Effective Stack Wear Leveling for NVM

Jifeng Wu, Wei Li, Libing Wu, Mengting Yuan, Chun Jason Xue, *Senior Member, IEEE*, Jingling Xue, *Fellow, IEEE*, and Qingan Li

***Abstract*—With the rapid growth of data processed by computer systems, nonvolatile memory (NVM), represented by phase change memory (PCM), is regarded as a promising next-generation storage technology as it offers superior advantages over DRAM. However, PCM suffers from a severe write durability problem, leading to an extremely short lifespan under the uneven write patterns of real-world programs. We observe that loops are one of the primary causes of uneven writes on the stack. To alleviate this problem, we present Loop2Recursion, a compiler-assisted stack wear leveling technique that automatically transforms loops into recursive functions. In addition, we propose several optimizations to reduce the stack sizes and instruction counts of the generated recursive functions, two schemes to limit recursion depth, and selective loop transformation for cache-enabled architectures. Experimental results demonstrate that Loop2Recursion outperforms state-of-the-art methods by significantly improving stack wear leveling with a greatly reduced performance overhead.**

***Index Terms*—Compiler, loop, nonvolatile memory (NVM), recursion, wear leveling.**

I. Introduction

In the era of Big Data and AI, with the amount of data processed by computer systems increasing exponentially, existing DRAM-based main memory systems struggle to cope with the ever-growing demands for performance, energy efficiency [1], and scalability, especially for data-intensive, memory-heavy applications such as deep neural networks [2]. Emerging nonvolatile memory (NVM) technologies, such as phase change memory (PCM), feature byte addressability, high density, low power, storage-like persistence, and DRAM-compatible performance [3], making them a competitive candidate for the next generation of main memory.

Manuscript received 11 August 2022; revised 22 November 2022; accepted 6 January 2023. Date of publication 30 January 2023; date of current version 20 September 2023. This work was supported in part by the State Key Laboratory of Computer Architecture (ICT,CAS) under Grant CARCH A202112; in part by LIESMARS Special Research Funding; in part by the Key Research and Development Plan of Hubei Province under Grant 2020BAA021 and Grant 2021BAA025; in part by the Industry-University-Research Innovation Fund for Chinese Universities under Grant 2021FNA04004; in part by the National Key Research and Development Program of China under Grant 2022YFB4400704; and in part by the Australian Research Council under Grant DP210102409. This article was recommended by Associate Editor A. Gamatie. *(Corresponding author: Qingan Li.)*

Jifeng Wu, Libing Wu, Mengting Yuan, and Qingan Li are with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: jifengwu2k@gmail.com; wu@whu.edu.cn; ymt@whu.edu.cn; qingan@whu.edu.cn).

Wei Li and Jingling Xue are with the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia (e-mail: liwei@cse.unsw.edu.au; jingling@cse.unsw.edu.au).

Chun Jason Xue is with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: jasonxue@cityu.edu.hk).

Digital Object Identifier 10.1109/TCAD.2023.3240873

However, as a relatively young technology, PCM suffers from various problems, of which limited write endurance is particularly salient. For example, each PCM cell can only withstand $10^7 - 10^9$ writes, eight orders of magnitude smaller than DRAM [4]. Moreover, given that the writes of real-world programs tend to be concentrated in limited areas, causing intensive writes to wear out a small proportion of PCM cells prematurely, the lifespan of PCM may be just months [5].

To address this issue, researchers have proposed various wear leveling techniques that spread writes uniformly over the entire address space at multiple levels, including the hardware level [6], [7], [8], [9], [10], [11], [12], the operating system (OS) level [5], [13], [14], [15], [16], [17], and the program level. However, most hardware- and OS-level techniques neglect the uneven distribution of writes inside pages and require extra hardware to track page wear. In contrast, program-level methods aim at achieving wear balance by optimizing data allocation and access patterns of programs at compile- or run-time [18], [19], [20], [21]. Compared with hardware- and OS-level schemes, they provide a more economical and flexible wear leveling solution, as they do not require specialized hardware and OS support. They are especially suitable for embedded systems without a memory management unit (MMU) and caches, where hardware- and OS-level wear leveling techniques will become substantially more inefficient and impractical.

Existing program-level wear leveling solutions mainly address wear leveling for the heap [18], [19], [22] and stack memory segments of a program. As for the stack, Li et al. [20], [23] put forward a compiler-assisted dynamic stack that dynamically allocates stack frames in the manner of allocating heap objects. Although such a strategy spreads stack frames more evenly in memory, it overlooks unbalanced writes within stack frames. Many writes will converge to a few fixed locations when some variables are frequently updated within a stack frame. In addition, dynamically allocating stack frames comes with a significant performance overhead.

To address these two issues, we focus on optimizing loops to achieve improved stack wear leveling. This is based on the observations that loops usually introduce frequently updated hot variables, leading to uneven wear within stack frames, and that function calls within loops lead to a large number of stack frames being allocated on the same memory locations, which leads to an unbalanced distribution of stack frames. We propose Loop2Recursion, a compile-time wear leveling scheme that eliminates hot variables inside stack frames and mitigates the imbalance of stack frame distribution by automatically transforming loops into recursions. We have designed Loop2Recursion based on the intermediate representation (IR) of a popular compiler infrastructure [low level virtual machine (LLVM)]. This makes it applicable to a wide range of programming languages this infrastructure supports. In addition, as LLVM IR is hardware- and OS- independent, and Loop2Recursion can achieve wear leverling at a fine-granularity level, i.e., variable level, Loop2Recursion can be used together with other hardware- or OS-level wear leveling schemes for enhanced wear leveling. Since transforming loops into recursions may significantly increase stack memory usage, even leading to stack overflow, and incur a performance cost, we propose several optimizations to reduce the stack sizes and instruction counts of the generated recursive functions. In addition, we propose flexibly limiting the recursion depth to deal with deep recursions. Furthermore, we propose selective loop transformation for cache-enabled architectures, where we preserve loops not causing uneven stack writes.

Using programs from MiBench [24], we compare Loop2-Recursion with the state-of-the-art dynamic stack [20], [23], and comprehensively evaluate our stack size and instruction count optimizations, our two schemes to limit recursion depth, and Loop2Recursion on cache-enabled architecture. Experimental results demonstrate that Loop2Recursion outperforms the dynamic stack, as it exhibits better wear leveling effectiveness while incurring significantly lower overhead. In addition, our optimizations effectively reduce the stack sizes and instruction counts of the generated recursive functions, and limiting recursion depth by iteratively invoking a depth-limited recursive function works better to help achieve a satisfactory tradeoff between wear leveling effectiveness and memory overhead. Furthermore, with selective loop transformation, Loop2Recursion can also work well on a cache-enabled architecture with only a minor reduction in cache hits. In summary, the significant contributions of this article are as follows.

1) An empirical study showing that loops in programs significantly contribute to highly unbalanced stack writes.
2) The design of Loop2Recursion, an approach to achieve stack wear leveling by automatically transforming loops into recursions.
3) Optimizations to reduce the space and performance overhead of the recursive functions generated by Loop-2Recursion.
4) Two schemes are proposed to limit recursion depth and to provide a flexible tradeoff between wear leveling effectiveness and memory overhead.
5) Selective loop transformation to preserve loops not causing uneven writes for cache-enabled architectures.
6) A comprehensive experimental evaluation on the effectiveness and performance of Loop2Recursion, optimizations, two schemes to limit recursion depth, and Loop2Recursion on cache-enabled architecture.

The rest of this article is organized as follows. Section II reviews related work and Section III introduces the motivation of the Loop2recursion method. Section IV presents the Loop2Recursion method and our optimizations. We propose two schemes to limit recursion depth in Section V, and selective loop transformation in Section VI. Section VII conducts an extensive experimental evaluation of the methods above. Finally, Section VIII concludes the work.

II. Related Work

To address the write endurance issue of NVM, researchers have proposed various techniques at the hardware, OS, and program levels. These techniques can be divided into write reduction techniques that reduce the number of writes to NVM storage cells and wear leveling techniques that spread writes more uniformly.

A. Hardware-Level Techniques

1) Hardware-Level Write Reduction Techniques: To reduce the number of writes to NVM storage cells at the hardware level, we can utilize a hybrid main memory architecture consisting of both DRAM and NVM, where most writes are allocated to DRAM instead of NVM [6], [25], [26]. In a pure NVM main memory architecture, writes to NVM storage cells can be reduced by reducing writes at the bit level [7], [27], [28], [29], or compressing the data to be written [28], [29].

2) Hardware-Level Wear Leveling Techniques: At the hardware level, wear leveling can be achieved by shifting or swapping data between physical locations at different granularities, such as pages [8], blocks [7], and lines [6], [9]. Generally, finer granularities enable more precise wear leveling but at the cost of more significant performance and memory overhead. In addition, [10] also proposes maintaining a table mapping frequently updated hot data to NVM storage cells with few writes to achieve wear leveling, whose performance and memory overhead are further improved in [11] and [12].

B. OS- and Program-Level Techniques

1) OS- and Program-Level Write Reduction Techniques: As hardware-level techniques usually require additional, specialized hardware, many researchers have also proposed OS- and program-level write reduction techniques.

In a DRAM-NVM hybrid main memory architecture, it is possible to allocate frequently written data to DRAM using compile-time data allocation [30], [31], [32], [33] or runtime page scheduling [34]. As for a pure NVM main memory architecture, [35] proposes a compile-time write reduction approach based on recalculation. The basic idea is to compare the cost of writing data to NVM and accessing it later with the cost of recalculating the data when needed. A write is avoided when recalculation is more economical.

