Loop2Recursion: Compiler-Assisted Wear Leveling for Non-Volatile Memory

Wei Li*, Libing Wu†‡, Mengting Yuan*, Chun Jason Xue§, Jingling Xue¶, Qingan Li*✉
*School of Computer Science, Wuhan University, China
†School of Cyber Science and Engineering, Wuhan University, China
‡Shenzhen Research Institute of Wuhan University, Shenzhen, China
§Department of Computer Science, City University of Hong Kong, Hong Kong
¶School of Computer Science and Engineering, UNSW Sydney, Australia
{weili, wu, ymt}@whu.edu.cn, jasonxue@cityu.edu.hk, jingling@cse.unsw.edu.au, qingan@whu.edu.cn

***Abstract*—Non-Volatile Memory (NVM) technologies, such as Phase Change Memory (PCM), herald the next generation of main memory as they offer superior features compared with DRAM. Unfortunately, NVM's limited write endurance hinders its adoption as its lifetime can be extremely short under skew writes. This paper observes that the loops in programs are one of the primary causes of uneven writes as they introduce the hot data and cause a large number of stack frames to be allocated to the same locations. To alleviate this problem, we present Loop2Recursion, a compile-time wear leveling technique for transforming loops into recursions automatically. Our approach is flexible as it can avoid a substantial memory overhead by limiting the depth of recursion. Experimental results demonstrate that Loop2Recursion can significantly improve the wear leveling over stack area compared to the state-of-the-art methods, while incurring only negligible performance overhead.**

***Index Terms*—NVM, wear leveling, loop2recursion**

I. Introduction

Emerging Non-Volatile Memory (NVM) technologies, such as Phase Change Memory (PCM) [1], feature high density, low standby power, storage-like persistence, byte addressability and DRAM comparable performance [2]. These characteristics make NVM a competitive candidate for the next generation of main memory. However, NVM suffers from limited write endurance. An NVM cell will wear out after a certain number of writes. For example, each PCM cell is only expected to endure 10^7 – 10^9 writes [3]. In the absence of wear management, NVM's lifetime can be as low as 1.1 months [4]. To address the endurance issue and prolong NVM's lifetime, system designers and software developers must manage wear carefully and prevent intensive writes wearing a small proportion of NVM cells out prematurely.

In the past decade, many wear leveling techniques that spread writes uniformly over the entire NVM space have been proposed at various levels, e.g., at the hardware level, Operating System (OS) level and program level. Each level is somehow orthogonal to and capable of working in tandem with one another. Specifically, hardware wear leveling is achieved by shifting or swapping data between physical locations at different granularities (e.g. memory line [3] or page [5]). Finer granularities provide better wear leveling while incurring higher storage and performance overhead. In contrast, OS-level techniques focus on page-level wear leveling by employing wear-aware page allocation, replacement and swapping schemes [4], [6]. However, most OS-level techniques neglect the wear imbalance inside a page and involve additional hardware to track page wear. Program-level techniques aim at distributing writes uniformly in program's runtime memory by optimizing data allocation and access patterns [7]–[9]. These techniques avoid the need for hardware and OS support and therefore provide a low-cost and effective wear leveling solution, although they are somewhat limited due to the requirement on the availability of the source code. Program-level wear leveling can be especially important when NVM technologies are applied to embedded systems without a Memory Management Unit (MMU) or caches, such as Cortex-M3/M4 [10] processor-based systems, as hardware level and OS-level wear leveling techniques would become substantially more inefficient and impractical in such cases.

Existing program-level techniques are mainly concerned with wear leveling for the *heap* and *stack* segments of a program. For stack wear leveling, a *dynamic stack* was put forward to allocate *stack frames* dynamically like allocating heap objects [8], [9]. However, this technique suffer from two limitations. First, when some variables are frequently updated within a stack frame (referred to as "hot" data), a large number of writes will still converge to a few fixed locations. Second, allocating stack frames with a dynamic allocator imposes significant performance overhead.

To address these two issues, this paper focuses on optimizing the loops in programs for stack wear leveling, as loops are often the most computation intensive part of applications. We observe that (1) a loop usually changes a few variables iteratively, inducing the hot data and uneven wear inside stack frames, and (2) a loop may contain some function calls, causing a large number of stack frames to be allocated on the same memory location and thus exacerbating the wearing of specific stack addresses. Based on these two observations, we propose Loop2Recursion, a compile-time wear leveling technique for NVM. Loop2Recursion eliminates hot data inside stack frames and mitigates the imbalance of stack frame distribution by automatically transforming loops into recursive functions. Furthermore, considering that completely transforming loops into recursions may increase the stack memory footprint significantly and even lead to stack overflow, we propose to optimize for stack memory usage by adjusting the depth of recursions during the transformation.

Loop2Recursion operates on a low-level Intermediate Representation (IR) [11], and thereby is generalizable to multiple programming languages. Being also independent of hardware and operation systems, Loop2Recursion can also cooperate with other OS-level or hardware-level wear leveling techniques. We evaluate its wear leveling effectiveness and impact on performance. Experimental results demonstrate that, compared with no wear leveling and the state-of-the-art *dynamic stack* [9], Loop2Recursion reduces the number of writes to the hottest spot on the stack by 91.6% and 50.1% (on average), respectively, at negligible performance overhead.

