A Recursive Counter for Linked-List Return Stack Buffer

Ke Sun, Kekai Hu, Rodrigo Branco

Intel - STrategic Offensive Research & Mitigations (STORM)

V1.42 (For the latest version, please check: https://github.com/intelstormteam/Papers)

0. OBJECTIVES

While the scope of our daily work focuses on Intel's product security, there are ideas from our research that are not necessarily security-related, but may still be beneficial to the overall platform in other aspects (some of those ideas end up in real implementations, some are disclosed as patents or published in other forms). This paper shares one of the ideas that our team considered, including the theory and possible implementation options, with the objective to give a ground base for other designers and researchers to improve upon and also for the industry to consider.

1. INTRODUCTION

In order to avoid pipeline stalls, modern CPUs make use of internal buffers to store speculative branch targets that would be otherwise slow to resolve in committed execution. Different buffers can be used for prediction of different branch types, e.g., a buffer for direct branches, a buffer for indirect branches or a buffer specifically for return instructions. Among these buffers, the one for return instructions is usually called return stack buffer (RSB).

A RSB can be implemented in different sizes (with different number of entries) and different data structures (a stack-like buffer or a linked list, etc.). The size of RSB is usually in the range of 4 - 16 entries, with a stack-like design or a linked-list design according to published information regarding processor architecture (Ref[1]). Among all the existing RSB implementations, this proposed idea is targeting the linked-list version of RSB to improve its performance by increasing

the chance of correct predictions and reduce the chance of RSB overflows and underflows, which is also desirable for certain speculative side-channel mitigation.

2. EXISTING SOLUTIONS

As one of the most common data structures, linked list (Ref[2]) is widely used in computer designs. In this paper, we use a typical linked-list RSB design as an example to illustrate our proposal, while the idea itself shall also be applicable in general to other RSB designs as well. As shown in Figure 1, the RSB entries are organized in a single linked list where each entry stores a cached return address based on previous call instruction, a previous pointer that links the current entry to the "previous entry" that contains the return address of an earlier call on the logical call stack. Two additional pointers, a read pointer and a write pointer, are associated with this linked list to keep track of the operations to the list.

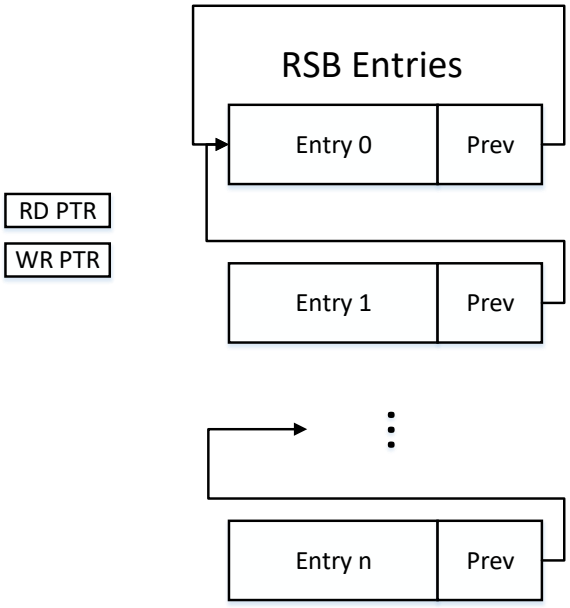

Figure. 1 Linked list RSB design

This linked-list RSB design has the following characteristics:

1. It is a singly linked list with fixed total number of entries since it is a hardware implementation. A per-entry “previous pointer" links the list toward lower entry indices.
2. The list is populated by write toward higher entry indices and consumed by read toward lower entry indices. The “next entry” for write is always obtained by simply incrementing the entry index until an RSB overflow, while the “previous entry” for read is specified by the “previous pointer” of each list entry.
3. Upon a write operation, the write pointer increments after writing the list (until it reaches the end of the list), while the read pointer is always updated to the newly written entry (the entry before the updated write pointer), and the “previous pointer” of the newly written entry is updated to the old value of the read pointer.
4. Upon a read operation, the write pointer is not affected, the read pointer is updated to the “previous pointer” of the read entry after the read operation.
5. The “previous pointer” of entry 0 always points back to entry 0

In the initial state, both write and read pointers point to entry 0 and all the RSB entries are empty. Every time a call instruction gets executed, a write operation happens to the RSB, the expected return address of this call is written to the RSB entry pointed by the write pointer, and then the write pointer increments to the next entry. In the meanwhile, the read pointer is also updated and always points to the newly written entry. For example, at the initial state, if there is a write, entry 0 gets the return address, the write pointer is updated to entry 1 and the read pointer still points to entry 0. The previous pointer of entry 0 will be updated to point back to itself. After that, if there is another write, entry 1 gets the address, the write pointer moves to entry 2 and the read pointer now points to entry 1 (previous pointer of entry 1 will point to entry 0). Every time a return instruction gets executed, a read operation happens to the RSB, the return address of the entry pointed by the read pointer will be consumed and the read pointer will move to the entry pointed by the “previous pointer” of the consumed entry. Continuing the previous example, when the write pointer is at entry 2 and the read pointer is at entry 1, a read operation will read the return address from entry 1 and then the read pointer will point back to entry 0. Then upon the next call, the write pointer will move from entry 2 to entry 3 and the read pointer will move

from entry 0 directly to entry 2 (the newly written entry) while the “previous pointer” of entry 2 will be updated to entry 0 (value of the old read pointer), skipping the already consumed entry 1.