2) OS- and Program-Level Wear Leveling Techniques: At the OS level, wear leveling is usually accomplished through custom page scheduling algorithms, which involve keeping track of pages with write count [13], [14], [15], [16], [17], or predicting the wear of pages [5]. Similar page scheduling and page replacement algorithms have also been proposed for DRAM-NVM hybrid main memory architectures [36], [37].

In contrast, program-level techniques modify memory allocation and access patterns to distribute writes more evenly. For the heap memory segment of a program, wear-aware heap allocators [18], [19], [22] can prevent most heap objects from being allocated to the same addresses. As for wear leveling for the stack, inspired by the effectiveness of the heap allocators mentioned above, a compiler-assisted dynamic stack [20], [23], which allocates stack frames in the manner of allocating heap objects, has been proposed. Although this approach spreads stack frames more evenly, it neglects uneven writes within them and suffers from significant performance overhead. These two issues are addressed in Loop2Recursion.

```
#define NUM_NODES 100

int dijkstra(int chStart, int chEnd);

int main(int argc, char* argv[]) {
  // ...
  int i, j;
  for (i=0, j=NUM_NODES/2; i<NUM_NODES; ++i,++j) {
    j %= NUM_NODES;
    dijkstra(i, j);
  }
  // ...
}
```

Listing 1. Example of function calls within loops.

C. Transforming Loops to Recursion

The theoretical relation between iteration and recursion has been studied for decades from different perspectives [38], [39]. In general, compilers prefer iteration over recursion for performance, and there are lots of work on transforming recursive functions into loops. In contrast, transformations from loops to recursive functions are rare. Liu and Stoller [40] proposed an algorithm to transform iterative loops into tail-recursive functions for Java automatically. However, this algorithm does not work well in our work. First, this algorithm is tailored to the Java language. It can only handle well-defined loops in Java and cannot handle more general loops, like those implemented using the *goto* statement. Second, to handle premature exits from a loop (due to a *break*, *continue*, *return*, or *throw* statement), the generated recursive functions commonly return lots of information containing the specific statement executed in the recursive function, and parameters associated with that statement. Such an approach may incur significant overhead. In this work, we design Loop2Recursion to overcome these limitations and build a more general tool for transforming loops into recursions.

III. Motivation

In the stack segment of a program's memory space, memory is allocated in the form of stack frames storing functions' return addresses, arguments, local variables, etc., by adjusting the stack pointer register. Each stack frame corresponds to a function instance that has yet to terminate with a return. Whenever a function is invoked or returned, its frame is allocated or deallocated. The allocation and deallocation of stack frames happen on contiguous memory blocks in a last-in-first-out (LIFO) order. Such a mechanism leads to some memory areas associated with a relatively large number of stack frames while other areas are rarely used, leading to uneven writes. This is especially the case for function calls within loops, as shown in Listing 1, where the stack frames of calls to `dijkstra` are all allocated on the same location.

To overcome such a problem, Li et al. [20] proposed a compiler-assisted dynamic stack that allocates stack frames dynamically in a way analogous to heap allocation. Specifically, every time a function is called, an allocator based on the next-fit policy is immediately employed to obtain a free memory area for its frame. Li et al. [23] further improved such an approach by using a new wear-aware memory allocator with better support for wear leveling.

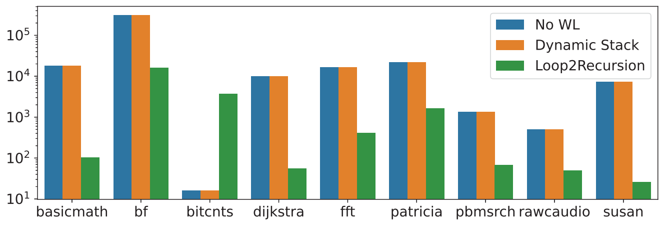


Fig. 1. Maximum number of writes on a memory address in stack frames.

```
#define NUM_NODES 100

int dijkstra(int chStart, int chEnd);

void main$0(int i, int j) {
  if (i < NUM_NODES) {
    j %= NUM_NODES;
    dijkstra(i, j);
    main$0(i + 1, j + 1);
  }
}

int main(int argc, char* argv[]) {
  // ...
  main$0(0, NUM_NODES / 2);
  // ...
}
```

Listing 2. Recursive version of Listing 1.

Although such an approach enables stack frames to be more uniformly distributed, mitigating the problem of uneven writes, some shortcomings still exist. On the one hand, the performance overhead of dynamically allocating and deallocating stack frames is much more significant than that of the traditional FIFO stack allocator. On the other hand, such an approach regards each stack frame as a whole and overlooks the imbalance of writes within a stack frame.

In fact, wear inside stack frames is far from equilibrium since frequently updated variables may produce intensive writes on their memory locations. For example, the loop in Listing 1 has a total of 100 iterations, and its loop variables `i` and `j` are anticipated to be updated 101 and 201 times, respectively. Under such circumstances, each update to these hot variables may cause a write to memory on a cacheless architecture, typically for embedded systems. Even when write-back caches are used, these writes may not be entirely avoided as cache conflicts will frequently occur in the worst-case scenario. As a result, the number of writes on their memory locations is much higher than other locations inside the stack frame. Fig. 1, which compares the maximum number of writes on a memory address in stack frames under no wear leveling, the dynamic stack, and our proposed Loop2Recursion, indicates that most programs suffer from uneven wear inside stack frames, with tens of thousands of writes concentrated on a few memory locations.

Based on the discussion above, loops are one of the primary causes of uneven wear on the stack, and it is feasible to mitigate this issue by transforming loops into recursive functions.

For example, Listing 2 is a recursive version of the example of function calls within loops in Listing 1. The original loop within `main` is replaced with a recursive function, `main$0`, in which the original frequently updated loop variables become its parameters, and the original loop conditions are used as its recursive conditions.

In Listing 2, each time `main$0` is called, its stack frame is allocated on the top of the stack, and the stack frames of `dijkstra`, which were originally collocated, are allocated on different memory locations, enabling a uniform stack frame distribution. Furthermore, local variables in stack frames are only written a limited number of times, allowing wear leveling inside stack frames, as demonstrated in the Loop2-Recursion data in Fig. 1. Thus, the recursion could provide better wear leveling than the original loop.

IV. Loop2Recursion

In this section, we elaborate on our wear-leveling technique Loop2Recursion, which automatically transforms loops into equivalent recursions.

A. Design Considerations

We implement Loop2Recursion as a pass run by LLVM's code optimization tool, *opt*. This greatly simplifies the process of loop transformation and exhibits many advantages.

1) As there are various front-ends to compile different languages (such as C, C++, Rust, and Swift) into LLVM IR, Loop2Recursion can be applied to many languages.
2) Compared with assembly language, LLVM IR is platform-neutral. Thus, Loop2Recursion is applicable to a myriad of target platforms.
3) By running relevant passes, *opt* can detect loops in LLVM IR and convert them into canonical forms. As a result, we can focus on a unified, standardized form of loops without having to handle the burden of different types of loops (for, while, do-while, etc.) and control statements (break, continue, return, goto, throw, etc.).

Our work mainly targets embedded systems, where a full cache hierarchy might not be preferable [41]. For systems with caches, we propose a selective loop transformation method as depicted in Section VI.

B. Process of Transforming Loop

The process of transforming a loop can be divided into four steps: 1) converting the loop into a canonical form; 2) determining the recursive function's parameters and return values; 3) creating the recursive function; and 4) substituting the loop with the recursive function. Below we describe each of these steps in detail.

1) Converting the Loop Into Canonical Form: The first step of transforming a loop is converting the loop into a canonical form known as the LoopSimplify form. The LoopSimplify form adds the following constraints to loops.

1) Basic blocks within the loop induce a maximal strongly connected subgraph.
2) There is only one basic block outside the loop, *Preheader*, with an out edge pointing into the loop. The target of the out edge is known as *Header*.
3) There is only one basic block within the loop, *Latch*, with an out edge pointing toward *Header*. *Latch* may also have other out edges pointing toward other basic blocks within the loop.

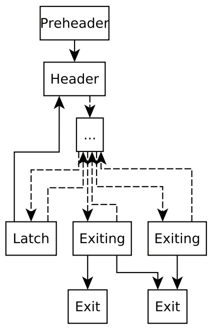


Fig. 2. Loop in the LoopSimplify form.

4) There is at least one *Exiting* basic block within the loop with out edges pointing toward basic blocks outside the loop. Each target of such an out edge is known as *Exit*. *Exiting* blocks may also have other out edges pointing toward other basic blocks within the loop, but *Exit* blocks only have *Exiting* blocks as predecessors.

Fig. 2 depicts a loop in the LoopSimplify form.

The LoopSimplify form greatly simplifies the analysis and transformation of loops. By converting loops into the LoopSimplify form, we can take the same approach to handle different loops present in source code. This conversion is already supported by the LLVM *opt* tool.

2) Determining the Recursive Function's Parameters and Return Values: A loop generally uses some variables defined outside the loop. To make these variables accessible inside the generated recursive function, we can pass them as arguments to the recursive function.

These variables can be divided into two classes: 1) variant variables and 2) invariant variables.

Variant variables are variables updated in each loop iteration, such as loop variables. As entering a loop from outside of the loop implies arriving at the loop's *Header* basic block, and entering the next iteration of the loop involves branching to *Header* from *Latch*, we can fetch variant variables from the Φ functions in the loop's *Header*.

