# NC State University

## Department of Electrical and Computer Engineering

## ECE 463/563: Fall 2021 (Rotenberg)

## Project #1: Cache Design, Memory Hierarchy Design

by

STEVAN DUPOR

NCSU Honor Pledge: "I have neither given nor received unauthorized aid on this project."
Student's electronic signature: Stevan M Dupor
Course Number: 563

# Introduction and Methods

Cache design and performance simulation are critical components in the workflow of designing complete computer architecture. For this project, a flexible cache simulator was developed which accepts memory traces (flatfiles detailing serial memory accesses) and simulates the behavior of one or more levels of hardware cache as well as main memory.

The simulator maintains the state/contents of all caches and keeps track of statistics on hit/miss rate, evictions, writebacks, and swaps. These statistics can then be cross-referenced to hardware design simulations of latency, chip area, and energy to make performance tradeoff decisions of a certain cache hierarchy. The simulator is flexible, allowing multiple levels of caches of arbitrary size as well as optional victim cache at the L1. For this simulator, all caches share the same Block size for each simulation run, but the block size may be adjusted between runs.

The simulator is implemented in C++ (2011 standard), and takes significant advantage of object-oriented programming to simulate the hardware layout in software. Cache levels use identical interfaces and internal pointers to handle data accesses to lower levels; eg, the CPU will only ever ask the L1 cache for a byte of data, and the cache hierarchy will handle all of the transactions below this surface interface. Significant recursion is used between levels such that each level owns its own operations, and only speaks to other levels through the well-defined read/write interface. The main memory module is a limited instance of the Cache class, with infinite size and associativity (always hits). As much as possible, the hierarchy models a hardware design with one exception: when printing the cache contents at the end of a run, each Set structure is copied and destructively sorted by recency, to simplify the reporting process.

Numerous experiments across varying cache parameters were conducted, and plots of these results as well as discussion of the findings follow in proceeding sections. The experiment configuration/automation and data collection via comma-separated-value file were performed using C++ (code excluded from submission for convenience). The data analysis and plots were implemented in Python – code also excluded from submission for convenience.

# L1 Cache Exploration: SIZE and ASSOC

The first series of experiments concern varying the size and associativity of a cache hierarchy consisting of only a Level-1 cache and a main memory.

## Graph #1 L1 Miss Rate vs Cache Size

55 simulation runs were performed for eleven different cache sizes (1KiB to 1MiB) across five associativities: Direct Mapped, 2,4, and 8-Way set associative, and fully associative caches. The miss rates for each run as compared to cache size are displayed in Figure 1. From this plot, it is clear that larger caches result in lower

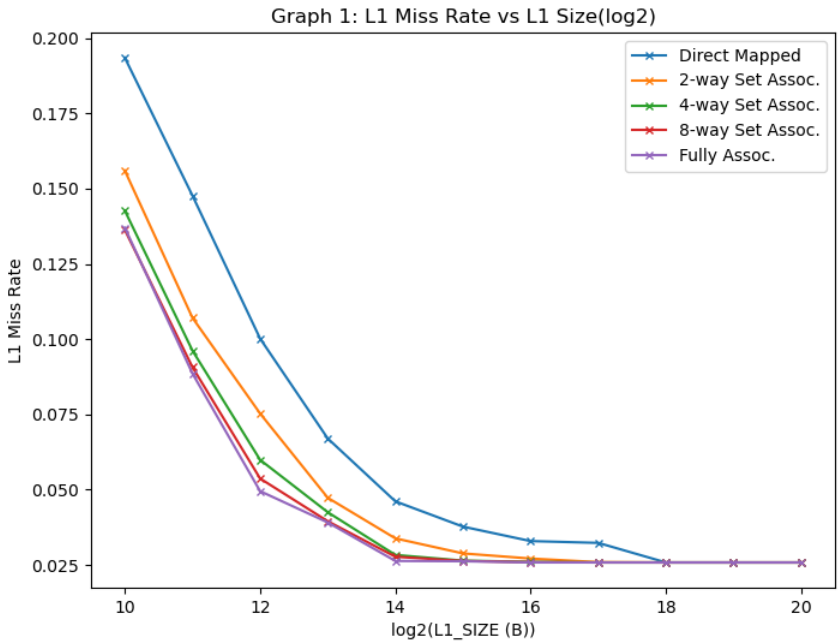


Figure 1: Miss Rate vs Size

miss rates, as the miss rate falls as cache size grows. Notably, when observing a single **given associativity**, as we increase cache size the miss rate tends to fall quickly at lower sizes (eg, 1KiB to 2KiB), then more slowly as we approach larger caches. This trend approaches an asymptote as where all conflict and capacity misses are resolved, leaving only compulsory misses. Also, it's clear that for a **given cache size**, as we increase associativity (particularly for smaller caches) the miss rate falls thanks to the resolution of conflict misses.

