Characterizing Wear Characteristics of Non-Volatile Memory Last Level Cache Replacement Policies

MEng CPEN 511 Final Project Report
Michel Kakulphimp #63542880
michelk@gmail.com

1 Introduction

Fast, non-volatile memory presents new opportunities for computer systems designers to exploit due to their unique characteristics, most important of which being their retention of data when power is no longer applied to a system. New non-volatile technologies are emerging from research, showing great promise in being able to replace cache memories that lie close to the processor.

Traditionally, these caches are built out of SRAM cells, which store a single bit of information in a bistable latching circuit. In modern semiconductor process technology, these circuits usually contain six transistors for a single bit cell. Memory made of SRAM has the advantage of being very fast, simple to implement, and reliable. For this cell to retain its data, a constant voltage must be supplied to the circuit which draws a non-trivial amount of current. This is what is known as leakage current. As the memory arrays containing these cells scale, so do their combined leakage currents. This static power consumption as well as the complexity of the SRAM cell plays a significant role in limiting the scalability of these technologies.

The use of fast, non-volatile memory in caches provides an opportunity at addressing the challenges of SRAM by introducing fundamentally smaller cell sizes, good scalability, and non-volatility [1]. Instead of storing a bit in a bistable latching circuit, non-volatile memories encode it in a physical property of a material. This property's state can be electrically set and determined, allowing one to store arbitrary data in an array of these non-volatile cells. Examples of these non-volatile cell types include using the resistance of a dielectric material (memristor) [1] in ReRAM, the state of a chalcogenide glass (crystal or amorphous) [2] in PCRAM, and the magnetic field orientation in a magnetic tunnel junction [3] in STT-RAM.

Non-volatile memories are not without their drawbacks. The fundamental physical properties that are being exploited for their data storage capabilities must be successfully miniaturized and be consistently reproducible within certain margins of characteristics and performance. Logic implementation alone is a challenging task for silicon lithography at a given process node and is made more complicated with the introduction of new materials and techniques required to create these new non-volatile cells. If a cell is hard to manufacture and/or has poor yields, there is little incentive in its implementation.

Another challenge with these non-volatile cells is their read/write asymmetry. With SRAM cells, a read and a write usually takes a single cycle. With non-volatile cells, a read will also usually take a single cycle, but a write may many more. This read/write asymmetry [4] is due to the time it takes to modify the physical property of the non-volatile cell.

What this project concerns itself with is non-volatile memory's challenges surrounding write endurance. Traditional memory technologies such as SRAM or DRAM have endurances in the order of more than 10^{15} writes [7]. This translates to perceptibly infinite write cycles that can be performed on a cell. Non-volatile memories, on the other hand, are not as fortunate with their endurance capabilities, largely due to the physical properties of the state changes that occur within the non-volatile cell encoding the bit. For the non-volatile memories mentioned thus far: ReRAM, PCRAM, and STT-RAM, their endurances are in the orders of 10^{11}, 10^{8}, and 10^{15} respectively [7]. Of note, STT-RAM's write endurance is theoretically the highest at 10^{15}, but it has not yet been tested to these levels according to [7]. Since write endurance remains a common challenge amongst most, if not all non-volatile memories, there is incentive to ensure that its implementation will withstand the lifetime of the device it is resident in.

As investigated in [4], this project will explore the use of non-volatile memory use in last-level caches. This project's focus is on the wear characteristics that these

caches would experience under different workloads, cache replacement policies, and hashing behaviours.

2 Implementation

Exploring this problem space required several different components to come together. First, a simulator capable of running arbitrary program data was required. It needed to be extensible and configurable to observe the cache array in detail. The usually invisible cache read and writes must be instrumented to determine if a bit cell's value changes or not. By having a way to measure how many bit flips each cell of a cache experiences, it is possible to infer the wearing characteristics a given cell will experience.

Another important piece that this project aims to analyze is the effect that a cache replacement policy has on the wear characteristics of a non-volatile cache. This requires that the simulator be able to switch between different algorithms in an easy manner.