To summarize, the major contributions of this paper include:

- An empirical study showing that loops in programs are apt to cause an unbalanced write distribution;
- Loop2Recursion, a compiler-assisted wear leveling technique that achieves stack wear leveling by automatically transforming loops into recursions; and
- an optimization scheme for dealing with the increased stack memory usage caused by Loop2Recursion.

II. Background and Motivation

A. Background

Stack memory allocation: For the *stack* segment of a program, memory is allocated to *stack frames* through adjusting the *stack pointer* (*sp*) register. Each *stack frame* corresponds to a function instance which has not yet terminated with a return, holding the return address, incoming parameters, local variables and saved registers. Every time when a function is invoked/returned, its frame is allocated/deallocated. The allocation/deallocation happens on contiguous memory blocks in a Last-In-First-Out (LIFO) order. Thus, the most recently allocated frame is always the next to be freed. Due to this mechanism, a large disparity exists in stack frame distribution as some memory regions are associated with a relatively large number of frames while some others are rarely used, which in turn leads to the uneven writes.

Stack wear leveling: Recently, compiler-assisted techniques are proposed for stack wear leveling. Q. Li et al. [8] proposed a *dynamic stack*, where stack frames are allocated dynamically in a way analogous to heap allocation. Specifically, every time a function is called, a stack allocator based on the *next-fit* policy is immediately employed to obtain a free memory area for its frame. W. Li et al. [9] further improved the *dynamic stack* by employing a new wear-aware memory allocator, considering that the *next-fit* allocator is suboptimal for wear leveling.

Despite the fact that the *dynamic stack* enables stack frames to be more uniformly distributed, some limitations and shortcomings still exist. Firstly, this method regards each stack frame as a whole without handling the wear inside the frame. Actually, the inside wear is far from equilibrium, since frequently-updated variables will produce intensive writes on their corresponding memory locations. Additionally, the *dynamic stack* incurs significant performance overhead that is positively correlated to the number of function calls, owing to the additional allocation and deallocation operations.

B. Motivation

Uneven wear inside stack frames: Excessive writes inside stack frames commonly arise from loops. Fig. 1 shows a code snippet selected from the benchmark *dijkstra* in Mibench [12]. A loop of 100 iterations is contained in $main()$ function. The local variables i and j are anticipated to be updated for 101 and 201 times, respectively. On a cacheless architecture, each update may cause a write to memory if the register spills. Even when write-back caches are used, these writes may not be all avoided as cache conflicts will frequently occur in the worst-case scenario. Consequently, the number of writes on memory locations of i and j is much higher than the others inside the stack frame. In the real case of a large register file, i and j are quite likely kept in registers and no register spilling occur. But there are still many situations hot variables cannot be stored in registers, such as address taken. Fig. 7 in Section V-B suggests that most programs suffer from the uneven wear inside stack frames. Tens of thousands of writes may concentrate on a few memory locations inside one stack frame.

Impact of loops on stack frame distribution: As mentioned in Section II-A, conventional stack frames are allocated in a LIFO order, due to which function calls inside loops will cause the callee functions in each iteration to be allocated on the same memory locations of stack. For example, in the code snippet shown in Fig. 1, function $dijkstra()$ is called inside the loop (line 9). For each execution of $dijkstra()$, its stack frame is allocated on top of $main()$'s, as shown in Fig. 3 (a). As a result, the corresponding memory locations carry out an enormous number of stack frames, leading to an extremely unbalanced stack frame distribution.

A motivating example: Based on the discussion above, we conclude that the loops in a program are one of the primary causes for uneven wear on the stack. To mitigate this problem, we can transform the loops into recursive functions.

Let us consider a motivating example. Fig. 2 shows a code snippet equivalent to the example shown in Fig. 1, which replaces the loop with a new recursive function $main\_loop$. The original frequently-updated variables i and j are used for the parameters of the recursive function, and the loop

```
1  #define NUM_NODES 100
2  int dijkstra(int chStart, int chEnd);
3  int main(int argc, char *argv[]) {
4      … // omitted code
5      int i, j;
6      for (i = 0, j = NUM_NODES / 2; i < 100; i++, j++) {
7          j = j % NUM_NODES;
8          dijkstra(i, j);
9      }
10     … // omitted code
11 }
```

Fig. 1: A snippet of code in *dijkstra* from MiBench [12].

```c
#define NUM_NODES 100
int dijkstra(int chStart, int chEnd);
void main_loop(int i, int j) {
    if (i < 100) {
        j = j % NUM_NODES;
        dijkstra(i, j);
        main_loop(i +1, j + 1);
    } else
        return;
}
int main(int argc, char *argv[]) {
    … // omitted code
    int i = 0, j = NUM_NODES / 2;
    main_loop(i, j);
    … // omitted code
}
```

Fig. 2: Recursive version of the code snippet in Fig. 1.