Based on the asymptote which all cache configurations approach as their size grows, we can estimate that the **compulsory miss rate**, or misses caused by an address having never been accessed prior, is approximatley **0.0258** or 2.58% of all misses.

Furthermore, we can estimate the **conflict miss rate** by examining the difference at a given size between a **fully-associative cache (zero conflict misses**, only compulsory/capacity) and the other cache associativities. For the purpose of this analysis, we will examine the change of associativity from the raw data across the lowest capacity simulated, in Table 1.

| Associativity | Conflict Miss Rate | Percent Misses attributed to conflicts |
|---|---|---|
| Fully Assoc. | 0.0 | 0% |
| 8-Way Set Assoc. | 0.001867 | 0.19% |
| 4-Way Set Assoc. | 0.0079 | 0.79% |
| 2-Way Set Assoc. | 0.0211 | 2.11% |
| Direct Mapped | 0.0554 | 5.54% |

Table 1: Conflict Miss Rate Comparison vs Associativity in an L1-Only Hierarchy

## Graph #2 Average Access Time (AAT) vs Size of an L1 Cache

Next, the same raw dataset was processed using Python and provided CACTI hit-time data to estimate the average access time for the provided trace for each cache configuration. These data are plotted in Figure 2, below. Note that the plot omits a datapoint for the AAT of a 1024KiB 8-way set-associative cache because this access time datapoint was unavailable from the CACTI dataset.

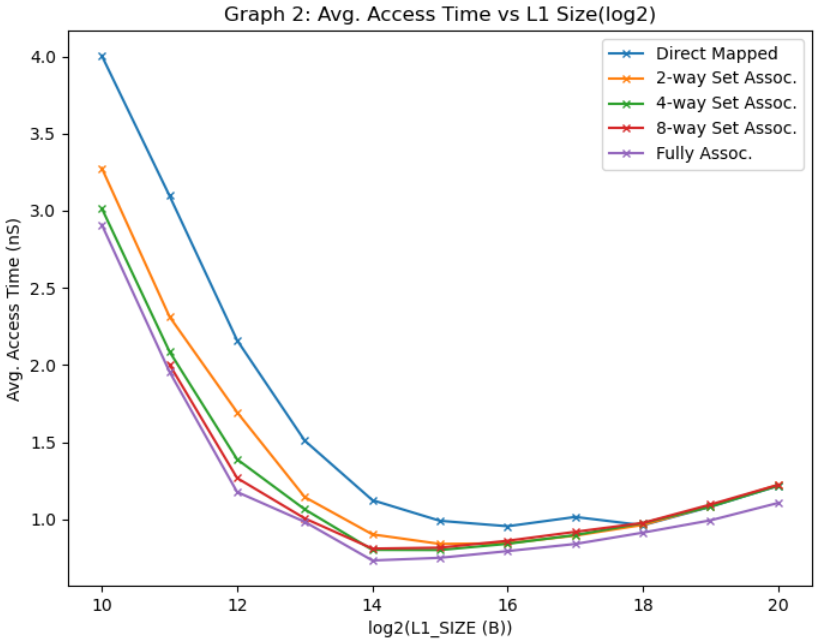


Figure 2: Avg. Access Time vs Size in an L1-Only Hierarchy

Immediately, a clear take-away from this plot is that larger cache size does not necessarily equate to improved performance. As cache size grows beyond 64KiB, for all configurations, we note a worsening in average access time. This is due to the fact that larger caches require more time for each hit, which is a compulsory latency. As this latency grows, the improvements from reduced misses no longer offset the latency of slower lookups in a larger cache, and performance falls. Notably, we can estimate from the plot that of these memory hierarchies with only an L1 cache and 32B blocksize, **the hierarchy with the best AAT is the fully-associative L1 cache of size 16KiB**.