Workloads are also required for analysing a cache's wear characteristics. Each workload will use the cache in a unique manner, and an instrumented cache should be able to reflect its effect on it through the number of bit flips that are observed. A way to analyze and visualize the cache wear characteristics of these workloads is also important to be able to understand the raw numbers that are obtained during testing.

In the following sections, discussion of the simulator chosen, *ZSim*, takes place and an overview of the various challenges encountered during its setup for the project. *ZSim*'s class extensions as well as the instrumentation performed to the simulator are outlined to get a better understanding of how data is gathered. Finally, details of the workloads used in the testing are provided as well as a discussion of the analysis methods used for the raw results.

2.1 ZSim Simulator

ZSim [8] is an architectural CPU simulator designed to be fast and accurate. It was originally designed to evaluate the compressed cache *ZCache* [9] (hence its namesake), *ZSim* has been repurposed as a microarchitectural simulator capable of emulating a variety of different microarchitectural components found in modern processors. It uses dynamic binary translation and instrumentation to execute programs natively on the simulator platform while being able to emulate a user-defined architecture. Most importantly, its design is highly extensible and provided a flexible and configurable underpinning for evaluating the goals of this project.

The first hurdle to overcome was the setup of the *ZSim* simulation environment. This was challenging for a number of reasons. First, *ZSim* requires the use of pin tool [10] version 2.x, the dynamic binary instrumentation required for the simulation. The API in pin tool 3.x was updated and is unsupported by the latest version of *ZSim*. The time required to understand the API changes between 2.x and 3.x version of the pin tool and subsequently refactoring *ZSim*'s calls into pin tool would have taken much longer than the project's length would have allowed.

Since this project was limited to using pin tool 2.x, it was also limited to running a version of Linux which uses the 3.x kernel. The latest 4.x kernel is only supported by pin tool 3.x which was found to be incompatible with *ZSim*, as explained above. The last version of Ubuntu that had a 3.x kernel was 14.04.3, which had the 3.19 kernel. This kernel version was compatible with pin tool 2.x and subsequently *ZSim*.

A single core system was initially simulated as it was the canned example used in the setup for *ZSim*. This simple configuration consisted of a *ZSim SimpleCore* with separate L1 instruction and data caches attached directly to it which were 8 kB and 4 kB respectively. These two L1 caches fed into a 2MB L2 cache which would be instrumented into our non-volatile last-level cache.

2.2 ZSim Instrumentation

Instrumenting *ZSim* to obtain the wear levelling information was a key focus for the majority of this project. The simulator provides a flexible hierarchy of classes of which many emulated devices are derived from.

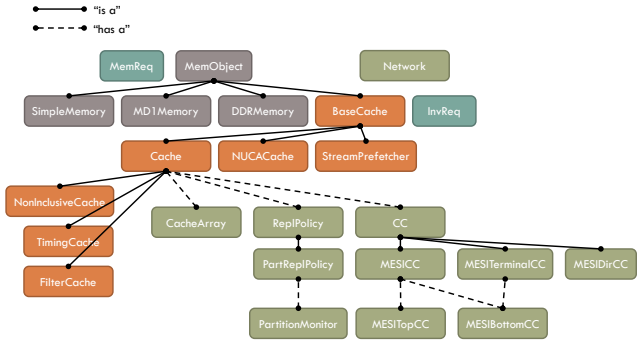

Figure 1 - ZSim Memory Classes Hierarchy [11]

For example,

Figure 1 gives an overview of the different memory classes available in *ZSim*. The entire simulator is a complex hierarchy of classes used to define the behaviours of the different microarchitectural components present in a modern processor.

The simulator requires a properly configured configuration file as an input when running the program. It sets a variety of different parameters which govern how the simulated system will be structured architecturally and how its components will behave. Broadly, the configuration file has settings for the CPU(s), its cache hierarchy, how each component behaves, simulator runtime settings, and finally the external process(es) to attach the simulator to via pintool.