In contrast, invariant variables remain unchanged for the whole loop. To fetch them, we can iterate all instructions in the loop (aside from the Φ functions in the loop's *Header*) and check their operands. If an operand is defined outside the loop and is not a global variable, it is an invariant variable.

In addition to variables defined outside the loop and used within the loop, there may also be variables defined within the loop and used outside the loop. Having transformed the loop into a recursive function, to allow these variables to be used after calling the recursive function, we can return them from the recursive function. Specifically, we tailor a structure known as the Return Value Structure, which packs all return values. An instance of the Return Value Structure is created in the last call to the recursive function before being returned from the recursive function.

3) Creating the Recursive Function: After acquiring information about the loop, we can create the recursive function.

In the original loop structure, a branch from *Latch* to *Header* allows us to enter the next iteration of the loop, while a branch from *Exiting* to *Exit* enables us to exit the loop. Thus, in the

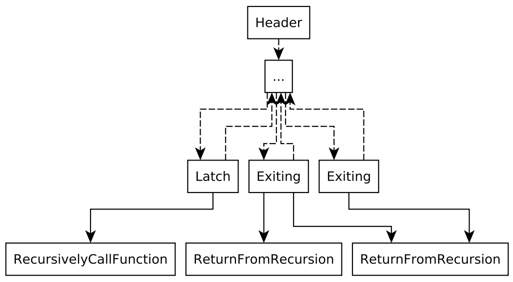


Fig. 3. Structure of the recursive function.

generated recursive function, it is viable to recursively call the function after entering the *Latch* basic block and begin returning from recursion after entering the *Exiting* basic block. To implement this, we can create an empty recursive function, move all the basic blocks within the loop to the recursive function, and add the following modifications. The structure of the recursive function is depicted in Fig. 3.

1) Insert a *RecursivelyCallFunction* basic block that recursively calls the function and returns. Modify the *Latch* to *Header* edge to point to *RecursivelyCallFunction*.
2) Insert a series of *ReturnFromRecursion* basic blocks that correspond with the original *Exit* blocks. Return from the recursive function in each *ReturnFromRecursion* block. Modify each out edge from *Exiting* to *Exit* to point to the corresponding *ReturnFromRecursion*.

When exiting the original loop, different *Exiting* basic blocks may lead to different *Exit* blocks. However, in the generated recursive function, no matter which *ReturnFromRecursion* block we enter, we will eventually reach the same block after returning from recursion—the block that initially called the recursive function. Thus, should there be multiple *Exit* blocks, we can label each *Exit* block and implement the ID of the target *Exit* block as an additional return value of the recursive function. Under such circumstances, we will branch to the relevant *Exit* block based on such a return value after returning from recursion.

4) Substituting the Loop With the Recursive Function: Having created the recursive function, we shall substitute the loop with the recursive function. The modifications are as follows.

1) Insert a basic block, *CallRecursiveFunction*, into the function containing the original loop.
2) Modify the out edge from *Preheader* to *Header* to point to *CallRecursiveFunction*.
3) Call the recursive function in *CallRecursiveFunction*.

Afterward, should there be multiple *Exit* blocks, then:

1) insert a basic block, *BranchToExit*, and branch to it after calling the recursive function in *CallRecursiveFunction*;
2) branch to the appropriate *Exit* block according to the returned target *Exit* block ID in *BranchToExit*.

The process of calling the recursive function in the case of multiple *Exit* blocks is depicted in Fig. 4. However, should there be only one *Exit* block, we shall directly branch to that block from *CallRecursiveFunction* instead.

Finally, we shall replace each of the Φ functions in the *Exit* blocks with the appropriate return values of the recursive function.

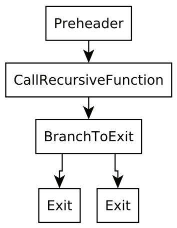


Fig. 4. Process of calling the recursive function in the case of multiple *Exit* blocks.

5) Handling Nested Loops: It is often the case that there are loops nested within other loops. To handle this situation, whenever a recursive function is generated, we detect whether there are loops within the function body. Then, for each detected loop, we transform it into a new recursive function. This process is repeated for each generated recursive function until no more nested loops are detected.

Following such a procedure to handle nested loops, we adopt the following scheme to name our generated recursive functions. If there are *m* loops within a function `f`, the recursive function generated from the *i*th loop will be named `f$i`. Thus, if there are *n* loops nested within the *i*th loop in `f`, the recursive function generated from the *j*th nested loop will be named `f$i$j`, as it is the *j*th loop in `f$i`.

C. Optimizations to Loop2Recursion

Although transforming loops into recursions improves stack wear leveling, it may also significantly increase stack memory usage, even leading to stack overflow, and incur a performance cost. As a result, we propose several optimizations to reduce the stack sizes and instruction counts of the generated recursive functions.

1) Optimizing Passing Invariant Variables: As invariant variables remain unchanged throughout the loop, the same values are passed as arguments each time the recursive function is called. In addition, a recursive function generated from an inner loop may share many invariant variables with a recursive function generated from an outer loop. This opens up an opportunity for optimization.

Instead of passing invariant variables as arguments to the generated recursive functions, we can store these invariant variables as global variables in the data segment of the memory before calling the recursive function, and access them through such global variables in the recursive function.

We give an example of this optimization below. Listing 3 presents two recursive functions, fft_float$2 and fft_float$2$0, generated from a loop hierarchy in the *fft* benchmark before optimizing invariant variables. Listing 4 presents the same two recursive functions after optimizing invariant variables. By comparing Listings 3 and 4, it is evident that invariant variables account for a large portion of the generated recursive functions' parameters before optimization. In addition, three invariant variables of fft_float$2 generated from an outer loop, `i32%arg2, float* %arg3, float* %arg4`, are also passed on as invariant variables to fft_float$2$0 generated from an inner loop.

By converting invariant variables from function parameters to global variables, the number of parameters of the generated recursive functions is reduced, and invariant variables can be

```
define void @"fft_float$2"(
  i32 %arg,
  i32 %arg1,
  i32 %arg2,
  float* %arg3,
  float* %arg4,
  double %arg5,
  i1 %arg6
) {
  ; omitted code
"fft_float$2$0_call_recursive_function":
  call void @"fft_float$2$0"(
    i32 0,
    i32 %arg,
    i32 %arg2,
    i32 %arg1,
    float* %arg3,
    float* %arg4,
    double %call22,
    i1 %cmp37236,
    double %call20,
    double %mul27,
    i32 %arg,
    double %cos,
    double %call24
  )
  br label %for.end110.loopexit
  ; omitted code
}


define void @"fft_float$2$0"(
  i32 %arg,
  i32 %arg1,
  i32 %arg2,
  i32 %arg3,
  float* %arg4,
  float* %arg5,
  double %arg6,
  i1 %arg7,
  double %arg8,
  double %arg9,
  i32 %arg10,
  double %arg11,
  double %arg12
) {
  ; omitted code
}
```

Listing 3. Two recursive functions generated from a loop hierarchy in the fft benchmark before optimizing invariant variables.

```
define void @"fft_float$2"(i32 %arg, i32 %arg1) {
  ; omitted code
"fft_float$2$0_call_recursive_function":
  call void @"fft_float$2$0"(i32 0, i32 %arg)
  br label %for.end110.loopexit
  ; omitted code
}

define void @"fft_float$2$0"(i32 %arg, i32 %arg1) {
  ; omitted code
}
```

Listing 4. Two recursive functions in Listing 3 after optimizing invariant variables.

shared between recursive functions generated from outer loops and those generated from inner loops. This reduces the generated recursive functions' stack sizes and eliminates instructions passing the invariant variables as arguments.

2) Optimizing Passing Return Values: When using a Return Value Structure, the created instance behaves like an invariant variable, as it is modified only once in the last call to the recursive function before being returned from each recursion layer. Accordingly, we can allocate the instance in the data segment of the memory as well. This reduces stack usage and eliminates instructions returning the instance.

V. Limiting Recursion Depth

Although the optimizations mentioned above can reduce the stack sizes and instruction counts of the generated recursive functions, deep recursions left unchecked still have the potential to consume excessive stack space, resulting in stack overflow exceptions. To solve this problem, it is viable to limit recursion depth. We can accomplish this in two ways: 1) iteratively invoking a depth-limited recursive function or 2) invoking a recursive function containing loop iterations.

A. Iteratively Invoking Depth-Limited Recursive Function

To limit recursion depth to k, we can transform a large loop of n iterations into a smaller loop of n/k iterations, with each iteration calling a recursive function with a depth limit of k.

Such a scheme requires us to track recursion depth. This can be implemented using an additional parameter serving as a recursion depth counter. When the recursive function is called from outside the recursive function, 0 is passed. The caller's parameter is incremented with each recursive call and passed to the callee. Should the incremented parameter equals the recursion depth limit, we will return from recursion instead of continuing recursion before invoking the recursive function from the outside again.

Such a scheme also requires knowing the appropriate values of variant variables to pass to the recursive function each time and when to stop. Thus, we can tailor a Recursion State Structure that encompasses such information. Analogous to the Return Value Structure, we can also allocate an instance of the Recursion State Structure in the data segment of the memory. Whenever we return from recursion due to the recursion depth limit, the subsequent values of the variant parameters and the value *false* (meaning the recursive function has to be invoked again from the outside) are written to the Recursion State Structure instance. In contrast, the value *true* is written in the last call to the recursive function. In iteratively invoking the recursive function, such a boolean value is checked each time the recursive function returns. If it is *false*, the next values of the variant variables are fetched from the Recursion State Structure instance, and the recursive function is invoked again. Elsewise, we have finished.