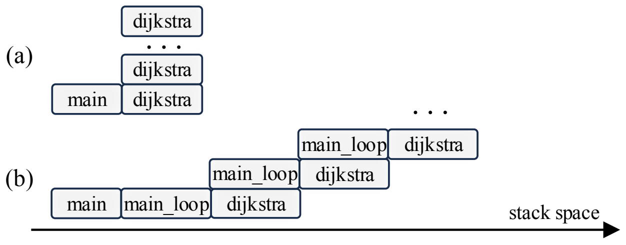


Fig. 3: Comparison between stack memory allocations, with the stack frames on the same column are collocated on the same memory location. (a) stack allocation for the code snippet in Fig. 1. (b) stack allocation for the code snippet in Fig. 2.

condition $i < 100$ is taken as the recursive condition. In this code snippet, each local variable needs a limited number of updates, enabling wear leveling inside stack frames. In addition, each time $main\_loop()$ is recursively called, its stack frame is allocated on the top of stack, as shown in Fig. 3 (b). Accordingly, the stack frames of $dijkstra()$, which were originally collocated, are allocated on different memory locations, enabling a uniform stack frame distribution. Thus, the recursion provides better wear leveling than the loop.

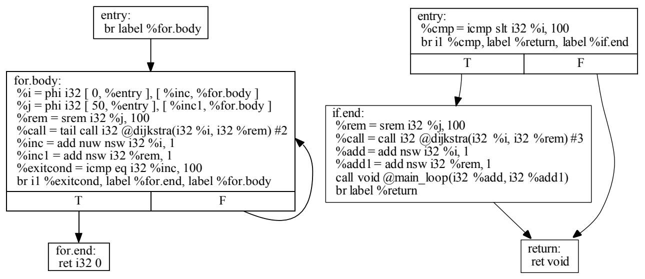
III. Loop2Recursion

In this section, we detail our proposed wear-leveling technique—Loop2Recursion, which automatically transforms loops into equivalent recursions.

A. Overview of Loop2Recursion

Loop2Recursion performs transformations on a low-level, Static Single Assignment (SSA) [13] based Intermediate Representation (IR). Specifically, LLVM [11] IR is used in this study.

Some higher-level information, such as Control Flow Graphs (CFGs), can be built from an IR representation of a program. Fig. 4 shows two IR examples in a CFG representation, corresponding to the $main()$ function in Fig. 1 and $main\_loop()$ function in Fig. 2, respectively. In a CFG, a function is a set of basic blocks, where each basic block is a sequence of instructions. The edges between basic blocks represent control flow paths. A loop is a sequence of basic blocks in a function's graph that constitute a directed cycle.


```
entry:
 br label %for.body

for.body:
 %i = phi i32 [ 0, %entry ], [ %inc, %for.body ]
 %j = phi i32 [ 50, %entry ], [ %inc1, %for.body ]
 %rem = srem i32 %j, 100
 %call = tail call i32 @dijkstra(i32 %i, i32 %rem) #2
 %inc = add nuw nsw i32 %i, 1
 %inc1 = add nsw i32 %rem, 1
 %exitcond = icmp eq i32 %inc, 100
 br i1 %exitcond, label %for.end, label %for.body

for.end:
 ret i32 0
```

```
entry:
 %cmp = icmp slt i32 %i, 100
 br i1 %cmp, label %return, label %if.end

if.end:
 %rem = srem i32 %j, 100
 %call = call i32 @dijkstra(i32 %i, i32 %rem) #3
 %add = add nsw i32 %i, 1
 %add1 = add nsw i32 %rem, 1
 call void @main_loop(i32 %add, i32 %add1)
 br label %return

return:
 ret void
```

Fig. 4: LLVM IR examples represented as Control Flow Graphs (CFGs). (a) IR for $main()$ function in Fig. 1, where the *for* loop is by default optimized into *do-while* format by LLVM. (b) IR for $main\_loop()$ function in Fig. 2.

The head block is called a *loop header* and dominates all the other blocks in the loop. Some basic blocks with outgoing edges from loop nodes are called *exit basic blocks* of the loop. For example, the CFG shown in Fig. 4 (a) contains a loop, which consists of a single basic block *for.body*. The basic block *for.body* is the loop header and the block *for.end* is the exit basic block.

Our IR-based Loop2Recursion exhibits many advantages. First, the higher-level information in the IR facilitates the transformation process from loops to recursions. Second, as an IR is independent of any specific source or target language, Loop2Recursion is applicable to various programming languages. Furthermore, Loop2Recursion only needs to focus on the transformation for the low-level representation of loops without being concerned with the loop types (for/while/do-while) in source programs as well as the control statements (break/continue/return) inside loops, because all of them have been mapped to control flow paths in an IR.

Loop2Recursion processes all the loops in an input IR sequentially. Nested loops are processed separately inside out. Note that not all loops will induce hot spots on the stack and need to be transformed. Loop2Recursion will first analyze the loop being processed and decide whether or not to transform this loop. Then the transformation is performed only if needed. The transformation process can be further divided into three steps: (1) the preparation for the transformation, (2) generating the recursive function, and (3) substituting the recursion for the loop. Lastly, Loop2Recursion may still fail to eliminate some hot data when a loop includes some pointer operations. Special handling will be provided in such a case.

Below we describe each of these steps in detail.

B. Loop Analysis

To minimize the performance and memory overhead, we do not transform loops that will certainly not lead to uneven wear on the stack. Such loops meet the following two requirements related to the causes of the uneven stack frame distribution and the hot data as stated in Section II-B: (1) No function call is contained, and (2) No variable write is involved or all the writes take place in registers.