Specific to this project is the section that deals with setting up the cache hierarchy of the system. An example configuration was provided which setup a system with an L1 cache for data, another L1 for instructions, and an L2 as the LLC. At first glance, there isn't much to see for the L2 cache as a lot of the defaults were being used and therefore did not require a specific configuration change. However, when the program is executed, a full configuration file is generated with all of the settings defined. Shown in Figure 2 is the full definition of the L2 cache.

```
l2 :
{
  children = "l1i|l1d";
  isPrefetcher = false;
  size = 2097152;
  banks = 1;
  caches = 1;
  type = "Simple";
  array :
  {
    ways = 4;
    type = "SetAssoc";
    hash = "None";
  };
  repl :
  {
    type = "Rand";
  };
  latency = 10;
  nonInclusiveHack = false;
};
```

Figure 2 - Configuration of the L2 Cache

This configuration sets up a cache of 2MB (2,097,152 bytes) in one bank split into four ways. It is a set-associative in its configuration and has a random replacement policy setup for it. It is connected to the L1 caches through the children parameter and is defined as a "Simple" *ZSim* cache. Instead of *Simple*, one can also specify a *Timing* or *Tracing* cache based on one's requirements. The hash option specifies the whether or not to use hashing when addressing the lines in a cache. This has the effect of scrambling the mapping of the cache in a pseudo-random pattern.

ZSim's configuration is interpreted in the body of its *InitSystem* function found in init.cpp. This function is responsible for parsing through the configuration file to obtain and validate the configuration information. Of interest to this project is the cache creation that takes place through respective calls to *BuildCacheGroup* and *BuildCacheBank*, where the former will call the latter. Inside the *BuildCacheBank* function, it checks what parameter was set for the *type*. This is where it was discovered that a different *CacheArray* object is selected based on the input from the configuration file.

The *CacheArray* class which was instrumented for specific use in this project. Originally, *ZCache* was implemented as an extension of this base class. The extended class for this project was named *NvmArray*. The interface for the *CacheArray* class defines the following functions for implementation:

```
int32_t lookup(const Address lineAddr, const MemReq* req,
bool updateReplacement)

uint32_t preinsert(const Address lineAddr, const MemReq* req,
Address* wbLineAddr)

void postinsert(const Address lineAddr, const MemReq* req,
uint32_t lineId)

void initStats(AggregateStat* parent) {}
```

The *lookup* function returns the tag ID, if it exists in the cache, of a specified line address. The hashing scheme will be used to access the cache array and the replacement policy's update function will be called on the line. If the tag ID doesn't exist, it signals this by return -1 to the caller.

The *preinsert* function runs the replacement scheme on the cache. It will return the tag ID of the new position and the address of the line to write back. This write back address is used by the subsequent function.

The *postinsert* function is guaranteed to be called after the *preinsert* function and performs the actual insertion of the new data in the cache.

Finally, the *initStats* function is used to initialize the built-in statistics functionality that is present in most *ZSim*

classes. It was originally intended to leverage *ZSim*'s statistics functionality as it provided a built-in mechanism to tabulate the data we would be producing from the instrumentation. Unfortunately, however, the data that would be generated from the instrumented simulator would prove to be too large for the statistics functionality to handle. Instead, a vector was stored in memory and written out to a file when the simulation finished. Future work would encompass upgrading the statistics functionality of *ZSim* in order to leverage existing logging functionality.

To gather our data, the *postinsert* function was instrumented in the project's *NvmArray* class. The general idea is that we want to count the number of times a bit in a cache line flips from 0 to 1 and from 1 to 0. By keeping a tally of how often this happens for each bit in a cache line, we can observe the wearing characteristics of each theoretical non-volatile memory cell. To perform this bit flip counter check, a 64-bit counter is stored in memory for every byte in the cache. Every time a bit flips in a byte, the counter is incremented by one. The state of the line before and after the insertion is XORed and the number of bits that remain are counted to give us an indication of how many bit flips occurred in the byte. Instead of iterating through each bit in a byte, which would have taken a long time for each cache line insertion, a 256 byte lookup table was defined in memory for a quick lookup of how many bits are set in a byte. For example, passing an index of 53 (0x35 in hexadecimal) to the lookup table, a result of 4 is obtained. For every combination possible in a byte, a value is set in the table. By incrementing the counter by the result of this table, we can quickly shift through the bytes of a cache line and determine how many bits have been flipped through the XOR check.

