WRITE COUNT PREDICTION FOR ENCRYPTED MAIN MEMORY

Team 3

Team Members

● Ajit Mathew

● Daulet Talapkaliyev

● Sudha Ravali Yellapantula

● Pranavi Rambhakta

● Xiaolong Li

Motivation

CPU executes code and operates on data and addresses in plaintext only, and writes data into memory and reads data from memory, through buses by providing the corresponding logical addresses over the the address bus. Physical and logical attacks on could happen in a lot of levels, the memory is usually the target, so encryption standard like DES and AES are established. So we will encrypt plaintext into ciphertext to be written into memory, while also decrypt the ciphertext to plaintext to be read from memory. However, this takes time and brings overhead to system speed. If we look into this problem, the secret key, addresses and the write count of the memory block, would be needed to decrypt the ciphertext. Write count prediction would be significant to reduce the time for decryption. In Intel SGX(a new CPU extension for executing secure code in Intel processors), it requires write count of memory location before decrypting it.

Design

Key Idea:

To be able to successfully predict the write count of a cache miss, the algorithm should be able to do the following:

● Predict the cache miss address This is done using a mechanism similar to linked-list based prefetch. The intuition is that if a cache miss pattern happened in the past, then it is highly likely that this will happen again.

● Predict the write count of the miss address To predict the write count, we append each cache block with approximate write count of the next N predicted misses. Also to improve the chances of correct prediction, we use the stored approximate write count to generate a range of predicted write counts.

Components:

The design can be divided into 4 main component:

● Main Memory(MM): This is the backing store of cache

● Cache with Write Count History Holder(WCHH): Unlike a normal cache block, our design modifies cache like to hold the actual write count (AWC) and write count of next N predicted misses in a container we named write count history holder.

● Predicted Write Count Buffer(pwcBuffer): This buffer stores the predicted write count of the next miss. On a cache miss, the actual write count is compared to each item in the pwcBuffer.

Performance Tuning Parameters

● WCHH Size: If we increase size of the WCHH, it means we are tracking longer miss patterns. To do this we have to increase the size of the pattern fifo and the size of cache block. But this will improve the WC prediction rate.

● Prediction Range: This refers to the range of prediction write count generated for a single approximate write count stored in WCHH. Increasing this increases the size of the pwcBuffer but it also improves prediction rate.

● Writeback Frequency: Every time WCHH is updated, the cache block becomes dirty. This has a high bandwidth overhead for cases when cache block was fetched was read only. To reduce the bandwidth, we do not update WCHH every time but only at a certain rate defined by the writeback frequency. This reduces the bandwidth but possibly reduces prediction rate.

● Pattern Fifo: This is used to keep track of cache miss history. On every miss, the address of the cache miss and the write count of miss address is pushed into the fifo. When an miss address is popped out of the pattern fifo to make space for new miss addresses, then the the previous N write count values in the pattern fifo is appended to the WCHH of the popped miss address.

Algorithm

We present the pseudo-code of the algorithm in this section

Microbenchmarks

Microbenchmark is a small program that measures the performance of a model. The write count prediction model for encrypted memory has been designed and it is to be tested on benchmarks to evaluate and analyze the performance of the model . The evaluation metric used is coverage which is the ratio of the correct predictions made over the total predictions, when cache misses occur. Micro-benchmarks are used to identify the problems associated with the model and fix them before the model is checked with the commercial and scientific macro-benchmarks. This helps us save a lot of time and money. This is because testing the model directly with macrobenchmarks is expensive and iteratively repeating this while optimizing the model would not be plausible.

The strategy that we have adopted is quite simple.

Test model with microbenchmarks Check the coverage Optimize the model

To test the model in the best case as well as the worst case scenario, we use good as well as bad benchmarks.

The micro-benchmarks that were used are:

● **Good benchmarks (Coverage ~ 100)**

○ Array traversal

○ Linked list traversal

○ Binary search tree creation using an array The coverage obtained for good benchmarks were almost 100%.

● **Bad benchmarks (Coverage <50)**

○ Random access memory in an array

○ Random memory access in binary search tree using an array

Implementation of random memory access in an array: Randomly access a node in an array and write to it. Since, any node could be accessed multiple times because of the random access, the write count of the address of each node is also varying i.e. random. This randomization has effected in a drop in the coverage of the cache model. As the randomization of the write count of the addresses increase, the coverage decreases further. The least coverage obtained was 20%. The bar plot shown below helps us visualize and compare the coverage for the microbenchmarks.

The coverage obtained for the linked list prefetching model was almost 0% when tested on the bad benchmarks. By testing on the bad benchmarks, the problems associated with the model could be identified which would help us better optimize the model.

Linked-List Prefetching

It is a known fact that the cache is not always effective, one reason for that may be that they are not large enough to hold a program's working set. Another reason might be that, memory access patterns don't exhibit behavior that matches a cache memory's demand-driven, line-structured organization. Linked list prefetching is used to design the cache memory controller such that it can predict memory addresses that are likely to be accessed soon and then prefetch the data into the cache.