## Graph #3 AAT vs Size of an L1+L2 Hierarchy

For this experiment, an L2 cache was added to the hierarchy of size 512KiB and 8-way set associativity. The L1 cache size was varied from 1KiB to 256KiB, and across the same range of associativities as in prior experiments. The results of these simulations and estimated AATs are displayed in Figure 3.

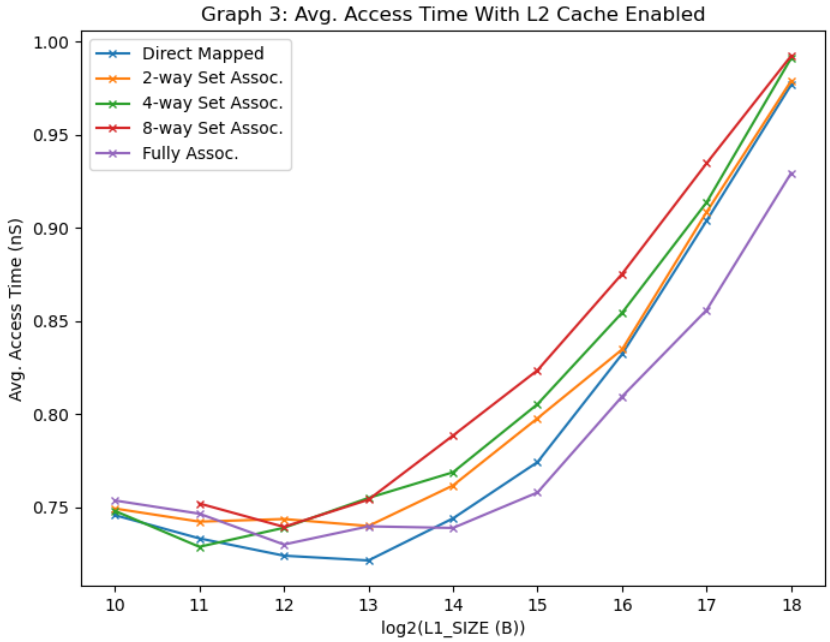


Figure 3: Avg. Access Time vs Size of an L1+L2 Hierarchy

Clearly, the addition of the level-2 cache to the hierarchy significantly improves the performance at small L1-cache sizes. This is due to reducing penalty for L1-misses at small sizes – even though we still have numerous L1-misses, these misses often require simply accessing the on-chip L2 cache, rather than needing to go off-chip to the main memory.

*Of particular importance when analyzing Figure 3 is to note the scale of the AAT range: The best performers have an AAT of approximately 0.70ns, and the worst, close to 1ns. Notably, in Figure 2, the L1-only hierarchy, the best performer has an AAT of 0.73ns, and the worst performer, 4ns, or quadruple the worst performance of the L1-L2 hierarchy*

Inspection of the raw AAT data confirms that the smaller L1-sizes when coupled with the L2 tend to perform most closely to the observed best-performer for the L1-only architecture. Notably, ALL associativites for ALL sizes between 1KiB and 8KiB, the Direct, 2-way, 4-way, and fully-assoc. 16KiB, and the fully-associative 32KiB all perform within ±5% of the AAT of the best L1-Only performer.

**The simulations where AAT was within 5% of the best-case for L1-only are enumerated below.**

- 1KiB L1 - Direct, 2-way SA, 4-Way SA, 8-Way SA, Fully Associative
- 2KiB L1 - Direct, 2-way SA, 4-Way SA, 8-Way SA, Fully Associative
- 4KiB L1 - Direct, 2-way SA, 4-Way SA, 8-Way SA, Fully Associative
- 8KiB L1 - Direct, 2-way SA, 4-Way SA, 8-Way SA, Fully Associative
- 16KiB L1 - Direct, 2-way SA, 4-Way SA, Fully Associative
- 32KiB L1 - Fully Associative

Furthermore, it's clear from this plot that the **8192KiB Direct-Mapped L1** has the best AAT when paired with the L2 cache. From analysis of the raw data, I find that this configuration yields an AAT 0f 0.72159ns, which is **1.7% faster** than the best-performing L1-only configuration, with AAT 0.73424ns.

However, there is a notable physical trade-off. The best-performing L1 configuration, a 16KiB Fully-Associative cache, requires an **on-chip area of 0.063446019** $mm^2$, whereas the best-performing L1+L2 configuration requires **2.693435312** $mm^2$ **of chip space**.