A loop analysis is performed to check these requirements. The first requirement can be checked easily by iterating all the instructions in the loop. But for the second requirement, some lower-level information of the loop is needed, as an IR assumes an infinite set of pseudo-registers and cannot suggest which variable writes will finally occur in actual registers. Therefore, we perform a two-stage compilation here. The first stage completes the *register allocation*, which assigns variables to actual registers and manages data transfer between registers and memory, so that we can detect if any memory write exists inside each loop. The second stage then performs IR-level transformations for loops that fail to meet the requirements.

C. Preparation for transformation

This step includes fetching parameters and catching return values for the recursive function to be generated.

Fetching parameters: A loop generally uses some local variables defined outside its scope, such as i and j in the code snippet in Fig. 1. To make these variables accessible inside the recursive function, we choose to pass them as arguments to the recursive function.

In SSA form, we divide parameters into two classes: variant parameters and invariant parameters. The former are updated in each iteration while the latter remain unchanged for the whole loop. The two kinds of parameters are fetched in different ways.

To fetch the variant parameters, we only need to find the ϕ (phi) nodes in loop header, such as $\%i$ and $\%j$ in block $for.body$ in Fig. 4 (a). They correspond to the variables that have an initialized value (0 for $\%i$ and 50 for $\%j$) in the first iteration and are updated with a new value ($\%inc$ for $\%i$ and $\%inc1$ for $\%j$) in subsequent iterations. After transforming a loop into a recursive function, the initialized values need to be passed as arguments when first calling the recursive function in the loop's parent function (e.g. $main()$ function for the loop in Fig. 4 (a)), and the new values are used as arguments in the following recursive calls.

To fetch the invariant parameters, we iterate through the instructions in the loop and check all their operands. Those operands whose definitions are outside the loop belong to the invariant parameters, and need to be passed to the recursive function via the call from the loop's parent function. As for the recursive call inside the recursive function, the incoming invariant parameters are passed on.

Catching return values: In SSA form, some variables defined inside the loop may be used outside. For example, $\%inc$ and $\%inc1$ in Fig. 4 (a) represent the modified i and j after each loop iteration, and they will be used outside if the counterparts i and j are used after the loop in the source code. After transforming the loop into a recursive function, these variables will be inaccessible outside and thus need to be returned.

To catch return values, we iterate through all the instructions inside the loop. For each instruction corresponding to a variable definition, we obtain this variable's uses from the Definition-Use Chain built from the IR. The variable will be returned if it has one or more uses outside the loop.

In addition, a loop may have multiple exit basic blocks, which can happen when the loop contains return statements or goto statements. In this circumstance, we assign a unique ID to each exit block, and return an additional integer value (an *exit block indicator*) to indicate which exit block is going to be branched into when the generated recursive function terminates.

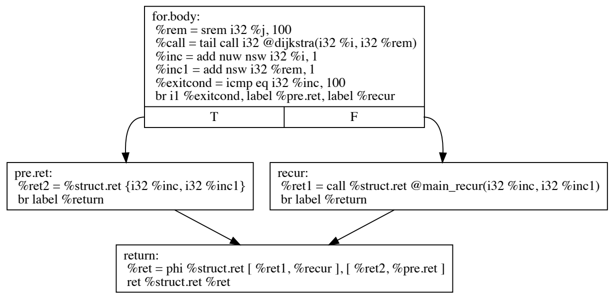
D. Generating recursive function

Since we have identified the parameters and return values, we can create a recursive function with the corresponding parameters' types and return type. For multiple return values, a structure type is used as a wrapper.

To ensure its equivalence to the original loop, the recursive function should preserve all the basic blocks in the loop and maintain the consistency of control flow. It should recursively call itself or return when the control flow originally branches back to the loop header or exits the loop. For example, Fig. 5 shows the CFG of a recursive function equivalent to the loop shown in Fig. 4 (a).

The following steps detail the process of generating the recursive function:

- Firstly, move all the basic blocks from the loop into this function (The loop header must be the entry block of the recursive function). Remove the ϕ nodes in the loop header and replace the uses of the variables defined outside the loop with the function's parameters.
- Create a basic block named $recur$, in which a recursive call to the function itself is contained. Proper arguments are passed to the call according to the parameters type as described in the previous subsection, and a value named $ret1$ is returned from this call.
- Create a basic block named $pre.ret$, in which a return value $ret2$ is set up. If multiple values need to be returned, $ret2$ will be of a structure type, and each element in $ret2$ will be assigned separately. If there are multiple exit blocks, a ϕ node is used to select an exit block ID for an *exit block indicator*.
- Create a basic block named $return$, and insert a branch instruction that branches to the $return$ block at the end of $recur$ and $pre.ret$ blocks. A ϕ node is used in the $return$ block to select a value from $ret1$ and $ret2$ as the final return value.



```
for.body:
 %rem = srem i32 %j, 100
 %call = tail call i32 @dijkstra(i32 %i, i32 %rem)
 %inc = add nuw nsw i32 %i, 1
 %inc1 = add nsw i32 %rem, 1
 %exitcond = icmp eq i32 %inc, 100
 br i1 %exitcond, label %pre.ret, label %recur
```

```
pre.ret:
 %ret2 = %struct.ret {i32 %inc, i32 %inc1}
 br label %return
```

```
recur:
 %ret1 = call %struct.ret @main_recur(i32 %inc, i32 %inc1)
 br label %return
```

```
return:
 %ret = phi %struct.ret [ %ret1, %recur ], [ %ret2, %pre.ret ]
 ret %struct.ret %ret
```

Fig. 5: CFG for the recursive function $main\_recur(i32\ \%i, i32\ \%j)$ which is equivalent to the loop in Fig. 4 (a). In this example, we suppose that $\%inc$ and $\%inc1$ need to be returned.