At first, no hashing was used in the simulator, which yielded some interesting results that are discussed in later sections. When we are not hashing, the memory address is shifted right by the number of line bits to associate with a given line. With hashing, this shifted line address is hashed and is placed in a resulting cache line based on the result. Hashing has a significant impact in cache usage which will also be discussed in later sections.

During project development, a file was generated containing a counter for every byte in the cache line to represent how many times a bit was flipped in a cache line's bytes. For example, for a 2MB cache, this generated a counter data file for each of the 2097152 bytes. This file is therefore sizeof(uint64_t) x 2097152 = 16 MB large, a modest size to capture our data. Originally, this file was opened, written to, and closed on every cache line insertion, which allowed results to be viewable as the simulator would progress. However, this came at a steep performance penalty. By incrementing the counter in memory and outputting the results to a file, this allowed the simulation to run a lot faster and yield results at a more reasonable pace (23 hours down to an hour).

The last piece that was required for this project was to ensure that *ZSim* would be able to emulate different cache replacement policies. For the purposes of this project three policies were tested. A random policy, which would evict a random victim line from a set in the cache. An LRU policy, where the least recently used line is victimized for replacement. And finally, an SRRIP [13] policy, where re-reference intervals are predicted to determine victim cache lines. SRRIP was not available as a built-in replacement policy to *ZSim* but it was later found and integrated into the simulator for testing.

2.3 PARSEC workloads

A workload must be provided to *ZSim* to exercise its virtual components. Unfortunately, the industry standard SPEC CPU benchmark is not freely available, requiring a pricey licensing fee. The SPEC benchmarks were obtained towards the end of this project, but insufficient time remained to get them working within the project's environment. This will be left for a future exercise. In its place, another benchmarking tool was found, called PARSEC [12]. This benchmarking suite contained a wide variety of multithreaded benchmarks which can stress systems in different ways. Similar to the struggles surrounding the installation of *ZSim*, PARSEC has to be modified in order to full compile and run on the system. The entire benchmarking suite was installed on the testbed PC and its constituent benchmarks were tested both natively and through the *ZSim* environment.

The first PARSEC benchmark that was used is the *Black-Scholes* workload. It is a financial modelling algorithm for the pricing of European style options. It calculates the prices for a given portfolio analytically with Black-Scholes partial differential equations. Since no closed-form expression exists for these, the results must be computed numerically. The largest dataset is approximately 600 MB, which means that the cache usage should be pronounced since we cannot fit its entire contents within it.

The second PARSEC benchmark that was used is the *raytrace* workload. This is a 3D rendering algorithm that casts light rays through a 3D scene and builds an image at the observer's camera by registering the resulting rays. This particular algorithm is said to be optimized for speed rather than realism, and its computational intensity depends on the resolution of the image. For this project, a 3D scene containing 400 MBs worth of 3D data was rendered at a resolution of 1920 x 1080. Again, none of this can fit into the cache, which should yield a lot of activity to observe after simulation has ended.

2.4 Parsing the Data

When the simulation ends, an output of 64-bit counters is obtained for every byte in the cache under test. In order to analyze this data, a parser was written in python to generate different plots of the data for easy analysis.

The first plot that is generated is a heatmap of the entire cache contents. The minimum and maximum values are determined from the data to provide the min and max of the colour gradient used in the heat map and every byte in the dataset is represented by a colour in the defined gradient.

Figure 3 shows an example of a heatmap generated from a dataset generated by the instrument *ZSim*.