# L1 cache exploration: Size and Blocksize

The following experiment explores the affect of blocksize on cache performance for an L1-only architecture.

## Graph #4 L1 Miss Rate vs Blocksize

A series of simulation runs were performed across varying blocksizes from 16 to 128B. At the same time, the L1 cache size was varied from 1KiB to 32KiB. The resulting plot of miss rates from these simulations is displayed in Figure 4. When considering the effects of blocksize, it's important to consider the data to be

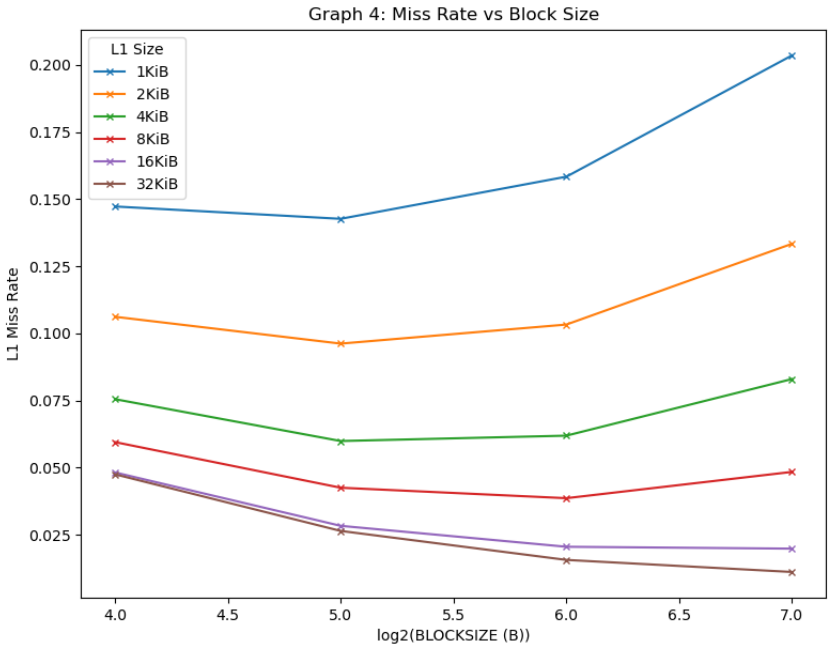


Figure 4: Miss Rate vs. Blocksize in an L1-Only Hierarchy

stored: for an architecture using 32-bit words, a 16-byte cache block can hold four words at a time. When we double the blocksize, we double the number of words that are brought in at a single time. This permits taking advantage of more spatial locality, eg, if we use (this) memory word, we are likely to use the (next) word as well. However, it's clear in Figure 4 that as block size grows beyond a threshold, the **gains from spatial locality** are overtaken by the **losses from increased conflict misses, also called cache pollution**. In other words, with larger blocks, we are bringing in more words than we actually use, on average, which leads to more misses/evictions.

There are also clear differences between how smaller and larger cache sizes are able to take advantage of different block sizes. Notably, the **smallest cache prefers smaller block sizes**, with an almost-flat performance increase between 16B and 32B, and an increasingly parabolic performance loss as block size grows beyond this threshold. Alternatively, the **largest cache sizes prefer larger blocks**, having more space to avoid conflict misses, and suffer the least from cache pollution. Notably, the largest cache continues to show performance improvement all the way to the 128B blocksize. Clearly, based on Figure 4, **given more available cache space** to accommodate larger blocks, and offset the conflict misses of cache pollution, the **tradeoff shifts in favor of larger blocks**.

# L1+L2 Co-Exploration

For this experiment, a hierarchy with an L1 and L2 cache of fixed block sizes and associativity is used.

## Graph #5 AAT vs L2 Cache Size

A series of simulations were run which varied the L1 Cache size from 1KiB to 256kIB, while varying the L2 cache size from 32KiB to 1MiB. Notably, in keeping with the design strategy where an L1 cache should always be smaller than an L2 cache, datapoints where the L1 cache would be equal to or larger than the L2 cache have been omitted. The results of these simulations are displayed in Figure 5. Immediately, I