In this method, the cache is first checked to see whether the required address is present or not. If the address is not present in the Cache, then its a miss. In this case, while fetching the miss address from the Main Memory, the addresses present in the history holders are also prefetched along with it. To maintain low overhead, we assumed the prefetch degree to be 2, ie only addresses present in the two history holders will get fetched along with the miss address. This means that for every miss, the total predictions would be incremented by a value of 2. Also, once the hit has occurred is due to the correct predictions, the value of correct predictions in incremented by 1. And if the same address is called again, it will be considered a hit due to its presence in the cache, but not due to prefetching.

Therefore, if it’s a hit, ie, if the required address is present in the cache, it is checked weather this hit is corresponds to the address prefetched. If it so the number of correct predictions are incremented and the coverage is calculated.

Coverage is calculated as the percentage of ratio of correct predictions and total predictions.

Each suite has a number of benchmarks, and due to the time constraints and the fact that we are mainly interested in bad benchmarks that produce bad results, we have chosen the below benchmarks to perform our tests as adviced and the following results were obtained. Since we are mainly interested in the Bad benchmarks that produce bad results, the commercial benchmarks that we focused on while conducting this experiment are :

● **SPEC2017 :**

○ *34 percent for 620.omnetpp_s (Discrete Event Simulation/Simulation of a 10Gbit Ethernet network)*

○ 1.3 percent for 657.xz_s (Data compression)

○ *0.4 percent for 605.mcf_s (Single-depot vehicle scheduling/Route planning)*

○ *0.14 percent for 631.deepsjeng_s (AI/Alpha-beta tree search for Chess)*

○ *0.07 percent for 625.x264_s (Video compression/H.264 video encoding)*

● **Other Benchmarks used :**

○ 3.8 percent for SPECjbb®2015

Macrobenchmarks (commercial benchmarks)

Since Intel SGX and similar memory encryption technologies primarily target commercial products, it is only fitting that we test our WC prediction algorithm on some commercial benchmarks. We focused on three standard commercial benchmark suites:

● *SPEC2017 (input date size = train/medium):*

○ *605.mcf_s (Single-depot vehicle scheduling/Route planning)*

○ *620.omnetpp_s (Discrete Event Simulation/Simulation of a 10Gbit Ethernet network)*

○ *625.x264_s (Video compression/H.264 video encoding)*

○ *631.deepsjeng_s (AI/Alpha-beta tree search for Chess)*

○ *641.leela_s (AI/Monte Carlo tree search for Go)*

● **PARSEC:**

○ *canneal (Simulated cache-aware annealing to optimize routing cost of a chip design)*

○ *raytrace (Real-time raytracing)*

○ *streamcluster (Online clustering of an input stream)*

● **SPECjbb2015:**

○ *composite (Single JVM process)*

While each suite has many different benchmarks (more than 70 in total), due to time constraints and the fact that many benchmarks result in close to 100% coverage, which are not very interesting for our design analysis, we focus only on a few interesting ones to analyse and present some bad (comparatively), moderately good and perfect results.

Macro-benchmark results

- SPEC2017 Y-axis: WC coverage; X-axis: (WCHH Size, Prediction Range), Writeback Frequency%
  - 605.mcf_s (Benchmark size: 685 MB; Write Count average: 21)
  - 620.omnetpp_s (Benchmark size: 226 MB; Write Count average: 151)

- 625.x264_s (Benchmark size: 148 MB; Write Count average: 7)
- 631.deepsjeng_s (Benchmark size: 6.8 GB; Write Count average: 11)

- ○ 641.leela_s (Benchmark size: 22 MB; Write Count average: 157)
- ● PARSEC Y-axis: WC coverage; X-axis: (WCHH Size, Prediction Range)
- ○ Canneal (WC avg: 7), Raytrace (WC avg: 9), StreamCluster (WC avg: 38)

● SPECJbb2015 Y-axis: WC coverage; X-axis: Prediction Range Size

○ **SPECJbb2015--composite (WC avg: 12)**

Analysis

Cost Analysis Our model has overheads to predict the write counts. The overheads can be summarized as follows:

● Main Memory Compression: If the write count history holder size is increased, each memory block will require higher compression in order to store more predicted write counts.

● Pattern FIFO Size: If the write count history holder size is increased, then the size of the pattern FIFO increases linearly.

● Predicted Write Count Buffer Size: If the prediction range is increased, then the size of the Predicted Write Count Buffer (pwcBuff) is increased linearly for a given write count history holder size.

● Bandwidth from WB due to WCHH: If writeback frequency is increased, then the bandwidth overhead from writebacks of dirty due to updated WCHH blocks increases.

Therefore, due to MM compression overhead and hardware overhead of larger pattern FIFO, increase in size of WCHH is more expensive than increasing Prediction Range. Moreover, since Prediction Range gives us some room for error, we can experiment with writeback frequency to minimize bandwidth overhead while keeping the coverage high.