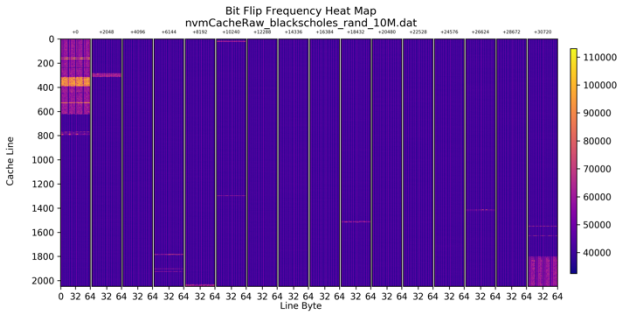

Figure 3 - Heat Map of Dataset

Since the cache is rather "long" as every line is only 64 bytes each, the cache is divided into 16 columns of equal height for easier viewing. The plot provides an offset at the top of every column which is used in conjunction with the ticks on the y-axis to get an idea of where one is situated in the cache for a given datapoint. This view was supplemented by a zoomed in plot which only shows a subset of the dataset for easier analysis. An example of this is shown in

Figure 4. This view makes it much easier to discern the bit flip patterns within a selection of cache lines. What was most interesting about this is that the different data types used in the program were clearly visible within the wear patterns as indicated by the heat of a given byte. The *black-scholes* dataset had chunks of 16-byte data scattered throughout the cache.

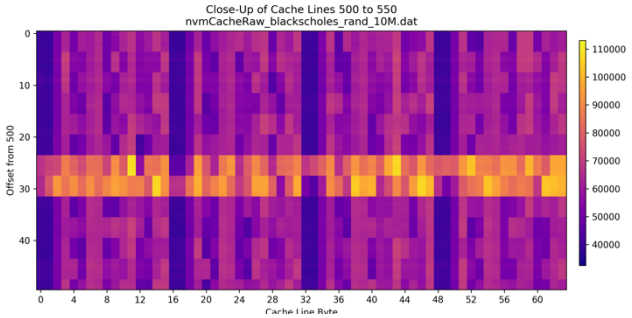

Figure 4 - Closeup View of Dataset

To supplement the visual heat maps, a scatter plot and a histogram were also generated of the same data, which can be seen in Figure 5 and Figure 6. The scatter plot allows one to get a quick index of more worn lines by looking at the peaks in the data.

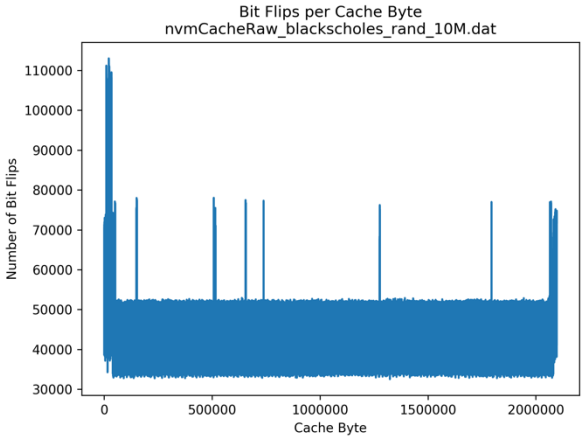

Figure 5 - Scatter Plot of Dataset

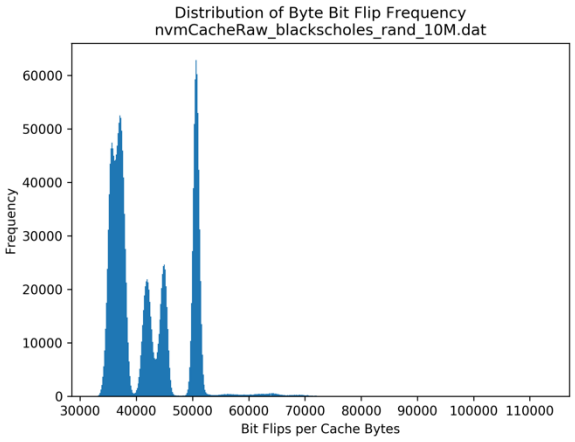

Figure 6 - Histogram of Dataset

The histogram provides a distribution of the different byte bit flip frequencies in the dataset. This is useful for identifying differences in the different wear levels that exist in a dataset. In an ideal case, we would see only a single peak, indicating that the cells are being worn at the same rate.