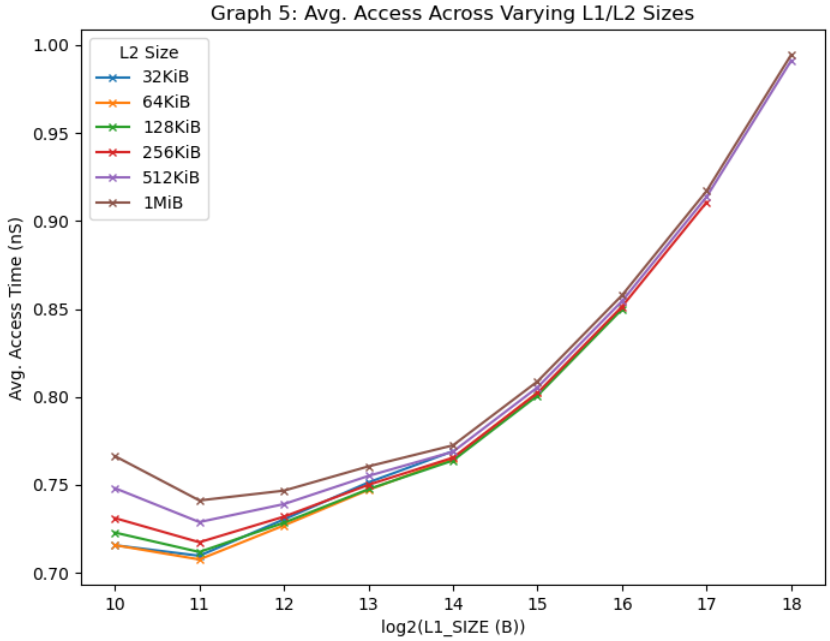


Figure 5: AAT vs. L1 size in an L1+L2 Hierarchy

note some similarities to Figure 3, where a similar hierarchy was studied with respect to associativity. Once again, the range of performance is more narrow than in an L1-only configuration, with AATs ranging from 0.708ns to 0.994ns. Notably, a clear trend in Figure 5 indicates that preference will be given to smaller L1 caches regardless of L2 size – this is likely due to the need at the L1 for absolute-minimum hit time. From analysis of the raw data, I find the fastest configuration with respect to AAT to be the **2KiB L1 + 64KiB L2 configuration**, at 0.707657833ns.

Furthermore, I find that fourteen configurations yield an AAT within 5% of the best AAT. Among these, with respect to chip area, the smallest configuration, or the **1KiB L1 + 32KiB L2 cache present the lowest total area, at 0.257285583** $mm^2$. Notably, this configuration performs only 1.144% slower than the fastest performer, which requires an on-chip area of 0.37897997$mm^2$, or 67.9% larger. This difference in size, when compared to the difference in performance, is quite significant.

# Victim Cache Study

For this series of experiments, a victim cache was added at the L1 cache. The purpose of the victim cache is to "catch" evicted blocks from the main L1 cache, in the anticipation of actually needing a recently-evicted block again in the near future. If an evicted block is requested and exists in the victim cache, it is swapped with the oldest block in the main L1 cache; if unavailable, the oldest block in the victim cache is written back if applicable, replaced with the oldest block in the L1 cache, and the requested block is brought in from lower levels.

## Graph #6 AAT vs L1 Size for an L1+VC+L2 Hierarchy

A series of simulation runs were performed across seven configuration scenarios: a Direct-mapped L1 cache with no Victim Cache, a Direct-mapped L1 cache with 2-entry Victim Cache, a Direct-mapped L1 cache with 4-entry Victim Cache, a Direct-mapped L1 cache with 8-entry Victim Cache, a Direct-mapped L1 cache with 16-entry Victim Cache,a 2-way set-associativL1 cache with no Victim Cache, and a 4-way set-associative L1 cache with no Victim Cache. For each scenario, simulations were run across a range of L1 sizes from 1KiB to 32KiB. The average access time was computed based on the results of these simulations. The results of these simulations are displayed in Figure 6. In Figure 6, I notice two distinct trends: the direct mapped