The following adjustments are made to the recursive function to implement such a scheme. The structure of the depth-limited recursive function is depicted in Fig. 5.

1) Insert two basic blocks, *IncrementDepthCounter* and *SaveRecursionState*, into the recursive function.
2) Modify the out edge from *Latch* to *RecursivelyCallFunction* to point to *IncrementDepthCounter*.
3) Increment the depth counter parameter and compare the incremented depth counter with the recursion depth limit in *IncrementDepthCounter*. If the incremented depth counter is less than the recursion depth limit, branch to *RecursivelyCallFunction*. Otherwise, branch to *SaveRecursionState*.

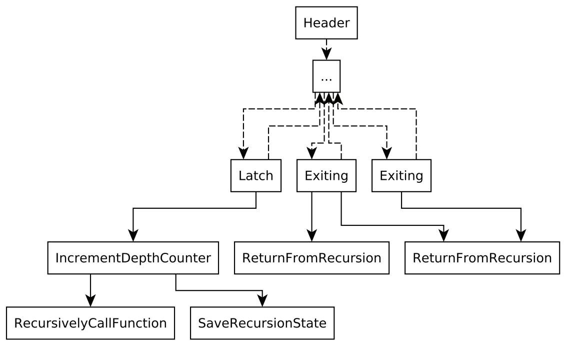


Fig. 5. Structure of the depth-limited recursive function.

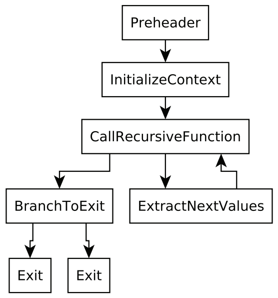


Fig. 6. Process of invoking the recursive function iteratively.

4) Pass the incremented depth counter in the recursive function call in *RecursivelyCallFunction*.
5) Write the following values of the variant parameters and the value *false* to the Recursion State Structure instance, before returning from the recursive function in *SaveRecursionState*.
6) Write the value *true* to the Recursion State Structure instance before returning from the recursive function in each *ReturnFromRecursion* block.

In addition, the function containing the original loop is also modified to support invoking the recursive function iteratively. The modifications are as follows, and the process of invoking the recursive function iteratively is depicted in Fig. 6.

1) Insert a basic block, *ExtractNextValues*, into the function containing the original loop.
2) Fetch the next values of the variant parameters from the Recursion State Structure instance and branch to *CallRecursiveFunction* in *ExtractNextValues*.
3) Modify *CallRecursiveFunction*.
    a) Add Φ functions to the beginning of *CallRecursiveFunction* to select the values of the variant parameters. If arriving at *CallRecursiveFunction* from *ExtractNextValues*, select the following values extracted from the Recursion State Structure instance. Elsewise, select the original values of the variant parameters.
    b) After calling the recursive function, extract the boolean value indicating whether we have finished iteratively invoking the recursive function from the Recursion State Structure instance. If it is *false*, branch to *ExtractNextValues*. Elsewise, branch to *CallRecursiveFunction*'s original successor.

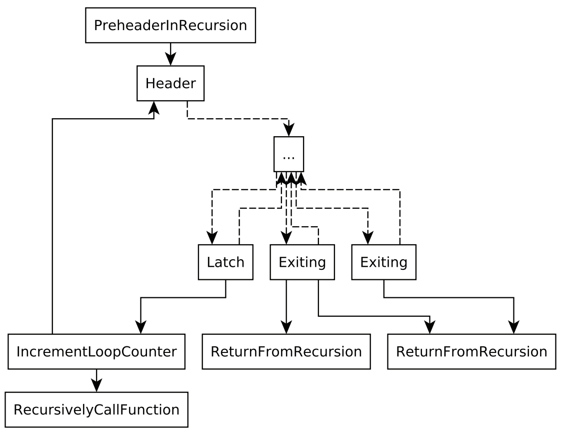


Fig. 7. Structure of the recursive function containing loop iterations.

B. Invoking Recursive Function Containing Loop Iterations

Alternatively, we can include n/k iterations of the original loop within the recursive function to limit recursion depth to a threshold value k. This is implemented as follows.

1) Insert a basic block, *PreheaderInRecursion*, at the beginning of the generated recursive function.
2) Branch to *Header* in *PreheaderInRecursion*.
3) Add a Φ function at the beginning of *Header* that serves as a loop iteration counter. Select the value 0 if the control flow is from *PreheaderInRecursion*.
4) Insert a basic block, *IncrementLoopCounter*, into the generated recursive function.
5) Modify the out edge from *Latch* to *RecursivelyCallFunction* to point to *IncrementLoopCounter*.
6) Increment the current loop iteration counter in *IncrementLoopCounter*. If the incremented loop iteration counter is less than the loop iteration limit, branch to *Header*. Otherwise, branch to *RecursivelyCallFunction*.
7) Modify the Φ function serving as the loop iteration counter at the beginning of *Header*. Select the incremented loop counter if the control flow is from *IncrementLoopCounter*.

The structure of the recursive function containing loop iterations is depicted in Fig. 7.

VI. Selective Loop Transformation for Platforms With Caches

As stated in Section IV-A, Loop2Recursion mainly targets embedded systems without a cache hierarchy due to caches often being undesirable in an embedded environment. However, we also extend Loop2Recursion to support platforms with caches by performing a selective transformation of loops. Specifically, the following features identify the loops that should be ignored during loop transformation.

1) Nested loops, which usually perform computational-intensive calculations with high memory locality.
2) Loops without function calls and making few writes to the stack.

It is easy to collect the first feature by visiting the loop structures. For the second feature, dynamic profiling can be employed to check whether a loop incurs hot writes to the stack. Alternatively, a precompilation can be used to generate assembly instructions first, and then all instructions within a loop are visited to check whether they iteratively write to the stack with the same offset.

VII. Experimental Study

We have implemented Loop2Recursion as an LLVM Pass targeting LLVM 13.0.1. We conduct the following experimental studies.

1) Comparing no wear leveling, the state-of-the-art dynamic stack [20], [23], and Loop2Recursion under typical parameters.
2) Evaluating our optimizations to Loop2Recursion.
3) Evaluating our two measures to limit recursion depth, which includes assessing both iteratively invoking a depth-limited recursive function and invoking a recursive function containing loop iterations, before determining which approach is better in terms of stack memory usage and wear leveling effectiveness.
4) Evaluating the selective loop transformation method on a cache-enabled architecture.

The aspects assessed and the corresponding indicators used in the experiment are as follows.

1) Wear leveling effectiveness. Due to the assumption that NVM's lifespan is determined by the worst wear degree, we adapt the number of writes on the hottest stack address as a metric of the overall wear leveling effectiveness.
2) Wear leveling overhead, measured by maximum stack usage and instruction count.

We perform our evaluations using programs from MiBench [24]. First, the source code for each program is compiled with "clang -O2" to LLVM IR. Then, the executables are generated by compiling the LLVM IR for each program, either directly (in the case of no wear leveling) or after running the appropriate passes with *opt* to transform the LLVM IR (as for dynamic stack and Loop2Recursion).

To acquire our data, we have implemented two profiling tools based on Intel Pin [42], one that generates logs recording events such as function calls, function returns, memory reads, and memory writes during the execution of a program, and the other that counts a program's executed instructions. Other measures, including stack sizes and the number of writes on each stack memory address, can be derived by analyzing the logs generated from the first Pin tool.

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). For the cache-enabled architecture, we implement a cache simulator simulating a 4KB, 64-way associative data cache, which references the parameters of ARM940T.[1] All source code, including the Loop2Recursion LLVM Pass, the Intel Pin-based profiling tools, and data analysis Python scripts, are available on GitHub.[2]

A. Comparing No Wear Leveling, Dynamic Stack, and Loop2Recursion

We first compare our Loop2Recursion under typical settings with no wear leveling and the state-of-the-art dynamic stack [20], [23]. For the dynamic stack, the wear-aware memory allocator UWLalloc [23] is employed, limiting the number of stack frames allocated on each memory address. We set the allocation limit to 300, as proposed in the original work. For Loop2Recursion, we iteratively invoke a depth-limited recursive function to limit recursion depth and set the recursion depth limit to 64.

TABLE I
Writes on the Hottest Stack Address

Benchmark	No WL	DynamicStack Change	Loop2Recursion Change
basicmath	3.73e+05	-79.44%	-86.88%
bf	1.28e+06	-24.41%	-87.81%
bitcnts	6.05e+06	-82.64%	-98.68%
crc_32	2.74e+06	-0.02%	-50.01%
dijkstra	4.51e+06	0.01%	-98.90%
fft	3.04e+05	-81.05%	-89.81%
patricia	3.86e+05	-74.73%	-89.13%
pbmsrch	1.46e+03	-34.29%	-97.12%
qsort	5.82e+04	-65.63%	-53.55%
rawcaudio	1.89e+06	-66.21%	-98.85%
rawdaudio	2.05e+06	-75.53%	-98.87%
sha	2.05e+05	-92.86%	-86.30%
susan	3.47e+06	0.00%	-99.48%

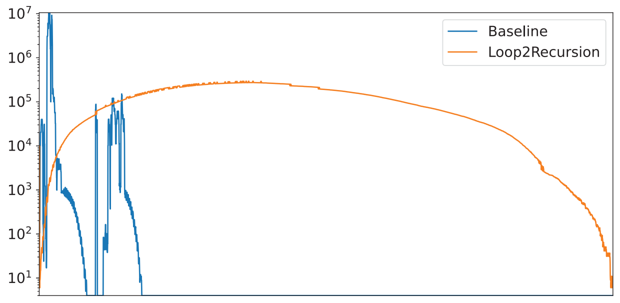


Fig. 8. Writes on stack addresses for Dijkstra. The *x*-axis is the stack address space, while the *y*-axis is the number of writes.