- Update the terminator instructions of the basic blocks in the recursive function. Replace the branch target with $recur$ or $pre.ret$ if it originally branches to the loop header or an exit block.

E. Substituting the recursion for the loop.

To substitute the loop by recursion, we first find the predecessor block of the loop header, then replace the branch to the loop header in it with a call to the recursive function. The return values of the recursion will replace the uses of the variables defined inside the original loop. Lastly, if the original loop only has one single exit block, we insert a branch instruction that branches to this exit block at the end of the predecessor block. Otherwise, we need to branch to a specific exit block according to the returned *exit block indicator*.

F. Handlings for cases with pointers

At this point, we have successfully transformed a loop into a recursive function. However, the hot data issue may still remain if a loop iteratively dereferences a pointer and stores values to the pointee. For this simple C fragment:

```
int *p = &i;
while (cond) { (*p)++; }
```

a pointer p is allocated and set to point to the variable i. Each time through the *while* loop, p is dereferenced and its pointee, i, is incremented. As a result, the number of writes on i's memory location equals to the number of iterations, causing i to become hot.

The aforementioned Loop2Recursion will transform this *while* loop into a recursive function that semantically like this:

```
void while_recur(int *p) {
    if (cond) {
        (*p)++;
        while_recur(p);
    } else
        return;
}
```

the pointer p is fetched as an invariant parameter. All the store operations still perform on i's memory location, resulting in a failure for eliminating the hot data.

To mitigate this problem, we propose to copy the pointer's pointee before the recursive call in the recursive function, and then use the reference to the copied variable as the argument in the recursive call instead of the current pointer. When the call returns, we copy the value back to ensure the pointee's final value to be identical to the correct one.

```
int copy = *p;
while_recur(&copy);
*p = copy;
```

***Aliasing* issue:** When dealing with pointers, we should also consider the *pointer aliasing* thoroughly, as copying a pointee referred by multiple pointers is error-prone. We employ a *flow-sensitive pointer analysis* in this study to analyse the aliasing relations of pointers and handle different aliasing conditions carefully. Below, we discuss our copy scheme in detail.

First, for each pointer fetched as a parameter of the transformed recursive function, we determine whether its pointee needs to be copied. Based on the conservative but safe philosophy, we do not execute the copy operation in the following cases. (1) The pointer is updated with each iteration, which indicates that it would barely store values to the same object; (2) There is no store to the pointer's pointee in the loop, so that the pointee is constant without introducing any write; (3) The pointee's type is uncertain, as the points-to set of the pointer may include both scalars and arrays; (4) The pointer may alias non-local variables, including global variables, heap objects or variables defined in other functions.

Next, we perform the copy operations for the candidate pointers under different conditions of pointer aliasing.

***No-alias* condition:** If a pointer is not aliased, its pointee can be directly copied like the example above.

***Must-alias* condition:** If two or more pointers must alias with each other, they will refer to the same pointee. We only need to copy this pointee once and replace these pointers with the reference to the copied value in the recursive call. The copy back operation only needs to be performed once as well.

***May-alias* condition:** If two pointers have a *may-alias* relation, we need to add a dynamic alias check at run-time. Let us consider the code below as an example:

```
int *p = &i, *q = &j;
if (cond1) { q = p; }
while (cond2) { (*p)++; (*q)++; }
```

p and q alias only when $cond1$ holds, which cannot be conclusively determined at compile time. As different aliasing relations will induce different copy operations, we add a dynamic alias check in the transformed recursive function:

```
if (p == q) {
    int copy_p = *p;
    while_recur(&copy_p, &copy_p);
    *p = copy_p;
} else {
    int copy_p = *p, copy_q = *q;
    while_recur(&copy_p, &copy_q);
    *p = copy_p, *q = copy_q;
}
```

This dynamic check works well for the circumstance where only two pointers may alias. However, when there are more pointers that may alias with each other, the number of required dynamic check conditions increases exponentially, resulting in a significant increase in code size and a performance degradation. Therefore, we only process the two pointers' case in this paper, leaving code with more pointers may aliasing unoptimized but still correct. Our evaluation shows that this suffices for eliminating most hot data.

IV. Reducing Stack Memory Usage For Loop2Recursion

Although recursions exhibit better wear leveling than loops, they may substantially increase the stack memory consumption. Without any control, excessively deep recursions may exhaust all the available stack memory and raise a stack overflow error. To reduce the stack memory usage, we propose to employ a compiler option to limit the depth of recursions, so programmers can flexibly exploit a good trade-off between wear leveling and stack expansion, as depicted in Section V-F.

```
1  #define DEPTH_LIMIT 20
2  struct RetType { int i; int j; };
3  struct RetType main_loop(int i, int j, int depth) {
4      if (i < 100 && depth < DEPTH_LIMIT) {
5          j = j % NUM_NODES;
6          dijkstra(i, j);
7          return main_loop(i + 1, j + 1, depth + 1);
8      } else {
9          struct RetType ret = { i, j };
10         return ret;
11     }
12 }
13 int main(int argc, char *argv[]) {
14     … // omitted code
15     int i, j;
16     for (i = 0, j = NUM_NODES / 2; i < 100; ) {
17         struct RetType ret = main_loop(i, j, 0);
18         i = ret.i; j = ret.j;
19     }
20     … // omitted code
21 }
```
Fig. 6: A code snippet equivalent to the example shown in Fig. 1. The recursion depth is limited to 20 in this example.