3 Evaluation

The following sections outline the results that have been obtained for the project thus far. After giving the project presentation, it was realized that hashing was not turned on for the first sets of data. This resulted in a cache whose mapping closely mapped the layout of the program and data in memory. With hashing turned on, different data was obtained which steered the conclusions away from replacement policy selection and more towards the hashing itself.

As mentioned previously, the replacement policies chosen for testing were random, LRU, and SRRIP. The system simulated for the non-hashed results were a single core processor with 8 kB of L1 instruction cache, 4 kB of L1 data cache, and 2 MB of L2 cache which is the last level cache of the system. The hashed results simulated a system with eight cores instead, each with the same number and size of L1 caches connected to the same L2 cache. The L2 cache was implemented using an *NvmArray* which is instrumented to count the number of bit flips for every byte in the array. Once simulation is complete, the results were parsed through the parser.

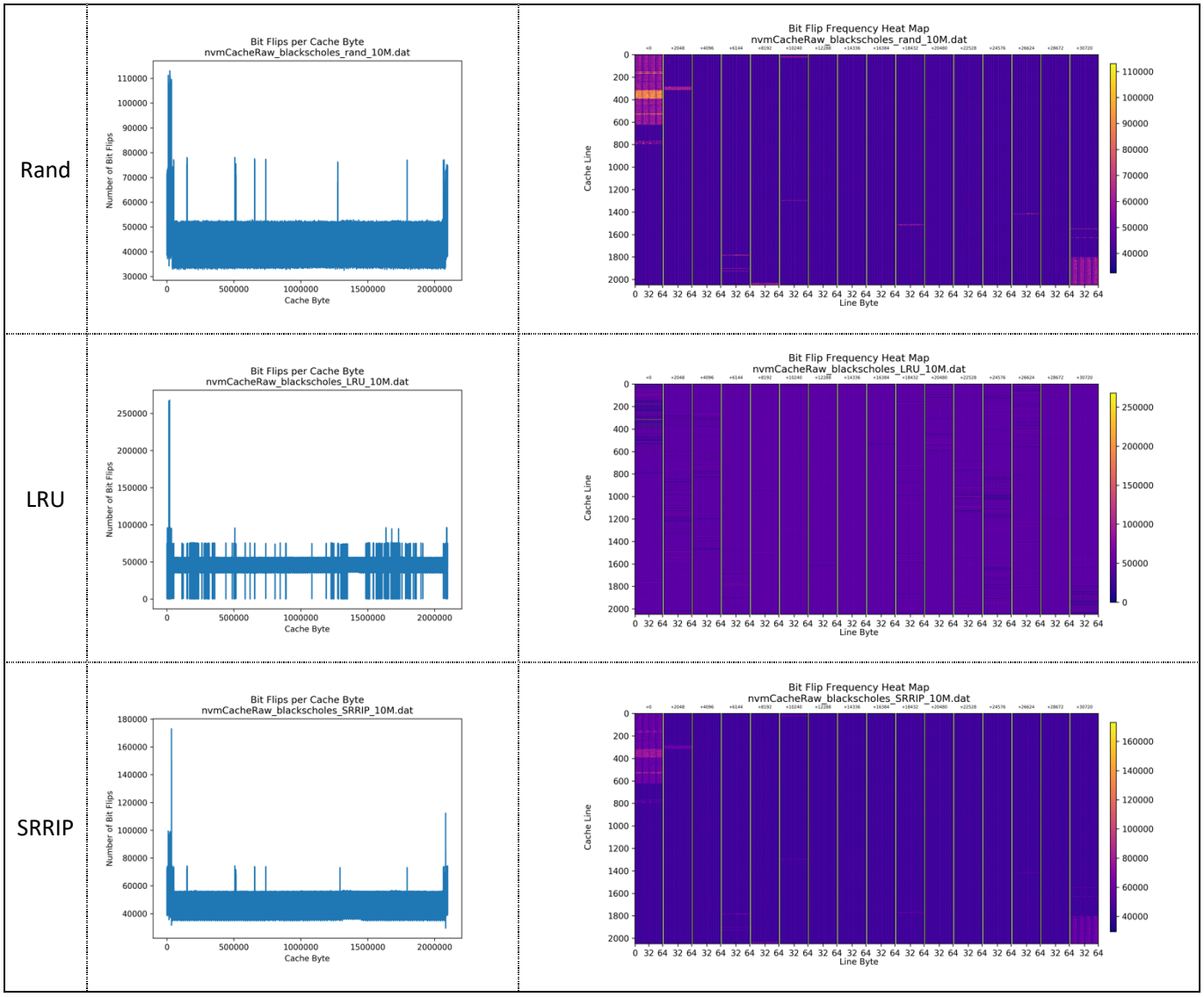

Figure 7 - Unhashed Black-Scholes Results

3.1 Unhashed Black-Scholes Results

Before hashing was implemented, unhashed results were obtained. The results are shown in

