single guest register, i.e., `x15`. Though some guest registers are still emulated

Table 3: Memory write reduction with guest CPU state reallocation. The guest ISA is RISC-V.

	256B		512KB		512MB	
	total	max	total	max	total	max
perlbench	1.9e12	5.1e11	9.4e8	2.5e8	9.2e5	2.4e5
gcc	7.4e11	1.5e11	3.6e8	7.3e7	3.5e5	7.2e4
mcf	9.2e11	1.4e11	4.5e8	7.0e7	4.4e5	6.8e4
omnetpp	7.5e11	1.4e11	3.6e8	6.6e7	3.6e5	6.5e4
xalancbmk	6.9e11	1.7e11	3.4e8	8.5e7	3.3e5	8.3e4
x264	3.0e12	3.2e11	1.5e9	1.6e8	1.4e6	1.5e5
deepsjeng	1.5e12	3.2e11	7.3e8	1.6e8	7.1e5	1.5e5
leela	1.7e12	2.3e11	8.2e8	1.1e8	8.0e5	1.1e5
exchange2	2.0e12	4.4e11	9.9e8	2.1e8	9.7e5	2.1e5
xz	2.0e12	2.8e11	9.9e8	1.4e8	9.6e5	1.3e5

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.

6.3 Wear Leveling

We next evaluate the wear leveling technique in WDBT. This is achieved by dynamically reallocating the emulated guest CPU state during the emulation process. Table 3 shows the number of writes with different sizes of memory space for guest CPU state reallocation. The guest ISA is RISC-V. We omit the results for x86-64 due to the high similarity. Note that it takes a memory space of 256 bytes to emulate all RISC-V general-purpose registers (64 bits × 32). Therefore, there is no reallocation for the 256B case in the table. As shown in the table, WDBT reduces the number of writes by reallocating the host memory locations used for emulating guest registers. For example, when the memory space is configured as 512MB for reallocation, 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.

6.4 Overall Effectiveness and Performance Analysis

Finally, we conduct experiments to study the overall effectiveness of WDBT. Table 4 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.

In terms of performance, the execution time of WDBT and the original QEMU is very similar. The reason is that most memory writes in the original QEMU don't cause cache misses. Therefore, even if WDBT replaces them with instructions accessing host registers, the performance benefit is moderate. At the same time, WDBT introduces additional memory access instructions during context switches and helper function calls, which can offset the benefit of the register-based emulation scheme. As a consequence, the overall performance overhead of WDBT is negligible.

Table 4: Memory writes reduced by both the wear reduction and leveling techniques in WDBT, Ori represents the original QEMU, Reg represents host register emulated guest registers, Full represents with both host register based emulation and CPU state reallocation.

Guest	**x86-64**			**RISC-V**		
Config	Ori	Reg	Full	Ori	Reg	Full
MemSize	128 (B)	128 (B)	128 (MB)	256 (B)	256 (B)	512 (MB)
perlbench	1.4e12	6.6e8	6.3e2	1.9e12	5.7e11	2.7e5
gcc	6.5e11	3.8e9	3.6e3	7.4e11	2.3e11	1.1e5
mcf	8.9e11	1.3e10	1.2e4	9.2e11	1.4e11	6.8e4
omnetpp	7.1e11	7.5e10	7.1e4	7.5e11	3.1e11	1.5e5
xalancbmk	5.1e11	9.0e7	8.6e1	6.9e11	1.6e11	7.6e4
x264	2.4e12	3.3e9	3.2e3	3.0e12	9.1e11	4.3e5
deepsjeng	1.5e12	2.3e9	2.2e3	1.5e12	3.1e11	1.5e5
leela	1.3e12	1.1e10	1.1e4	1.7e12	3.6e11	1.7e5
exchange2	1.3e12	1.7e8	1.6e2	2.0e12	3.1e11	1.5e5
xz	1.8e12	9.2e4	≤ 1	2.0e12	3.2e11	1.5e5