To limit the depth of recursions, we stop recursing after a certain number of recursive calls, and make the recursive function iteratively invoked so that the total number of times executing the body of the recursive function is identical to that of the original loop iterations.

We now illustrate this scheme by continuing with the loop example shown in Fig. 1. The depth-limited recursive function transformed from this loop will be like $main\_loop()$ shown in Fig. 6. In this recursion, an extra parameter $depth$ is added, representing the depth of the recursive call at runtime. This parameter is initialized with 0 when the recursive function is first called (line 17) and incremented by 1 when passed to the recursive call within this function (line 7). A depth-limiting condition is added to the current recursive condition (line 4). The if-body gets executed only when both the conditions hold. Therefore, the maximum depth of this recursion can be guaranteed to be under a predetermined threshold value (20 in this example). In the loop's parent function $main()$, the loop body is replaced with an invocation of the recursive function (lines 17 and 18). The for loop's update statements are also deleted as they are essentially a part of the loop body and will be executed in the recursion.

V. Experiment

A. Experimental Setup

We implement Loop2Recursion as an LLVM function pass based on LLVM 3.8.0. To evaluate the effectiveness, we compare Loop2Recursion with a baseline without wear leveling management (no WL) and the state-of-the-art *dynamic stack* [8], [9], in terms of wear leveling, performance and stack memory usage. For *dynamic stack*, the wear-aware memory allocator UWLalloc [9] is employed which limits the number of stack frames allocated on each same memory address. The allocation limit is set to 300 in this experiment. For Loop2Recursion, we limit the maximum depth of recursions to 20 to reduce stack memory usage.

Programs from Mibench [12] are selected for evaluation and compiled at the optimization level 'O2' for each evaluated method. In particular, we disable tail-call optimization when applying Loop2Recursion to prevent the generated recursive functions being transformed back to loops.

We develop a Pin-based [14] tool to trace the writes on the stack for evaluation. Both cacheless and cache-enabled architectures are evaluated. For the cache-enabled architecture, we implement a Pin-based cache simulator, simulating a 4KB, 64-way associative data cache which references the parameters of ARM940T [15]. All the evaluations are conducted on linux (kernel version 5.3.0) with an Intel Core i5-4278U 2.60 GHz CPU and 8 GB of DRAM memory (as a proxy of NVM).

B. Wear Leveling on Cacheless Architecture

As stated in Section II, loops in programs generally lead to the uneven wear inside stack frames and non-uniform stack frame distribution on the stack. We first investigate the effect of Loop2Recursion in solving the two issues respectively. Then we evaluate the overall wear leveling effectiveness. In this case, a cacheless architecture is assumed.

Wear inside stack frames: We compare the maximum number of writes on one address inside stack frames among the baseline, *dynamic stack* and Loop2Recursion, as shown in Fig. 7. Since the *dynamic stack* focuses on wear-aware memory allocation for stack frames only, the wear inside each stack frame is the same as that without wear leveling management. In most cases, Loop2Recursion drastically decreases the wear inside stack frames by tens to hundreds times.

The benchmark worth noticing here is $bitcount$, in which the result increases from 16 to 3751 after applying Loop2Recursion. This is caused by the return values of the generated recursive functions. When limiting the recursion depth, the recursive function will be iteratively invoked, and the return values for each iteration of recursive invocation are allocated on the same memory address inside the caller's stack frame. But this increase will not impair the final wear leveling performance, since this increased number is never larger than the loop bound divided by the depth limitation. Without transforming loops into recursions, a function call inside a loop will incur a much more severe hot spot where the number of writes equals or exceeds the loop bound.

Stack frame distribution: Fig. 8 compares the maximum number of stack frames allocated on one memory address. The *dynamic stack* achieves a more uniform stack frame distribution, as it strictly limits the allocation count on each memory location (300 in this experiment). Compared to the baseline, Loop2Recursion reduces the maximum stack frame count considerably, especially for $blowfish$, in which the count drops about 99.5%.

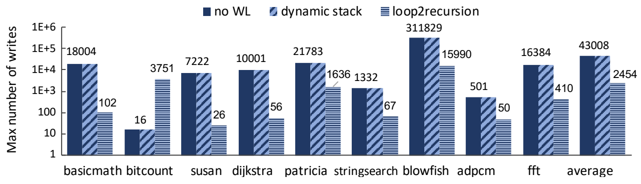


Fig. 7: Maximum number of writes on one memory address in stack frames. The last column represents the average number.

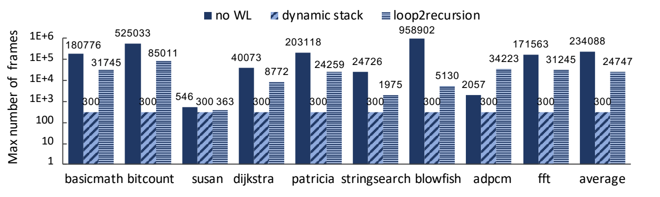


Fig. 8: Maximum number of stack frames allocated on the same memory location.