3.2 U

3.3 nhashed Raytrace Results

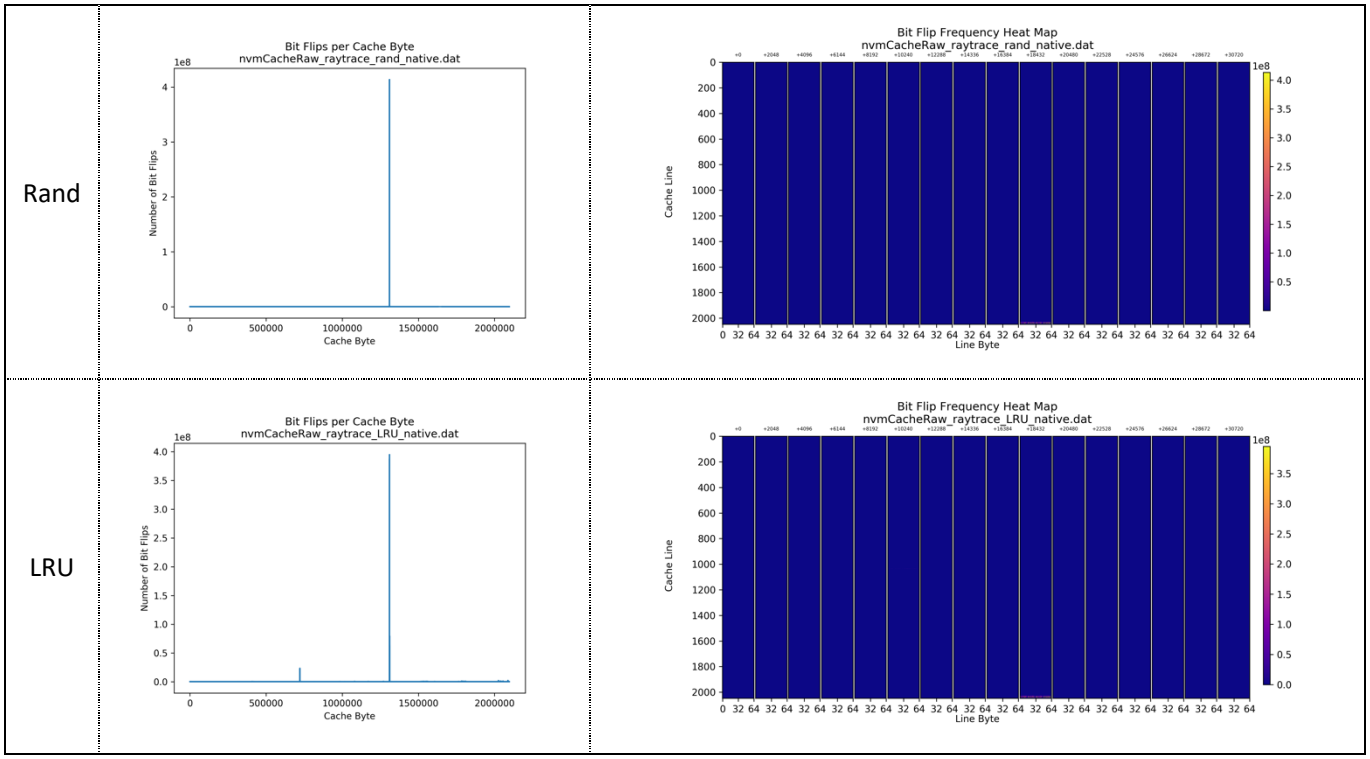

Figure 8 - Unhashed Raytrace Results

3.4 Hashed Black-Scholes Results

3.5 Hashed Raytrace Results

4 Related Work

- Cache bypassing methods for dealing with read/write asymmetry in [5]
- Design of LLC using STT-RAM in [4]
- Relaxing non-volatility in NVM caches [6]
- Improving NVM cache lifetime through write restriction [7]

5 Conclusions and Future Work

- Using ZSim's built-in functionality for data gathering
- Using SPEC instead of PARSEC
- Simulating a real system instead of using the built-in example

6 References

[1] Chen, A. (2016). A review of emerging non-volatile memory (NVM) technologies and applications. Solid-State Electronics, 125, 25–38. https://doi.org/10.1016/j.sse.2016.07.006

[2] Chien, T. K., Chiou, L. Y., Tsou, Y. S., Sheu, S. S., Wang, P. H., Tsai, M. J., & Wu, C. I. (2017). Write-energy-saving ReRAM-based nonvolatile SRAM with redundant bit-write-aware controller for last-level caches. *Proceedings of the International Symposium on Low Power Electronics and Design*, 2–7. https://doi.org/10.1109/ISLPED.2017.8009153

[3] Qureshi, M. K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L., & Abali, B. (2009). Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*. https://doi.org/10.1145/1669112.1669117

[4] Xu, W., Sun, H., Wang, X., Chen, Y., & Zhang, T. (2011). Design of last-level on-chip cache using spin-torque transfer RAM (STT RAM). *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *19*(3),

483–493. https://doi.org/10.1109/TVLSI.2009.2035509

[5] Sun, G., Zhang, C., Li, P., Wang, T., & Chen, Y. (2016). Statistical Cache Bypassing for Non-Volatile Memory. *IEEE Transactions on Computers*, *65*(11), 3427–3440. https://doi.org/10.1109/TC.2016.2529621