1) Wear Leveling Effectiveness: Table I compares writes on the hottest stack address of no wear leveling, dynamic stack, and Loop2Recursion. The dynamic stack and Loop2Recursion significantly decrease the write count on the hottest stack address compared to no wear leveling. However, Loop2Recursion achieves an overall reduction much higher than the dynamic stack. Although the dynamic stack slightly outperforms Loop2Recursion in some benchmarks (*qsort* and *sha*), Loop2Recursion provides uncompromising and more uniform reduction in writes on the hottest stack address for all benchmarks. As a result, Loop2Recursion exhibits higher wear leveling effectiveness.

The wear leveling effectiveness can be demonstrated another way, i.e., the distribution of writes over the stack memory space. For example, Fig. 8 shows the distribution of writes over the stack area for both no wear leveling and Loop2Recursion for the *dijkstra* benchmark. It illustrates that the Loop2Recursion could achieve a much more even distribution of writes over the whole stack memory space and reduce writes on the hottest stack address.

2) Maximum Stack Usage: Table II compares the maximum stack usages of no wear leveling, dynamic stack, and Loop2Recursion. The dynamic stack significantly increases maximum stack usage over no wear leveling. Although it is

[1]https://developer.arm.com/documentation/ddi0144/b/
[2]https://github.com/abbaswu/loop2recursion

TABLE II
MAXIMUM STACK USAGES

Benchmark	No WL	DynamicStack Change	Loop2Recursion Change
basicmath	2.78e+03	14961.49%	68.39%
bf	9.20e+03	2443.39%	89.57%
bitcnts	2.75e+03	8139.83%	148.26%
crc_32	2.16e+03	13541.85%	8.89%
dijkstra	2.58e+03	3209.63%	475.16%
fft	2.82e+03	5842.61%	277.84%
patricia	3.17e+03	12592.42%	127.27%
pbmsrch	2.37e+03	56.08%	511.49%
qsort	7.68e+06	1.85%	0.02%
rawcaudio	1.16e+04	4.34%	23.72%
rawdaudio	1.16e+04	4.34%	6.21%
sha	1.01e+04	212.10%	51.42%
susan	3.72e+05	0.05%	2.95%

TABLE III
INSTRUCTION COUNTS

Benchmark	No WL	DynamicStack Change	Loop2Recursion Change
basicmath	4.94e+07	2618.82%	2.05%
bf	5.19e+07	6086.06%	22.92%
bitcnts	9.13e+07	3640.74%	48.78%
crc_32	4.94e+07	8780.95%	25.43%
dijkstra	1.03e+08	212.18%	40.35%
fft	3.02e+07	3347.48%	2.83%
patricia	7.26e+07	2250.64%	1.23%
pbmsrch	1.57e+05	10132.18%	13.17%
qsort	1.46e+07	5951.21%	3.17%
rawcaudio	1.03e+08	16.51%	3.67%
rawdaudio	7.30e+07	23.38%	18.31%
sha	2.14e+07	255.43%	27.60%
susan	7.26e+07	21.93%	15.13%

TABLE IV
MAXIMUM STACK USAGES

Benchmark	Unoptimized	Optimized Change
basicmath	4.98e+04	-32.16%
bf	1.39e+06	-81.20%
bitcnts	7.20e+06	-33.33%
crc_32	8.76e+07	-50.00%
dijkstra	1.02e+05	-46.63%
fft	4.59e+05	-57.09%
patricia	1.22e+06	-57.00%
pbmsrch	1.34e+04	-19.02%
qsort	8.32e+06	-3.85%
rawcaudio	2.30e+05	-18.71%
rawdaudio	1.82e+05	-14.85%
sha	9.04e+04	-80.29%
susan	4.19e+05	-9.08%

TABLE V
INSTRUCTION COUNTS

Benchmark	Unoptimized	Optimized Change
basicmath	5.02e+07	-0.92%
bf	7.95e+07	-29.35%
bitcnts	1.16e+08	-10.28%
crc_32	5.90e+07	-13.93%
dijkstra	1.34e+08	-13.00%
fft	3.12e+07	-2.05%
patricia	7.31e+07	-0.95%
pbmsrch	1.68e+05	1.15%
qsort	1.51e+07	-3.12%
rawcaudio	1.03e+08	-2.01%
rawdaudio	8.26e+07	-3.33%
sha	2.49e+07	-9.30%
susan	9.38e+07	-26.73%

possible to reduce the stack usage of the dynamic stack by setting a larger allocation limit for UWLalloc, this may negatively affect wear leveling effectiveness. In comparison, the increase in maximum stack usage under Loop2Recursion is much lower and more uniform, which translates to better overall maximum stack usage.

3) Instruction Count: Table III compares the instruction counts of no wear leveling, dynamic stack, and Loop2-Recursion. The dynamic stack incurs significant performance overhead, as about twenty times more instructions are executed on average compared to no wear leveling. This overhead comes from the additional allocation and deallocation operations for dynamically allocating stack frames and positively correlates to the number of function calls. Compared to the dynamic stack, Loop2Recursion has substantially lower performance overhead, with each benchmark executing far fewer instructions than under the dynamic stack.

B. Evaluation of Our Optimizations to Loop2Recursion

As stated in Section IV-C, this article proposed to optimize the overhead of recursive function calling by improving passing invariant variables and return values. Here, the effectiveness of these optimizations is evaluated.

1) Maximum Stack Usage: Table IV compares the maximum stack usages of unoptimized and optimized Loop2-Recursion. Apart from *qsort*, in which the maximum stack usage remains relatively unchanged, our optimizations have significantly reduced the maximum stack usage of most benchmarks.

2) Instruction Count: Table V compares the instruction counts of unoptimized and optimized Loop2Recursion. Similar to the maximum stack usage trend, most benchmarks' instruction counts have also been reduced. A notable exception is *pbmsrch*, where there is a slight (1.15%) increase in instruction count. Such an increase is due to a generated recursive function frequently accessing an invariant parameter, which was formerly placed in a register but is now in the memory due to optimization, which incurs an extra instruction to pass the parameters. However, its effect on the total instruction counts of the benchmarks is quite limited, as evidenced by the reduction in instruction count for all other benchmarks.

3) Wear Leveling Effectiveness: Table VI compares the number of writes on the hottest stack address of unoptimized and optimized Loop2Recursion. The number of writes on the hottest stack address is determined by both the distribution of stack frames and the distribution of writes within stack frames. The former can be measured by the maximum number of stack frames on a stack address, while the maximum number of writes within each stack frame indicates the latter. Table VII compares the maximum number of stack frames on a stack address of unoptimized and optimized Loop2Recursion, while Table VIII compares the maximum number of writes within each stack frame.

It is clear that our optimizations have a negligible effect on the maximum number of writes within each stack frame as illustrated in Table VIII, while causing a significant increase in the maximum number of stack frames on a stack address as shown in Table VII. Thus, the increase in the number of

TABLE VI
WRITES ON THE HOTTEST STACK ADDRESS

Benchmark	Unoptimized	Optimized Change
basicmath	3.60e+04	55.55%
bf	1.56e+05	0.00%
bitcnts	8.60e+01	37.21%
crc_32	1.37e+06	0.00%
dijkstra	1.00e+04	6.77%
fft	5.78e+02	34.26%
patricia	2.18e+04	0.00%
pbmsrch	5.20e+01	17.31%
qsort	2.30e+04	0.06%
rawcaudio	1.25e+03	13.39%
rawdaudio	1.37e+03	6.64%
sha	1.46e+04	0.00%
susan	1.18e+04	50.44%

TABLE VII
MAXIMUM NUMBER OF STACK FRAMES ON A STACK ADDRESS

Benchmark	Unoptimized	Optimized Change
basicmath	42182.00	23.96%
bf	138.00	234.06%
bitcnts	64.00	37.50%
crc_32	17.00	5.88%
dijkstra	7516.00	39.61%
fft	334.00	30.54%
patricia	96.00	91.67%
pbmsrch	48.00	2.08%
qsort	20014.00	0.07%
rawcaudio	742.00	2.16%
rawdaudio	825.00	-6.42%
sha	776.00	794.46%
susan	7125.00	61.74%

TABLE VIII
MAXIMUM NUMBER OF WRITES WITHIN EACH STACK FRAME

Benchmark	Unoptimized	Optimized Change
basicmath	3.60e+04	0.00%
bf	1.56e+05	0.00%
bitcnts	2.20e+01	-4.55%
crc_32	1.37e+06	0.00%
dijkstra	1.00e+04	0.00%
fft	6.22e+02	0.00%
patricia	2.18e+04	0.00%
pbmsrch	1.40e+01	0.00%
qsort	1.00e+04	0.00%
rawcaudio	1.00e+01	0.00%
rawdaudio	1.00e+01	0.00%
sha	1.46e+04	0.00%
susan	1.00e+01	0.00%

writes on the hottest stack address can be mainly attributed to a denser distribution of stack frames.

It is worth mentioning that the denser distribution of stack frames is mainly due to the aggressive reduction of maximum stack usage as illustrated in Table IV. Therefore, we further compare the unoptimized and optimized Loop2Recursion on wear leveling under similar maximum stack usage.

By iteratively invoking a depth-limited recursive function in both unoptimized and optimized Loop2Recursion, we can profile all benchmarks to acquire the average maximum stack usages and writes on the hottest stack address under different recursion depth limits, as shown in Fig. 9.