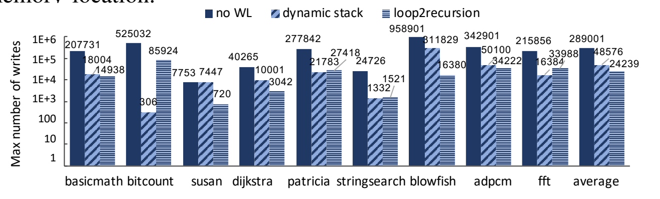


Fig. 9: Number of writes on the hottest memory address on the stack.

However, $adpcm$ is noteworthy. The generated recursive functions account for this exception. When a loop is transformed to an depth-limited recursion, the recursive function is iteratively called and each iteration will start from the same address. When this loop contains hot data without function calls, the stack frame count on the address will increase. However, the increased number is definitely less than the the number of writes caused by the hot data, so it will not impact the final wear leveling performance.

Wear leveling effectiveness: Due to the fact that NVM's lifetime is determined by the worst wear degree, we evaluate the overall wear leveling effectiveness by comparing the number of writes on the most-written memory address among the entire stack area as shown in Fig. 9. Both the *dynamic stack* and Loop2Recursion significantly decrease the write count compared to the baseline. For most benchmarks, Loop2Recursion achieves a similar or a better result than the *dynamic stack*. On average, Loop2Recursion reduces 91.6% and 50.1% of the maximum write count compared to the baseline and *dynamic stack*, theoretically improving the achievable lifetime of NVM by 11.4x and 1.9x, respectively. As a result, Loop2Recursion exhibits higher wear leveling effectiveness.

C. Wear Leveling on Cache-enabled Architecture

When a write-back cache is used, a memory write only happens when a modified cache line is written back to the memory. To evaluate the wear leveling performance, we compare the number of writebacks to the hottest memory address between the baseline and Loop2Recursion.

As shown in Fig. 10, some benchmarks ($bitcount$, $susan$, and $adpcm$) exhibit extremely high locality, so there are only a few writebacks to the hottest address. After applying Loop2Recursion, the program's locality declines, resulting in the increase of writeback count. But the increase has only a negligible impact since the writeback count is fairly small.

For other benchmarks, the number of writebacks is comparatively higher. Especially in $patricia$, there are more than ten thousand of writes to one memory address. We can see that Loop2Recursion is still effective to improve the wear leveling for most of these benchmarks. On average, the maximum writeback count is reduced nearly by half.

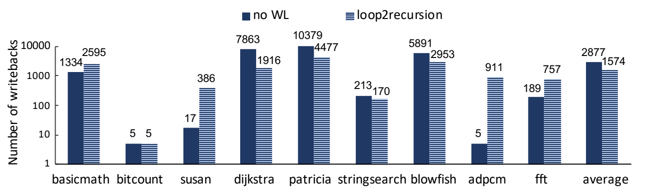


Fig. 10: Number of writes to the hottest memory address.

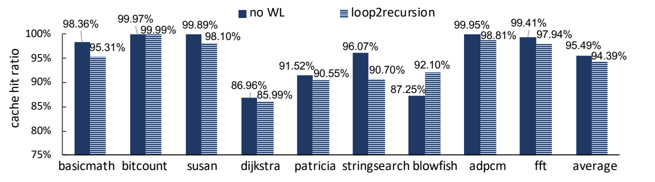


Fig. 11: Cache hit ratio.

We further compare cache hit ratio between the baseline and Loop2Recursion, as shown in Fig. 11. For most benchmarks, the cache hit ratio drops slightly after applying Loop2Recursion, thus Loop2Recursion has a negligible impact on the cache performance. In conclusion, Loop2Recursion can also work well for a cache-enabled architecture, as it reduce the maximum write count by half while only incurring about 1% reduction in rate of cache hits on average.

D. Performance Overhead

To evaluate the performance, we evaluated the total number of instructions and write instructions executed. Fig. 12 and 13 show the normalized instruction count and write instruction count, respectively, in which the last column gives the geometric means of these normalized values. Clearly, the *dynamic stack* incurs significant performance overhead, as about twenty to forty times more instructions are executed in some benchmarks. This overhead comes from the additional allocation and deallocation operations for dynamically allocating stack frames, and therefore is positively correlated to the number of function calls.

Compared to the *dynamic stack*, Loop2Recursion has substantially lower performance overhead, which arises from the invocation of recursive functions and the copy operations for pointees. When there are some variables of a large storage size to be copied, the overhead can be much higher. This is why the number of instructions increases greatly in $blowfish$.

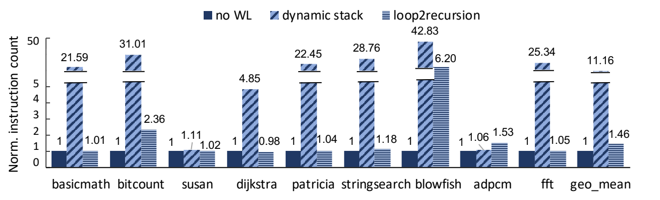


Fig. 12: Normalized instruction count.