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 [1, 11, 13, 26, 27]. The basic idea is to place write-intensive data in DRAM while keeping only persistent data in NVM. For example, PDRAM [11] proposes a hybrid DRAM and NVM memory system, which allocates pages according to a hardware-based write frequency strategy. Kingsguard [1] places read-intensive objects in NVM while write-intensive ones in DRAM for garbage collectors. RaPP [27] ranks memory pages by a sophisticated memory controller and the top-ranked pages are migrated to DRAM. LLWB [26] proposes an architectural approach to reduce write backs to NVM and combine DRAM as a layer of cache. Ferreira et al. [13] 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 don't 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. [7] use an age-based mechanism to guide the migration and re-mapping of frequently written pages. WAFA [8] allocates memory resources in a rotational manner for fine-grained wear leveling of NVM. Kevlar [16] shuffles pages in NVM to avoid intensive writes to fixed cells. Security Refresh [28] is a hardware-based approach with runtime randomization to mitigate the wear-out vulnerability. Start-Gap [25] 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 [44] reallocates `inodes` which are frequently updated in NVM. Also, both wear reduction and leveling can be combined for further prolonging the lifetime of NVM [16, 26].

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.

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

[1] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 62–77. https://doi.org/10.1145/3192366.3192392

[2] Apple. 2021. About the Rosetta Translation Environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment.

[3] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 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* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 645–659. https://doi.org/10.1145/3037697.3037738

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(USENIX ATC '05)*. USENIX Association, USA, 41.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, USA, 265–275.

[6] Derek L. Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. USA. AAI0807735.

[7] Chi-Hao Chen, Pi-Cheng Hsiu, Tei-Wei Kuo, Chia-Lin Yang, and Cheng-Yuan Michael Wang. 2012. Age-Based PCM Wear Leveling with Nearly Zero Search Cost. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 453–458. https://doi.org/10.1145/2228360.2228439

[8] Xianzhang Chen, Zhuge Qingfeng, Qiang Sun, Edwin H.-M. Sha, Shouzhen Gu, Chaoshu Yang, and Chun Jason Xue. 2019. A Wear-Leveling-Aware Fine-Grained Allocator for Non-Volatile Memory. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 116, 6 pages. https://doi.org/10.1145/3316781.3317752

[9] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 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* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, USA, 15–24.

[10] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 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* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/2150976.2151004

[11] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *2009 46th ACM/IEEE Design Automation Conference*. IEEE, 664–669.

[12] Peter Feiner, Angela Demke Brown, and Ashvin Goel. 2012. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2150976.2150992

[13] Alexandre P Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. 2010. Increasing PCM main memory lifetime. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 914–919.

[14] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

[15] Tiejun Gao, Karin Strauss, Stephen M. Blackburn, Kathryn S. McKinley, Doug Burger, and James Larus. 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/2491956.2462171

[16] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2019. Software Wear Management for Persistent Memories. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 45–63.

[17] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proc. VLDB Endow.* 14, 5 (Jan. 2021), 785–798. https://doi.org/10.14778/3446095.3446101

[18] Shiliang Hu and James E. Smith. 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* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 213.

[19] Jinhu Jiang, Rongchao Dong, Zhongjun Zhou, Changheng Song, Wenwen Wang, Pen-Chung Yew, and Weihua Zhang. 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)*. 415–426. https://doi.org/10.1109/MICRO50266.2020.00043

[20] Daixuan Li, Benjamin Reidys, Jinghan Sun, Thomas Shull, Josep Torrellas, and Jian Huang. 2021. UniHeap: Managing Persistent Objects across Managed Runtimes for Non-Volatile Memory. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '21)*. Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/3456727.3463775

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

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

[23] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[24] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 12, 16 pages. https://doi.org/10.1145/2901318.2901324

[25] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. 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* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 14–23. https://doi.org/10.1145/1669112.1669117

[26] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[27] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[28] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. 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* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 383–394. https://doi.org/10.1145/1815961.1816014

[29] Changheng Song, Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Weihua Zhang. 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* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 77–89.