From Fig. 9, it is clear that with average maximum stack usage controlled, our optimized Loop2Recursion comes with significantly reduced average writes on the hottest stack address. Thus, when limiting recursion depth, our optimized Loop2Recursion has a significant edge over unoptimized Loop2Recursion in terms of wear leveling effectiveness.

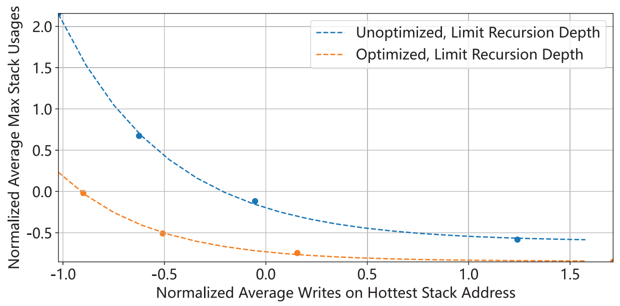


Fig. 9. Average maximum stack usages and writes on the hottest stack address under different recursion depth limits.

TABLE IX
WRITES ON THE HOTTEST STACK ADDRESS

Benchmark	Unlimited	8 Change	16 Change	32 Change	64 Change	128 Change
basicmath	5.60e+04	12.16%	-9.82%	-14.63%	-12.60%	-14.70%
bf	1.56e+05	30.64%	0.00%	0.00%	0.00 %	0.00%
bitcnts	1.18e+02	317711.86%	150893.22%	107110.17%	67490.68%	33910.17%
crc_32	1.37e+06	0.00%	0.00%	0.00%	0.00%	0.00%
dijkstra	1.07e+04	3122.34%	1532.07%	736.83%	363.52%	160.42%
fft	7.76e+02	16612.50%	11421.01%	7659.79%	3895.49%	2051.42%
patricia	2.18e+04	530.03%	327.99%	159.89%	92.65%	-0.09%
pbmsrch	6.10e+01	345.90%	157.38%	32.79%	-31.15%	-31.15%
qsort	2.30e+04	57.68%	46.83%	16.94%	17.55%	8.70%
rawcaudio	1.41e+03	12418.74%	6058.27%	2978.50%	1436.92%	669.65%
rawdaudio	1.46e+03	12347.50%	6173.51%	3086.58%	1481.25%	688.90%
sha	1.46e+04	835.55%	360.47%	201.55%	91.81%	0.14%
susan	1.78e+04	683.21%	285.23%	108.82%	0.30%	-12.56%

C. *Evaluation of Two Measures to Limit Recursion Depth*

The following section will evaluate our two measures to limit recursion depth. Both measures are implemented in our optimized Loop2Recursion.

1) Results of Iteratively Invoking Depth-Limited Recursive Function:

a) Wear leveling effectiveness: Table IX compares the writes on the hottest stack address under different recursion depth limits. There is a clear trend that writes on the hottest stack addresses decrease rapidly as the stack depth limit increases. The reason is that, with a deeper recursive depth, the frames will be distributed over a larger memory space, and thus the hottest addresses may be cooled. Thus, avoiding a shallow recursion depth limit is essential to avoid negatively impacting the effectiveness of wear leveling.

b) Maximum stack usage: Table X compares the maximum stack usages under different recursion depth limits. Limiting recursion depth is very effective at further reducing stack memory usage. Furthermore, when the recursion depth limit is not too large (below 64 in this experiment), although increasing the recursion depth limit does affect maximum stack usage, the increase is slight both overall and for most benchmarks, as demonstrated in the slowly changing increases in stack size relative to unlimited recursion depth. However, one should avoid setting a recursion depth limit that is too high

TABLE X
MAXIMUM STACK USAGES

Benchmark	Unlimited	8 Change	16 Change	32 Change	64 Change	128 Change
basicmath	3.38e+04	-88.64%	-86.88%	-86.13%	-86.13%	-80.64%
bf	2.62e+05	-96.08%	-95.54%	-94.47%	-93.34%	-91.78%
bitcnts	4.80e+06	-99.91%	-99.91%	-99.90%	-99.86%	-99.77%
crc_32	4.38e+07	-99.99%	-99.99%	-99.99%	-99.99%	-99.99%
dijkstra	5.47e+04	-95.08%	-92.33%	-85.54%	-72.90%	-53.02%
fft	1.97e+05	-98.52%	-97.88%	-96.37%	-94.60%	-91.66%
patricia	5.26e+05	-99.31%	-99.21%	-99.02%	-98.63%	-97.85%
pbmsrch	1.08e+04	-67.65%	-51.11%	-18.02%	33.68%	34.07%
qsort	8.00e+06	-4.00%	-3.99%	-3.99%	-3.97%	-3.92%
rawcaudio	1.87e+05	-93.79%	-93.79%	-93.79%	-92.32%	-84.66%
rawdaudio	1.55e+05	-92.51%	-92.51%	-92.51%	-92.05%	-84.12%
sha	1.78e+04	-37.20%	-30.73%	-23.18%	-14.02%	3.28%
susan	3.81e+05	-2.35%	-2.35%	-1.84%	0.58%	2.04%

TABLE XI
INSTRUCTION COUNTS

Benchmark	Unlimited	8 Change	16 Change	32 Change	64 Change	128 Change
basicmath	4.97e+07	1.57%	1.46%	1.42%	1.42%	1.39%
bf	5.62e+07	15.45%	14.53%	14.07%	13.61%	13.51%
bitcnts	1.04e+08	36.42%	32.98%	31.24%	31.12%	30.58%
crc_32	5.07e+07	25.29%	23.44%	22.51%	22.05%	21.86%
dijkstra	1.17e+08	26.06%	24.52%	23.75%	23.29%	22.77%
fft	3.05e+07	1.84%	1.73%	1.68%	1.66%	1.63%
patricia	7.24e+07	1.76%	1.44%	1.56%	1.44%	1.47%
pbmsrch	1.70e+05	4.93%	4.89%	4.88%	4.87%	4.87%
qsort	1.46e+07	1.21%	2.80%	1.12%	3.02%	2.90%
rawcaudio	1.01e+08	9.09%	7.61%	6.88%	6.49%	5.86%
rawdaudio	7.98e+07	11.45%	9.59%	8.66%	8.18%	7.96%
sha	2.26e+07	25.86%	23.22%	20.74%	20.73%	20.70%
susan	6.87e+07	25.51%	21.64%	21.63%	21.63%	21.60%

because this has the potential to go against our original motivation of limiting stack size, as we can see in the pronounced increase in stack size in several benchmarks such as *basicmath*, *dijkstra*, *rawcaudio*, *rawdaudio*, and *sha* when the recursion depth limit increases from 64 to 128.

It is worth noting that limiting recursion depth does increase the stack frame sizes of the generated recursive functions, resulting from the extra recursion depth counter parameter. This may increase maximum stack usage when the recursion depth limit is not significantly smaller than but on par with the number of recursive calls, as evidenced by the increase in maximum stack usage of the *susan* benchmark at a recursion depth limit of 64 and 128. However, the increase, even if present, is slight (as using the recursion depth counter adds a single integer of only a few bytes to the generated recursive functions' stack frames), and most of the time, with the recursion depth limit far less than the number of recursive calls, the benefits greatly outweigh such a cost.

c) Instruction count: Table XI compares the instruction counts under different recursion depth limits. There is a noticeable increase in instructions executed when limiting recursion depth. This stems from passing an extra recursion depth counter parameter to the generated recursive functions, incrementing and comparing the recursion depth counter in each invocation, saving the next values of variant variables when recursion depth reaches the limit, as well as extracting the next values when calling the recursive function again outside of the recursive function. As increasing the recursion depth limit leads to less saving to and loading from the Recursion State Structure instance, there is a decrease in the number of instructions executed. However, this decrease is slight, and overall, the instruction count is not sensitive to the recursion depth limit.

2) Results of Invoking Recursive Function Containing Loop Iterations:

a) Wear leveling effectiveness: Table XII compares the writes on the hottest stack address under different loop iterations. It is straightforward that, with the increase in loop iterations, writes on the hottest stack addresses increase rapidly, as the writes within each stack frame of the recursive function are repeated every loop iteration. Thus, it is vital to limit the number of loop iterations when retaining a portion of the loop in the recursive function to avoid compromising wear leveling effectiveness.