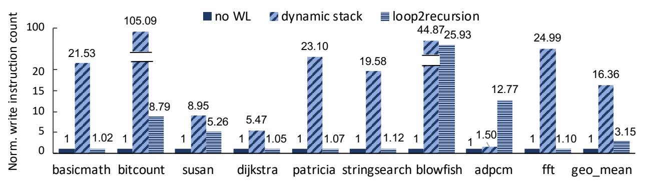


Fig. 13: Normalized write instruction count.

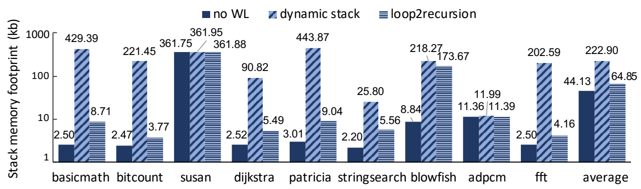


Fig. 14: Stack memory footprint.

This overhead can be alleviated by reducing the copy times for large-size pointees at the expense of wear leveling performance. Nevertheless, Loop2Recursion increases the instruction count and write instruction count by less than 5% and 10% for most benchmarks, which is negligible compared to the *dynamic stack*.

E. Stack Memory Overhead

In this subsection, we evaluate the stack memory footprint for the benchmarks under different schemes. The evaluation result is shown in Fig. 14. The *dynamic stack* significantly increases the stack memory usage, with an increase factor exceeding 100 for several benchmarks. The memory overhead of the *dynamic stack* can be lowered down by setting a larger allocation limit for UWLalloc, which however, may decrease the wear leveling performance. By contrast, the stack memory overhead caused by Loop2Recursion is much lower. On average, the stack memory consumption of Loop2Recursion is about 1.47 times as that of the baseline, which is somehow tolerable considering that the PCM memory provides much higher storage density.

F. Sensitivity Analysis to depth limitation

Lastly, we conduct a set of experiments to evaluate the sensitivity of our approach to the recursion depth limitation.

Fig. 15 compares the write count on the most-written address on a cacheless architecture under four different depth limitation configurations. The result suggests that the maximum write count decreases with the increasing depth limitation value. Therefore, Loop2Recursion can achieve better wear leveling when select a larger recursion depth limitation.

Fig. 16 compares the stack memory footprint under different depth limitation configurations. The stack memory usage increases generally with the increasing depth limitation value.

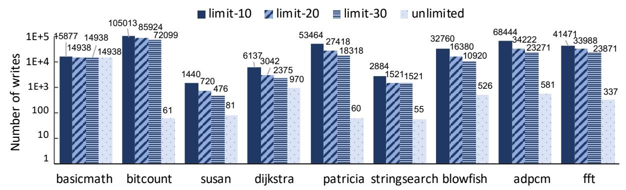


Fig. 15: Number of writes on the hottest memory address.

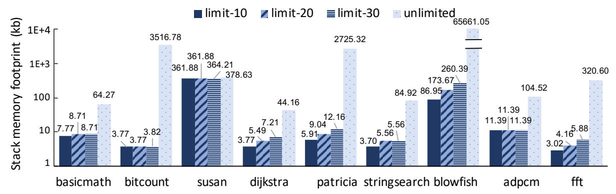


Fig. 16: Stack memory footprint under different depth limitation.

Also the stack memory consumption can be substantially high when the recursion depth is unlimited, which reflects that limiting the recursion depth effectively reduces the stack memory usage. What's more, by combining the results shown in Fig. 15 and Fig. 16, we can conclude that Loop2Recursion provides a flexible tradeoff between the wear leveling performance and stack memory usage.

VI. Conclusion

This paper presents Loop2Recursion, a compiler-assisted wear leveling technique for NVM. Loop2Recursion automatically transforms loops to recursive functions, eliminating the hot data inside stack frames and uniformly distributing stack frames among the stack area. In addition, this approach also enables a flexible tradeoff between wear leveling effectiveness and stack memory usage by limiting the depth of recursions. Experimental results indicate that Loop2Recursion can significantly reduce the maximum writes on the stack and improve the lifetime of NVM with negligible performance overhead.

VII. Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61502346, 61640221, 61972293, 61772377, 61672257, 91746206), and Science and Technology planning project of ShenZhen (JCYJ20170818112550194).

References

[1] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009.
[2] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable nvm," in *Proceedings of the International Symposium on Memory Systems*, 2019.
[3] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "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*, 2009.
[4] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Software wear management for persistent memories," in *FAST*, 2019.
[5] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing PCM main memory lifetime," in *DATE*, 2010.
[6] C. H. Chen, P. C. Hsiu, T. W. Kuo *et al.*, "Age-based PCM wear leveling with nearly zero search cost," in *DAC*, 2012.
[7] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia *et al.*, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.
[8] Q. Li, Y. He, Y. Chen, C. J. Xue *et al.*, "A wear-leveling-aware dynamic stack for PCM memory in embedded systems," in *DATE*, 2014.
[9] W. Li, Z. Shuai, C. J. Xue, M. Yuan, and Q. Li, "A wear leveling aware memory allocator for both stack and heap management in PCM-based main memory systems," in *DATE*, 2019.
[10] "Arm Cortex-M Series Processors," https://developer.arm.com/ip-products/processors/cortex-m.
[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.
[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization.*, 2001.
[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *TOPLAS*, 1991.
[14] C. K. Luk, R. Cohn *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM sigplan notices*, 2005.
[15] "ARM940T Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0144b/940T_TRM_S.pdf.