[30] Chenxi Wang, Ting Cao, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. 2016. Efficient Management for Hybrid Memory in Managed Language Runtime. In *Network and Parallel Computing*, Guang R. Gao, Depei Qian, Xinbo Gao, Barbara Chapman, and Wenguang Chen (Eds.). Springer International Publishing, Cham, 29–42.

[31] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

[32] Wenwen Wang. 2021. Helper Function Inlining in Dynamic Binary Translation. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3446804.3446851

[33] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2018. 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* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 84–97. https://doi.org/10.1145/3173162.3177160

[34] Wenwen Wang, Chenggang Wu, Tongxin Bai, Zhenjiang Wang, Xiang Yuan, and Huimin Cui. 2014. A Pattern Translation Method for Flags in Binary Translation. *Journal of Computer Research and Development* 51, 10 (2014), 2336–2347. http://crad.ict.ac.cn/EN/10.7544/issn1000-1239.2014.20130018

[35] Wenwen Wang, Jiacheng Wu, Xiaoli Gong, Tao Li, and Pen-Chung Yew. 2018. Improving Dynamically-Generated Code Performance on Dynamic Binary Translators. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA) *(VEE '18)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/3186411.3186413

[36] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2016. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 591–603.

[37] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 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* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3368826.3377919

[38] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (Niagara Falls, New York, USA) *(MobiSys '17)*. Association for Computing Machinery, New York, NY, USA, 319–331. https://doi.org/10.1145/3081333.3081337

[39] Wen Wen, Youtao Zhang, and Jun Yang. 2018. Wear Leveling for Crossbar Resistive Memory. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 58, 6 pages. https://doi.org/10.1145/3195970.3196138

[40] Jin Wu, Jian Dong, Ruili Fang, Wenwen Wang, and Decheng Zuo. 2020. PerfDBT: Efficient Performance Regression Testing of Dynamic Binary Translation. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 389–392. https://doi.org/10.1109/ICCD50377.2020.00071

[41] Jin Wu, Jian Dong, Ruili Fang, Ziyi Zhao, Xiaoli Gong, Wenwen Wang, and Decheng Zuo. 2021. Effective Exploitation of SIMD Resources in Cross-ISA Virtualization. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 84–97. https://doi.org/10.1145/3453933.3454016

[42] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the Challenges of Crossbar Resistive Memory Architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 476–488. https://doi.org/10.1109/HPCA.2015.7056056

[43] Jian Xu and Steven Swanson. 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* (Santa Clara, CA) *(FAST'16)*. USENIX Association, USA, 323–338.

[44] Chaoshu Yang, Duo Liu, Runyu Zhang, Xianzhang Chen, Shun Nie, Fengshun Wang, Qingfeng Zhuge, and Edwin H.-M. Sha. 2020. Efficient Multi-Grained Wear Leveling for Inodes of Persistent Memory File Systems. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference* (Virtual Event, USA) *(DAC '20)*. IEEE Press, Article 98, 6 pages.

[45] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile Gaming on Personal Computers with Direct Android Emulation. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 15 pages. https://doi.org/10.1145/3300061.3300122

[46] Hung-Chang Yu, Kai-Chun Lin, Ku-Feng Lin, Chin-Yi Huang, Yu-Der Chih, Tong-Chern Ong, Jonathan Chang, Sreedhar Natarajan, and Luan C. Tran. 2013. Cycling endurance optimization scheme for 1Mb STT-MRAM in 40nm technology. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. 224–225. https://doi.org/10.1109/ISSCC.2013.6487710

[47] Ziyi Zhao, Zhang Jiang, Ying Chen, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2021. Enhancing Atomic Instruction Emulation for Cross-ISA Dynamic Binary Translation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 351–362. https://doi.org/10.1109/CGO51591.2021.9370312

[48] Ziyi Zhao, Zhang Jiang, Ximing Liu, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2020. DQEMU: A Scalable Emulator with Retargetable DBT on Distributed Platforms. In *49th International Conference on Parallel Processing - ICPP* (Edmonton, AB, Canada) *(ICPP '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 11 pages. https://doi.org/10.1145/3404397.3404403