TABLE XII
WRITES ON THE HOTTEST STACK ADDRESS

Benchmark	0	8 Change	16 Change	32 Change	64 Change
basicmath	5.60e+04	94.31%	406.32%	527.37%	531.60%
bf	1.56e+05	0.00%	0.00%	0.00%	0.00%
bitcnts	1.18e+02	433.90%	1088.14%	3,194.92%	6372.88%
crc_32	1.37e+06	0.00%	0.00%	0.00%	0.00%
dijkstra	1.07e+04	781.07%	1660.62%	4124.89%	8331.87%
fft	7.76e+02	317.78%	787.11%	1675.52%	2893.17%
patricia	2.18e+04	0.00%	0.00%	0.00%	0.00%
pbmsrch	6.10e+01	457.38%	929.51%	1595.08%	2290.16%
qsort	2.30e+04	0.13%	0.37%	0.88%	1.84%
rawcaudio	1.41e+03	513.01%	1093.07%	2247.95%	4978.36%
rawdaudio	1.46e+03	497.74%	1078.85%	2306.71%	4821.29%
sha	1.46e+04	747.72%	1550.89%	3559.21%	6296.50%
susan	1.78e+04	1120.12%	2609.94%	5424.05%	10710.91%

TABLE XIII
MAXIMUM STACK USAGES

Benchmark	0	8 Change	16 Change	32 Change	64 Change
basicmath	3.38e+04	-70.93%	-82.86%	-88.73%	-91.48%
bf	2.62e+05	-78.37%	-87.45%	-91.99%	-94.31%
bitcnts	4.80e+06	-68.75%	-84.37%	-92.18%	-96.08%
crc_32	4.38e+07	-68.75%	-84.37%	-92.19%	-96.09%
dijkstra	5.47e+04	-79.72%	-89.73%	-94.70%	-95.41%
fft	1.97e+05	-74.89%	-87.36%	-93.60%	-96.63%
patricia	5.26e+05	-82.83%	-91.12%	-95.26%	-97.33%
pbmsrch	1.08e+04	-70.01%	-77.10%	-79.17%	-79.17%
qsort	8.00e+06	-2.75%	-3.37%	-3.69%	-3.84%
rawcaudio	1.87e+05	-76.16%	-88.04%	-93.80%	-93.80%
rawdaudio	1.55e+05	-75.09%	-87.50%	-92.52%	-92.52%
sha	1.78e+04	-31.90%	-38.10%	-41.96%	-43.22%
susan	3.81e+05	-2.44%	-2.44%	-2.44%	-2.44%

b) Maximum stack usage: Table XIII compares the maximum stack usages under different loop iterations. With the increased loop iterations of the retained loop within the recursive function, maximum stack usage decreases before gradually stabilizing. This is because more loop iterations lead to fewer calls to the recursive function. Thus, fewer allocated stack frames and reduced stack usage. Quantitatively, the number of calls to the recursive function, which is also the number of allocated stack frames, equals the ceiling of the number of iterations of the original loop divided by loop iterations. According to such a relationship, there will be gradually diminishing returns in reducing recursive function invocations with the increase in loop iterations. Thus, maximum stack usage will gradually stabilize.

c) Instruction count: Table XIV compares the instruction counts under different loop iterations. There is a noticeable

TABLE XIV
INSTRUCTION COUNTS

Benchmark	0	8 Change	16 Change	32 Change	64 Change
basicmath	4.97e+07	1.81%	1.68%	1.63%	1.63%
bf	5.62e+07	19.74%	18.49%	17.86%	17.23%
bitcnts	1.04e+08	58.61%	54.18%	51.92%	51.77%
crc_32	5.07e+07	40.13%	37.60%	36.33%	35.70%
dijkstra	1.17e+08	28.08%	25.84%	24.72%	24.06%
fft	3.05e+07	2.75%	2.61%	2.55%	2.52%
patricia	7.24e+07	2.38%	2.11%	2.22%	2.22%
pbmsrch	1.70e+05	4.45%	4.39%	4.37%	4.37%
qsort	1.46e+07	1.08%	0.98%	2.56%	2.56%
rawcaudio	1.01e+08	26.93%	24.69%	23.57%	23.00%
rawdaudio	7.98e+07	29.97%	27.41%	26.14%	25.48%
sha	2.26e+07	63.47%	60.18%	57.13%	57.13%
susan	6.87e+07	62.02%	57.47%	57.46%	57.46%

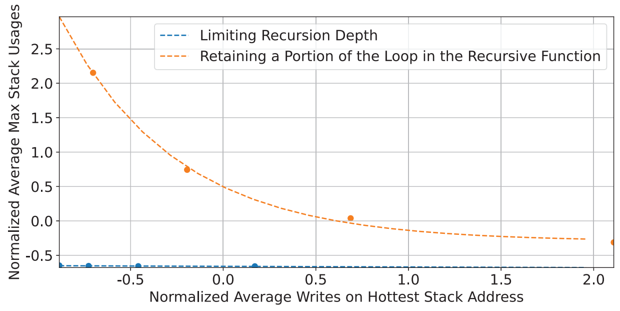


Fig. 10. Average maximum stack usages and writes on the hottest stack address under different recursion depth limits and loop iterations.

increase in instructions executed, which slowly declines and stabilizes with the increase in loop iterations. This is because retaining a portion of the loop in the recursive function involves initializing a loop iteration counter in each invocation of the recursive function, as well as incrementing and comparing the loop iteration counter in each loop iteration. With the increase in loop iterations, fewer calls to the recursive function will be made. Thus, the instructions for calling the recursive function and initializing the loop iteration counter are reduced. However, due to diminishing returns in reducing recursive function invocations, such a reduction in instructions will also gradually level out. In addition, as the total number of loop iterations in all invocations of the recursive function remains constant, instructions related to incrementing and comparing the loop iteration counter in each loop iteration remain unchanged. Thus, the number of instructions executed stabilizes with the increase in loop iterations.

3) Comparison of the Two Measures to Limit Recursion Depth: To determine which approach is better at reducing stack memory usage, we acquire maximum stack usages and writes on the hottest stack address averaged over all benchmarks under different recursion depth limits (for iteratively invoking a depth-limited recursive function) and loop iterations (for invoking a recursive function containing loop iterations) from Tables IX, X, XII, and XIII to draw the scatter plot depicted in Fig. 10.

Under the same average writes on the hottest stack address, iteratively invoking a depth-limited recursive function consistently provides lower average maximum stack usages, as depicted in Fig. 10. It is possible to reduce average writes on the hottest stack address without sacrificing average maximum stack usage when iteratively invoking a depth-limited recursive function, which is achieved by increasing the recursion depth limit. In comparison, there is an apparent tradeoff between average writes on the hottest stack address and average maximum stack usage when invoking a recursive function containing loop iterations. Thus, setting a mid-high recursion depth limit and iteratively invoking a depth-limited recursive function is the better approach, as it excels in both stack memory overhead and the effectiveness of wear leveling.

TABLE XV
WRITES ON THE HOTTEST STACK ADDRESS

Benchmark	No WL	Non-selective	Selective
basicmath	301	785	163
bf	15309	15592	15592
bitcnts	2	14064	2
crc_32	668	668	668
dijkstra	6120	30169	652
fft	46	1095	740
patricia	20864	10400	9080
pbmsrch	3	6	6
qsort	240	645	641
rawcaudio	0	21902	22
rawdaudio	0	21902	22
sha	39	156	154
susan	7	439	20

D. Evaluation on Cache-Enabled Architecture

When a cache is used, a memory write only happens when a dirty cache line is written back to the memory. To evaluate the wear leveling performance of Loop2Recursion and assess the effectiveness of selective loop transformation, we compare the number of writes on the hottest memory address between no wear leveling, nonselective Loop2Recursion, and selective Loop2Recursion in Table XV.

The situation here is different from that on a cacheless architecture. Some benchmarks, such as *bitcnts* and *fft*, exhibit very high locality, so there are few writebacks to the hottest address without wear leveling. However, after applying Loop2Recursion, the program's locality decreases, resulting in increased writebacks. However, this problem is generally mitigated when performing selective loop transformation. This is because the selective loop transformation skips transforming loops with high locality. An exception is the *bf* benchmark, where the writes to the hottest stack address are caused by writing to a local variable in the main function outside a loop. As a result, all methods exhibit similar writebacks.

We further the cache hit ratio between the no wear leveling, nonselective Loop2Recursion, and selective Loop2Recursion in Table XVI. For most benchmarks, although the cache hit ratio drops after applying Loop2Recursion, selective loop transformation raises it again (to 96.9% on average), and it is generally on par with no wear leveling (98.3% on average). Thus, under selective loop transformation, Loop2Recursion has a negligible impact on cache performance. In conclusion, Loop2Recursion can also work well on a cache-enabled architecture, as it significantly reduces the maximum write count while slightly affecting the cache hit ratio.

TABLE XVI
CACHE HIT RATIO

Benchmark	No WL	Non-selective	Selective
basicmath	0.99	0.99	0.99
bf	0.93	0.85	0.84
bitcnts	1.00	0.97	1.00
crc_32	1.00	0.99	0.99
dijkstra	0.97	0.86	0.96
fft	1.00	0.98	0.99
patricia	0.95	0.94	0.94
pbmsrch	0.99	0.95	0.93
qsort	0.96	0.95	0.95
rawcaudio	1.00	0.82	1.00
rawdaudio	1.00	0.81	1.00
sha	1.00	1.00	1.00
susan	1.00	0.99	1.00

VIII. CONCLUSION

We present Loop2Recursion, a compiler-assisted stack wear leveling technique for NVM that automatically transforms loops, significantly contributing to highly unbalanced stack writes, into recursive functions. To reduce both the space overhead and performance overhead of this transformation. In addition, as deep recursions are still prone to stack overflow exceptions, we propose two approaches to limit recursion depth, namely, iteratively invoking a depth-limited recursive function and a recursive function containing loop iterations. Furthermore, we propose selective loop transformation for cache-enabled architectures, where we preserve loops not causing uneven stack writes. Experimental results demonstrate that Loop2Recursion can significantly reduce the number of writes on the hottest stack address and improve the lifetime of NVM with low overhead, outperforming the state-of-the-art dynamic stack [20], [23], that our optimizations to the transformation scheme are effective, that iteratively invoking a depth-limited recursive function is the better approach to limiting recursion depth, and that Loop2Recursion can also work well on a cache-enabled architecture with only a minor reduction in cache hits.

REFERENCES

[1] M. Kim, I.-J. Chang, and H.-J. Lee, "Segmented tag cache: A novel cache organization for reducing dynamic read energy," *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1546–1552, Oct. 2019.

[2] B. Kim et al., "PCM: Precision-controlled memory system for energy efficient deep neural network training," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2020, pp. 1199–1204.