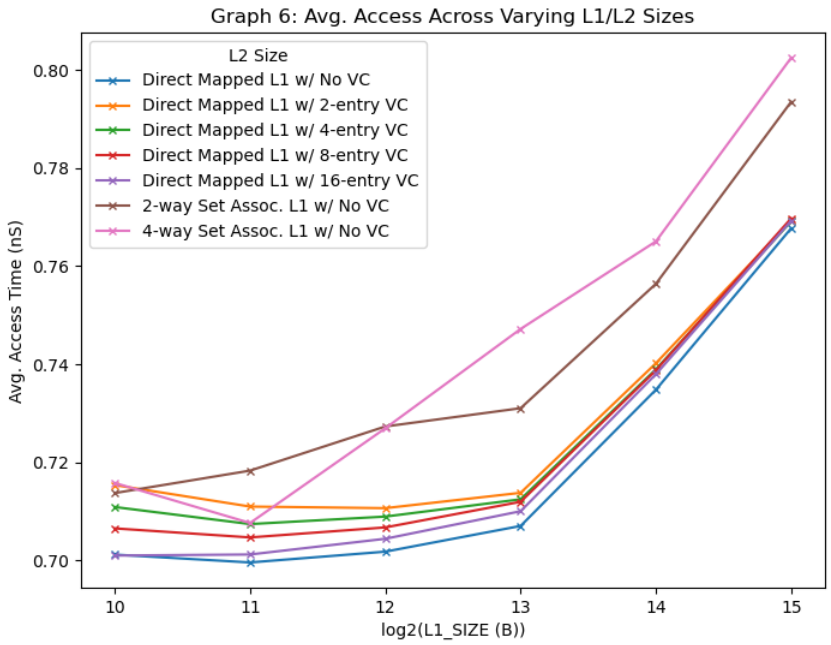


Figure 6: AAT vs. L1 size in an L1+VC+L2 Hierarchy

caches regardless of victim cache tend to perform similarly, especially at large size – which is to be expected when larger size reduces conflict misses, which victim caches are intended to combat. The two outlying trends represent the 2-way and 4-way set associative L1 caches with no VC implementation. Generally, the direct mapped L1 caches mostly outperform the fully-associative L1 caches regardless of presence of victim cache. Most interestingly, the direct-mapped cache with no victim cache outperforms all but one of the direct-mapped caches with victim caches.

The **1KiB direct-mapped 16-entry VC slightly beats the 1KiB direct mapped with no VC**, performing only 0.19 picoseconds faster. The **key reason** for VC-enabled caches tending not to beat the simpler direct-mapped with L2 configuration is in the relatively-high hit time as compared to the L1-alone hit time: a 1KiB direct mapped cache alone has a hit time of 0.114797ns, and the 2-block VC has a hit time of 0.131305ns. When this mandatory hit time is added to L1 misses, it tends to overshadow any performance gains from reduced L2 accesses – furthermore, since this victim cache can only accommodate 2 victim blocks, the likelihood of a conflict or swap failure within the VC is quite high. For this reason, it's a common strategy to treat another level of cache as the "victim cache" for a direct mapped cache to take advantage of reduced miss rate without introducing unnecessary mandatory hit time within the L1 in case of misses. A victim cache as added to a direct mapped cache **does present a performance improvement over a 2-way set associative cache of the same size**, but the improvement is still worse than a simple direct-mapped

cache. Notably, for **the 2KiB 4-way set-associative cache does outperform the direct-mapped with 2-entry VC cache**. However, as the **number of VC blocks grows,** the VC-enabled direct mapped cache again outperforms the 4-way set associative cache.

From the raw data, the hierarchy with the **best AAT performance is the 2KiB direct-mapped cache with no VC,** with an AAT of 0.699615664ns. Interestingly, 27 of these configurations yield an AAT within 5% of the best AAT; however, with respect to chip area, the **2-way Set Associative 1KiB L1 cache with no VC is the smallest** of these by a small margin. This cache is 0.000826734 $mm^2$ smaller than the smallest direct-mapped L1 with no VC, which also performs within 5% of the best AAT.

## Conclusions and Roadmap

Clearly, cache simulation is a key step towards designing a cache hierarchy which performs well in a computer architecture, and also meets required tradeoffs such as area and energy. Several unexpected results were encountered as part of these experiments, in particular with respect to victim caches. Conceptually, one would expect a victim cache, in helping to reduce reads to L2, to significantly improve the performance of a direct mapped cache but in many cases, this was not found to be true.

There are several options for interesting improvements and extensions to the capabilities of this simulator: the first, is extending the simulator to permit L3 and L4 cache simulations. This task is relatively accessible as the simulator was designed with flexibility in mind, and adding additional levels will only require modification of the initialization step and reporting steps. Another future extension of interest will be adding data-handling support, such that this simulator can be incorporated into a fully-featured architecture simulator that supports loading/storing data and running instructions on these data. This theoretical future simulator will permit complete end-to-end simulations of a fully functional, virtual, software-implemented computer architecture.