[6] Smullen IV, C. W., Mohan, V., Nigam, A., Gurumurthi, S., & Stan, M. R. (2011). Relaxing non-volatility for fast and energy-efficient STT-RAM caches. *Proceedings - International Symposium on High-Performance Computer Architecture*, (June 2014), 50–61. https://doi.org/10.1109/HPCA.2011.5749716

[7] Agarwal, S., & Kapoor, H. K. (2019). Improving the Lifetime of Non-Volatile Cache by Write Restriction. *IEEE Transactions on Computers*, *PP*(8), 1–1. https://doi.org/10.1109/TC.2019.2892424

[8] Sanchez, D., & Kozyrakis, C. (2013). ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. Isca'13, 475–486. https://doi.org/10.1145/2485922.2485963

[9] Sanchez, D., & Kozyrakis, C. (2010). The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 187–198). Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/MICRO.2010.20

[10] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., … Hazelwood, K. (2005). Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 190–200). New York, NY, USA: ACM. https://doi.org/10.1145/1065010.1065034

[11] Sanchez, D., & Beckmann, N. (2015). ZSim MICRO-48 Half-Day Tutorial - MIT. Retrieved January 30, 2019, from http://ZSim.csail.mit.edu/tutorial/

[12] Bienia, C. (2011). Benchmarking Modern Multiprocessors, 153.

[13] Jaleel, A., Theobald, K. B., Steely, S. C., & Emer, J. (2010). High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News*, *38*(3), 60. https://doi.org/10.1145/1816038.1815971

[14] Qureshi, M. K. (2018). CEASER: Mitigating conflict-based cache attacks via encrypted-Address and remapping. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, *2018–October*, 775–787. https://doi.org/10.1109/MICRO.2018.00068