[3] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the Intel optane byte-addressable NVM," in *Proc. Int. Symp. Memory Syst.*, 2019, pp. 304–315.

[4] 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 *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2009, pp. 14–23.

[5] V. Gogte et al., "Software wear management for persistent memories," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 45–63.

[6] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 24–33.

[7] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 14–23, 2009.

[8] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing PCM main memory lifetime," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2010, pp. 914–919.

[9] M. Zhao, L. Shi, C. Yang, and C. J. Xue, "Leveling to the last mile: Near-zero-cost bit level wear leveling for PCM-based main memory," in *Proc. IEEE 32nd Int. Conf. Comput. Design (ICCD)*, 2014, pp. 16–21.

[10] S. Kim, H. Jung, W. Shin, H. Lee, and H.-J. Lee, "HAD-TWL: Hot address detection-based wear leveling for phase-change memory systems with low latency," *IEEE Comput. Archit. Lett.*, vol. 18, no. 2, pp. 107–110, Jul.–Dec. 2019.

[11] H. Lee, H. Jung, H.-J. Lee, and H. Kim, "Bit-width reduction in write counters for wear leveling in a phase-change memory system," *IEIE Trans. Smart Process. Comput.*, vol. 9, no. 5, pp. 413–419, 2020.

[12] M. Kim, H. Lee, H. Kim, and H.-J. Lee, "WL-WD: Wear-leveling solution to mitigate write disturbance errors for phase-change memory," *IEEE Access*, vol. 10, pp. 11420–11431, 2022.

[13] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. 46th ACM/IEEE Design Autom. Conf.*, 2009, pp. 664–669.

[14] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y. M. Wang, "Age-based PCM wear leveling with nearly zero search cost," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 453–458.

[15] L. Yu, T. Chen, and J. Wu, "A software-hardware collaborating framework for wear leveling on phase change memory," in *Proc. IEEE 14th Int. Conf. High Perform. Comput. Commun. IEEE 9th Int. Conf. Embedded Softw. Syst.*, 2012, pp. 1360–1367.

[16] C. Pan, M. Xie, J. Hu, M. Qiu, and Q. Zhuge, "Wear-leveling for PCM main memory on embedded system via page management and process scheduling," in *Proc. IEEE 20th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2014, pp. 1–9.

[17] H. A. Khouzani, Y. Xue, C. Yang, and A. Pandurangi, "Prolonging PCM lifetime through energy-efficient, segment-aware, and wear-resistant page allocation," in *Proc. Int. Symp. Low Power Electron. Design*, 2014, pp. 327–330.

[18] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proc. 1st ACM SIGOPS Conf. Timely Results Oper. Syst.*, 2013, pp. 1–17.

[19] S. Yu et al., "WAlloc: An efficient wear-aware allocator for non-volatile main memory," in *Proc. IEEE 34th Int. Perform. Comput. Commun. Conf. (IPCCC)*, 2015, pp. 1–8.

[20] Q. Li, Y. He, Y. Chen, C. J. Xue, N. Jiang, and C. Xu, "A wear-leveling-aware dynamic stack for PCM memory in embedded systems," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2014, pp. 1–4.

[21] W. Li, L. Wu, M. Yuan, C. J. Xue, J. Xue, and Q. Li, "Loop2Recursion: Compiler-assisted wear leveling for non-volatile memory," in *Proc. IEEE 38th Int. Conf. Comput. Design (ICCD)*, 2020, pp. 581–588.

[22] X. Chen et al., "A wear-leveling-aware fine-grained allocator for non-volatile memory," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*, 2019, pp. 1–6.

[23] 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 *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2019, pp. 228–233.

[24] 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 *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization*, 2001, pp. 3–14.

[25] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 2–13.

[26] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 34–45, 2009.

[27] W. Xu, J. Liu, and T. Zhang, "Data manipulation techniques to reduce phase change memory write energy," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, 2009, pp. 237–242.

[28] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2009, pp. 2–13.

[29] S. Cho and H. Lee, "Flip-N-write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 347–357.

[30] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory," in *Proc. Design, Autom. Test Europe*, 2011, pp. 1–6.

[31] Q. Li, Y. Zhao, J. Hu, C. J. Xue, E. Sha, and Y. He, "MGC: Multiple graph-coloring for non-volatile memory based hybrid scratch-pad memory," in *Proc. 16th Workshop Interact. Compilers Comput. Archit. (INTERACT)*, 2012, pp. 17–24.

[32] Z. Shao, Y. Liu, Y. Chen, and T. Li, "Utilizing PCM for energy optimization in embedded systems," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2012, pp. 398–403.

[33] L. A. Bathen and N. Dutt, "HaVOC: A hybrid memory-aware virtualization layer for on-chip distributed scratchpad and non-volatile memories," in *Proc. DAC Design Autom. Conf.*, 2012, pp. 447–452.

[34] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, 2012, pp. 191–196.

[35] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Write activity reduction on non-volatile main memories for embedded chip multiprocessors," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 3, pp. 1–27, 2013.

[36] R. Liu, P. Jin, Z. Wu, X. Wang, S. Wan, and B. Hua, "Efficient wear leveling for PCM/DRAM-based hybrid memory," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun. IEEE 17th Int. Conf. Smart City IEEE 5th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, 2019, pp. 1979–1986.

[37] H. Wang, Z. Shen, M. Zhao, X. Cai, and Z. Jia, "CLOCK-RWRF: A read-write-relative-frequency page replacement algorithm for PCM and DRAM of hybrid memory," in *Proc. IEEE 22nd Int. Conf. High Perform. Comput. Commun. IEEE 18th Int. Conf. Smart City IEEE 6th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, 2020, pp. 189–196.

[38] A. Filinski, "Recursion from iteration," *Lisp Symbolic Comput.*, vol. 7, no. 1, pp. 11–37, 1994.

[39] J. Geuvers, *Inductive and Coinductive Types With Iteration and Recursion*, University of Nijmegen, Nijmegen, The Netherlands, 1992.

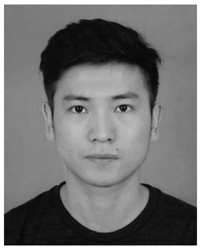
[40] Y. A. Liu and S. D. Stoller, "From recursion to iteration: What are the optimizations?" *SIGPLAN Notices*, vol. 34, no. 11, pp. 73–82, Nov. 1999. [Online]. Available: https://doi.org/10.1145/328691.328700

[41] C. Hakert et al., "Software-managed read and write wear-leveling for non-volatile main memory," *ACM Trans. Embedded Comput. Syst.*, vol. 21, no. 1, p. 5, Jan. 2022.

[42] C.-K. Luk et al., "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

Jifeng Wu received the B.S. degree in software engineering from Wuhan University, Wuhan, China, in 2022.

His research interests include programming languages, program analysis, and software engineering.



Wei Li received the B.S. degree in printing engineering and the M.S. degree in computer science and engineering from Wuhan University, Wuhan, China, in 2017 and 2020, respectively. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW, Australia, under the supervision of Prof. J. Xue.

His research interests include programming languages, program analysis, and nonvolatile memory.

Libing Wu received the Ph.D. degree from Wuhan University, Wuhan, China, in 2006.

He is currently a Professor with the School of Cyber Science and Engineering, Wuhan University. He was a Visiting Scholar with the Advanced Networking Lab, University of Kentucky, Lexington, KY, USA, in 2011. He was a Senior Visiting Fellow with the State University of New York, Albany, NY, USA, in 2017. His research interests include network security, Internet of Things, machine learning, and data security.

Mengting Yuan received the Ph.D. degree from Wuhan University, Wuhan, China, in 2006.

He is currently a Professor with the School of Computer Science, Wuhan University. His research interests are programming languages, compiler technology, and software engineering.

Chun Jason Xue (Senior Member, IEEE) received the B.S. degree in computer science and engineering from the University of Texas at Arlington, Arlington, TX, USA, in May 1997, and the M.S. and Ph.D. degrees in computer science from the University of Texas at Dallas, Richardson, TX, USA, in December 2002 and May 2007, respectively.

He is currently a Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research interests include memory and storage systems.

Jingling Xue (Fellow, IEEE) received the B.Eng. and M.Eng. degrees in computer science and engineering from Tsinghua University, Beijing, China, in 1984 and 1987, respectively, and the Ph.D. degree in computer science and engineering from Edinburgh University, Edinburgh, U.K., in 1992.

He is a Scientia Professor with the School of Computer Science and Engineering, UNSW Sydney, Sydney, NSW, Australia, where he leads the Programming Languages and Compilers Group. His research spans programming languages, compiler technology, and program analysis. He has supervised 29 Ph.D. students to completion. He is interested in sharing the outcomes of his research projects in the form of open-source tools, such as SVF (https://svf-tools.github.io/SVF) and Qilin (https://qilinpta.github.io/Qilin).

Dr. Xue has won many best/distinguished/test-of-time awards at CGO, ECOOP, ICSE, ASE, and ISSTA. He has served as the PC Chair/Co-Chair/Vice-Chair in 16 international conferences, including LCTES'13, CC'18, and CGO'20, and as a PC member in over 200 international conferences.

Qingan Li received the B.S. and Ph.D. degrees in computer science from Wuhan University, Wuhan, China, in 2008 and 2013, respectively, and the Ph.D. degree in computer science and engineering from the City University of Hong Kong, Hong Kong, in 2014.

He is currently an Associate Professor with the School of Computer Science, Wuhan University. His research interests include program analysis, memory management, and embedded systems.