Results analysis

Write count average is the most important data that can be used to determine how our algorithm will perform under certain benchmarks. Low average (ex. mcf, deepsjeng, etc.) allows the algorithm to take full advantage of the prediction range. When the prediction range is higher or equal to workload's write count average, at least half of all the actual write counts will be within the range. So even if we mispredict most miss patterns and the original write count predictions, we can still expect at least >50% of correct predictions. Larger write count average (ex. omnetpp, leela) on the other hand means that in order to get the correct prediction, algorithm now cannot rely solely on prediction range but needs to correctly predict the miss pattern. Even for the largest tested parameter values the coverage for omnetpp and leela is bellow 70%.

One distinct trend that can be seen in almost all graphs is that increase of write count history holder size has larger effect on the coverage compared to increase of prediction range. This happens because with large WCHH we are able to cover longer miss patterns and thus make more accurate predictions. The major issue with larger WCHH is that it results in hardware overhead, which is significantly larger compared to overhead from larger prediction range. Therefore, it is cheaper to have a relatively small WCHH size and a large range size. For most workloads this approach is an optimal solution which minimizes the overall hardware overhead while maximizing the coverage.

One unexpected trend that can clearly be seen on leela graph is that the WB frequency of 10% gives better coverage results compared to WB frequency of 100% (for the same WCHH size and range size = 4,16). Instinctively, higher WB frequency should result in more updated WCHHs with more accurate miss patterns and write count predictions. However, for this benchmark (also deepsjeng and x264) lower WB frequency parameter value results in higher coverage. One possible reason for this behaviour is repeating miss patterns where some miss addresses are constantly swapped. For example, if the history holder size is 1 and the miss patterns are AB AC AB AC, the algorithm will actually benefit more from not writing back C's write count in A' WCHH, since it will then successfully predict the next AB pattern (A's WCHH will hold B's write count). However, to confirm this we need to thoroughly analyze the address traces (write an algorithm to detect such behaviour).

Future work

For the future work there are three main parts that can be explored:

1. Write count prediction algorithm possible improvements:

● Introduce replacement policy within WCHH. Currently we flush the entire block's WCHH and update it with new updated write counts. We can introduce some modified LRU/LFU to only keep correctly predicted write counts (successfully used) and overwrite the rest. Smart replacement policy can help increase the correct predictions for workloads with moderately recurrent miss patterns. This can further improve the results obtained for linked list prefetching.

● Writeback policy for WBs due to WCHH. We can introduce a policy that will try to predict whether the future prediction for these blocks will benefit from the updated write counts in WCHH. This will help decrease the WB bandwidth without significantly affecting the overall coverage.

2. Possible optimizations for benchmark testing/experiments:

(Current slowdown for PIN with integrated WCP algorithm: 600x-800x)

● Restructure LRU cache. Instead of current cache model with C++ standard library linked-lists and unordered maps, we can try using simple 2D array cache model with clock bit approximate LRU replacement. This approach can result in possible speedup of 1.5-3x (based on other teams' experience), which will cut down current average experiment times from 10-20 hours to 5-10 hours.

● Under current implementation (algorithm integrated into Pintool) modifying the algorithm or changing any parameters forces us to rerun the entire experiment. Thus, instead of integrating our algorithm, we can integrate only the cache model to only store miss/evict address traces. With this method we only need to generate one large address trace file for each benchmark and use them for any future experiments.

3. More experiments and benchmark analysis:

● Test benchmarks with larger input data size (ref/large). Larger input data size will result in longer runs, which will better warmup our cache model, increase write count average, and thus, will better resemble real world applications. Results from such workloads will help us better analyse our design and understand any flaws and limitations.

● Test all possible permutations of parameters. Due to time constraints we were able to ran only a few permutations that we thought would produce some interesting and most promising results. Running more parameters variations will help us better analyse our design and find the most optimal parameters for different workloads.

Conclusion

We created a fully functional write count prediction algorithm that provides satisfactory results across all tested benchmarks. Even in the worst case scenario with lowest parameter values of WC history size = 1, prediction range = 1 and WB due to WCHH frequency = 0%, the minimum coverage is still 32% (mcf_s). We found that for most macro-benchmarks with medium input data size, the optimal parameters are WC history size = 4, range size = 16 and WB due to WCHH frequency = 10%. These parameters allow our algorithm to maximize the coverage, while minimizing both the hardware overhead from WCHH and bandwidth overhead from WB due to WCHH. The average coverage of all commercial benchmarks under these optimal parameters is 82%. Overall, we were able to complete all the tasks from our work plan: finished wc prediction algorithm (with some bandwidth optimizations), incorporated our implementation into Pintool for long runs, completed linked list prefetching algorithm, developed various microbenchmarks, tested our design with both micro-benchmarks and commercial macro-benchmarks.