There are two special cases of this linked-list RSB design: overflow and underflow. In a case that a new write request happens when the write pointer is already pointing to the last entry in the table (e.g., entry n in Figure. 1), the write pointer will “overflow” and point back to the first entry (entry 0), then it will start overwrite data from there. In a case that a read request happens when the read pointer is already pointing to the first entry and the entry is already used once, an RSB underflow happens. There could be different underflow handling mechanisms, for example: 1) the branch predictor will redirect to other branch target buffers (such as indirect branch buffer) and use the values in other buffers for the prediction, or 2) the branch predictor keeps reading the value in RSB entry 0 and use it for prediction.

Although this current linked-list RSB design works well for most return predictions, it cannot efficiently handle the cases like deep recursive calls which is commonly used in many algorithms and will very likely result in an RSB overflow/underflow.

Considering a recursive function with the call-stack depth of 100 (i.e. the same call is invoked 100 times before starting to return) in a system that has a 64-entry RSB. The recursive calls will eventually purge all the previous entries in the RSB list. Thus, when the returns happen, all the return predictions after the returns of the recursive calls will miss, since all the previously cached return addresses in the list are already lost.

3. OUR IDEA

In this paper, we propose to add a recursive counter to each entry of the RSB to keep track of the recursive calls/returns. With such design, only one RSB entry is needed for arbitrary depth of recursive calls/returns as long as the recursive count is within the max limit of the counter. In this way, the chance of RSB overflow/underflow will be largely reduced and the RSB capacity of caching effective return addresses will be optimized. This would increase the successful rate of return address predictions and boost the overall performance.

The RSB structure with the proposed recursive counter is demonstrated in Figure 2. The recursive counter is used to count the number of recursions of the same call with a default value 0. Its size can vary depending on the design requirements and the resource availabilities in the system. With such counters implemented, a write operation first compares the current return address with the last written return address, pointed by the read pointer, and checks if they are the same. If yes, instead of writing to a new RSB entry, the write operation simply increments the recursive counter of the last written entry; a read operation first checks if the counter of the current read entry (pointed by the read pointer) is 0, if yes, it proceeds with the default behavior, otherwise, it simply reads the return address value from the current read entry and decrements the counter. The following example is given to further illustrate how the RSB with the proposed recursive counter handles the recursive calls and returns.

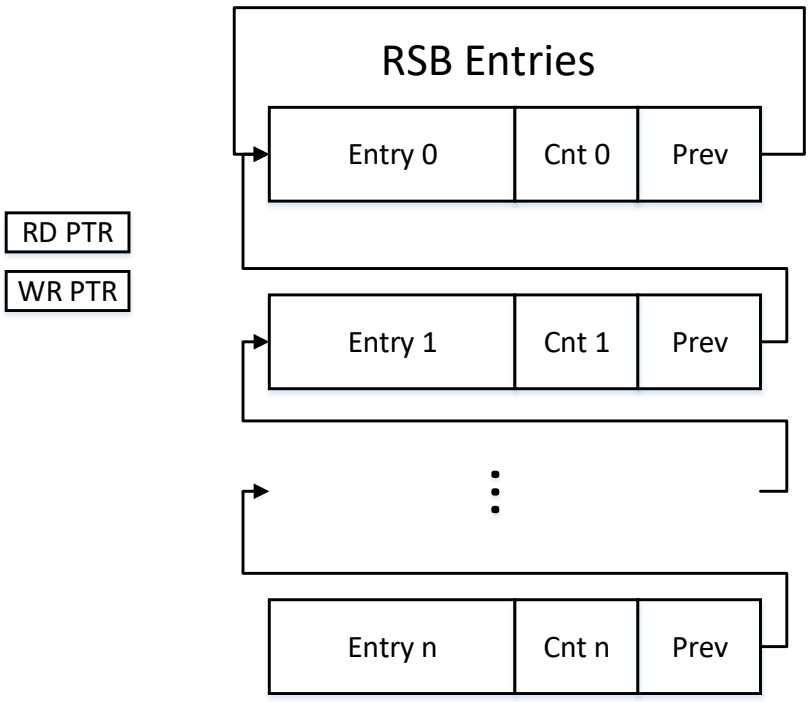

Fig. 2 Proposed design in the RSB table

Considering a recursive routine with the call-stack depth of 100 and a linked-list RSB with 64 entries, without the proposed design, there will be 100 consecutive writes of the same return address (assuming it is A), which will overflow the RSB after at most 64 writes. As a result, all the earlier return addresses saved in RSB before the recursive calls will be overwritten and the return address of all 64 entries will be populated with the same value A. Then when the recursive routines start to return, 100 continuous reads will happen and the RSB will underflow after at most 64 reads, depending on where do the writes start (e.g., if the writes start at entry 0, it will underflow after 36 reads). In this example, assuming that the read pointer (rd_ptr) points to entry 20 and the write pointer (wr_ptr) points to entry 21 right before the recursive calls, the following state change flow shows the process:

Initial State: *rd_ptr = 20, wr_ptr = 21,* *entry 0 to 20: earlier return addresses*

Call/Write 1: *rd_ptr = 21, wr_ptr = 22, entry 21: A,* *entry 0 to 20: earlier return addresses*

Call/Write 2: *rd_ptr = 22, wr_ptr = 23, entry 21 to 22: A,* *entry 0 to 20: earlier return addresses*

…

Call/Write 42: rd_ptr = 62, wr_ptr = 63, entry 21 to 62: A, entry 0 to 20: earlier return addresses

Call/Write 43: [RSB overflow]

rd_ptr = 63, wr_ptr = 0, entry 21 to 63: A, entry 0 to 20: earlier return addresses

Call/Write 44: rd_ptr = 0, wr_ptr = 1, entry 0, 21 to 63: A, entry 1 to 20: earlier return addresses

…

Call/Write 64: rd_ptr = 20, wr_ptr = 21, entry 0 to 63: A (all earlier return addresses overwritten)

…

Call/Write 100: rd_ptr = 56, wr_ptr = 57, entry 0 to 56: A

Ret/Read 1: rd_ptr = 55, wr_ptr = 57, entry 0 to 55: A

Ret/Read 2: rd_ptr = 54, wr_ptr = 57, entry 0 to 54: A

…

Ret/Read 56: rd_ptr = 0, wr_ptr = 57, entry 0: A

Ret/Read 57: [RSB underflow]

rd_ptr = 0, wr_ptr = 57

Ret/Read 58: rd_ptr = 0, wr_ptr = 57

…

Ret/Read 100: rd_ptr = 0, wr_ptr = 57

From the flow above, it can be clearly seen that the RSB overflows at write 44 and underflows at read 57, and all the earlier return addresses cached in RSB before the recursive calls are lost/overwritten.

In the proposed design, a recursive counter is introduced to each RSB entry to properly handle the recursive operations. For the same case of 100 recursive calls and returns, the new flow would be:

Initial State: rd_ptr = 20, wr_ptr = 21,

entry 0 to 20: earlier return addresses, all recursive counter = 0

Call/Write 1: rd_ptr = 21, wr_ptr = 22, entry 21 (return address = A, recursive counter = 0)

entry 0 to 20: earlier return addresses, all recursive counter = 0

Call/Write 2: *rd_ptr = 21, wr_ptr = 22, entry 21 (return address = A, recursive counter = 1)*

entry 0 to 20: earlier return addresses, all recursive counter = 0

…

Call/Write 100: *rd_ptr = 21, wr_ptr = 22, entry 21 (return address = A, recursive counter = 99)*

entry 0 to 20: earlier return addresses, all recursive counter = 0

Ret/Read 1: *rd_ptr = 21, wr_ptr = 22, entry 21 (return address = A, recursive counter = 98)*

entry 0 to 20: earlier return addresses, all recursive counter = 0

Ret/Read 2: *rd_ptr = 21, wr_ptr = 22, entry 21 (return address = A, recursive counter = 97)*

entry 0 to 20: earlier return addresses, all recursive counter = 0

…

Ret/Read 99: *rd_ptr = 21, wr_ptr = 22, entry 21 (return address = A, recursive counter = 0)*

entry 0 to 20: earlier return addresses, all recursive counter = 0

Ret/Read 100: *rd_ptr = 20, wr_ptr = 22, (entry 21 is invalid)*

entry 0 to 20: earlier return addresses, all recursive counter = 0

As shown by the new flow, both the overflow and the underflow of RSB in this case are prevented by the implementation of recursive counters and all the earlier return addresses are kept intact in RSB after the deep recursive calls/returns.

4. CONCLUSION

In this paper, we propose a new idea of adding a per-entry recursive counter in the return stack buffer and use a linked-list design as an example to illustrate the possible implementation of the idea. With the help of such counter, RSB can better handle deep recursive routines and reduce the chance of overflows and underflows, thus improving the overall performance by reducing the miss-predictions of return instructions.

Legal Notices & Disclaimers

This work is disclosed as a security research work for the community to consider, leverage and review. It has not been fully implemented or comprehensively tested for all potential corner cases and therefore is not presented nor guaranteed as a mature solution.

Intel provides these materials as-is, with no express or implied warranties. No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All products, dates, and figures specified are preliminary, based on current expectations, and are subject to change without notice.

Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration**. No product or component can be absolutely secure**. Check with your system manufacturer or retailer or learn more at http://intel.com.

Intel and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

References

1. Agner Fog, "The microarchitecture of Intel, AMD, and VIA CPUs", Retrieved 2017-03-22

2. Thomas H. Cormen, Charles E. Leiserson *et al.*, “Introduction to Algorithms”, MIT Press. pp. 205–213, 501–505. ISBN 0-262-03293-7