Statistical DRAM Modeling

Shang Li
shangli@umd.edu
University of Maryland, College Park

Bruce Jacob
blj@umd.edu
University of Maryland, College Park

ABSTRACT

Cycle-accurate DRAM models are prevalent in today's computer architecture simulations. However, cycle-accurate models by design are time consuming and not scalable. In this paper, we present a statistical approach of DRAM latency modeling. Unlike previous works, our approach converts DRAM latency modeling into a classification problem and employ machine learning models such as decision tree and random forest to solve the classification problem. We propose 4 basic DRAM latency classes to simplify and parameterize the classification, and extract features that help classification from memory request streams on the fly. We use synthetic traces to train the statistical model and test the model on real-world benchmarks in both accuracy and speed against a cycle-accurate simulator. The results show our statistical models improves the DRAM simulations speed by up to 400 times with 98% average classification accuracy for all the benchmarks we have tested.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**; *Massively parallel and high-performance simulations*; *Simulation evaluation.*

KEYWORDS

DRAM Modeling, Cycle Accurate Simulation, Architecture Simulation

ACM Reference Format:
Shang Li and Bruce Jacob. 2019. Statistical DRAM Modeling. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357526.3357576

1 INTRODUCTION

Architecture simulation is a vital method for designing and evaluating computer systems. Memory system simulation, especially cycle-accurate memory system simulation, is an important part of an accurate architecture simulation framework as it provides fine grained information about memory systems. However, the cycle-accurate model by design is time-consuming, and it limits the scalability of the simulation frameworks as the targeted systems nowadays are growing much larger in scale.

Modern architecture simulators, especially CPU simulators, adopts a variety of techniques to address the scalability problem. SST, Graphite [12] and Gem5[1] Timing CPU Model employs One-IPC model, meaning that every instruction is one cycle in the pipeline. Sniper[3] and ZSim[17] use abstract models for IPC which allows them to simulate out-of-order pipelines with relatively fast speed. In addition to speeding up simulation, another benefit of using abstract model is that CPU cores and caches can be simulated in parallel, allowing multi-core even many-core systems to be simulated efficiently.

Following the suit of abstract CPU models, we propose an abstract, statistical DRAM model to boost the DRAM timing simulation speed at the cost of modest inaccuracy. The statistical model is based on the fact that while there can be tens or even hundreds of unique latency values of DRAM requests, most of them can be neatly classified into only a few latency categories. We prototype a statistical model utilizing decision tree and random forest as classifiers, and test the model's accuracy and performance with real-world benchmarks. The benchmark results show our prototype statistical model achieves the goal of speeding up DRAM timing simulation significantly while retaining much of the accuracy of a cycle-accurate model.

The main contributions of this work can be summarized as follows:

- We propose turning DRAM timing modeling into a classification problem. The latency classes we propose are generic and apply to almost all commodity DRAMs.
- We exploit the temporal and spatial locality within memory request streams and extract high quality features from these localities that help the training and inference flow.
- Our flow of model training and feature extraction is highly parameterized and generic, meaning that the training only needs to be done once, and the model will work with all kinds of workloads. It also means applying different DRAM profiles to our model is made trivial.
- Our evaluation shows the statistical models are 5 to 400 times faster than state-of-the-art cycle-accurate DRAM simulator, with a 0.98 average classification accuracy across all the workloads.
- Our model is able to provide an atomic interface, meaning that the latency of a memory request can be returned by our model immediately upon inquiry. This enables the statistical model to be integrated in parallel, scalable simulation frameworks without introducing much synchronization overhead.

2 RELATED WORK

Before cycle-accurate simulators were widely adopted, researchers applied simplistic models for DRAM simulations. For example, the fixed-latency model assumes all DRAM requests take the same amount of time to finish, which completely ignores scheduling and

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357576

queuing contentions that may cause significantly longer latency. There are also queued models that account for the queuing delay, but they fails to comply with various DRAM timing constraints. Previous studies such as [18] have shown that such simplistic models suffer from low accuracy compared to cycle-accurate DRAM models.

Then came along cycle accurate DRAM models, such as [4, 7, 9, 16, 20] and DRAMsim3. These cycle-accurate DRAM simulators improved DRAM simulation accuracy, some are also validated by hardware models, but as we have shown, they started to lag the simulation performance.

Other than cycle-accurate models, there are also event based models such as[6, 8]. Event based models do not strictly enforce DRAM timing constraints, and can accelerate the simulation if the events are not frequent. But just as [8] pointed out, when memory workloads gets more intensive, memory events will be as frequent as every cycle, and therefore will undermine the advantage of the event-based approach.

Finally, there are analytic or statistical DRAM models such as [5, 21]. [5] presents a DRAM timing parameter analysis but does not provide a simulation model. The model in [21] provides predictions on DRAM efficiency instead of per-access timing information. [19] built a decision tree that classifies memory requests into conditional probability distributions but still needs a different scheduling algorithm to help producing the memory latency. Our work differentiates these previous work by treating DRAM timing modeling as a classification problem and classify each memory request into pre-defined latency classes instead of generating a conditional probability distribution. The latency classes comes from the understanding of how DRAM timing works instead of pure observation. We also do not treat memory controller as a black box, but instead extract high quality features from request streams that help the machine learning model "understands" the targeted controller model.

3 PROPOSITIONS

Different from analytic models that provide a high level analysis, which we discussed in Section 2, the statistical models here mean to provide an on-the-fly DRAM timing for each request based on a "trained" statistical or machine learning model.

The foundation of why such a statistical model would work on DRAM is that:

- DRAM banks only have a finite number of states.
- The timing of each DRAM request has already been largely dictated by the DRAM states when it arrives at the controller.
- Our observation shows most DRAM request latencies fall into a very few latency buckets, indicating that this behavior is likely the result of the previous two points.

And we will expand each of the claims one by one as follows.

DRAM banks only have a finite number of states: a DRAM bank can be modeled as a state machine: it can be in idle, open, refreshing, or low power states. Although there are typically thousands of rows that can be opened or closed, what matters to a specific request to a bank is whether the row of that request is open or not, so it will reduce to 2 states in this regard. Similarly, while there can be multiple banks in a rank and even multiple ranks in a channel, but for each request there is only a subset of these states that really matter to the timing of that request. Also, the queuing status when a new request arrives can also be accounted as states.

The timing of each DRAM request has already been largely dictated by the DRAM states when it arrives at the controller: intuitively speaking, when a request arrives at the DRAM controller, there are very limited actions that the controller can take. It can either A) process this request, whether because it is prioritized by the scheduler, or just because there are no other requests to be processed at the time, or B) hold the request whether because there is contention, other events are happening such as the current rank/bank is refreshing. Most of the scenarios here can be represented as a "state" like we previously discussed.

Our observation shows most DRAM request latencies fall into very few latency buckets, meaning that they are likely to be predictable: we ran cycle-accurate simulations on a set of 12 real world benchmarks, and discovered that although every benchmark has a long tail latency that stretches to over 400 cycles, (likely the result of having to wait for a refresh which is 420 cycles in this case), the latency of more than 90% of the requests are limited to just quite a few latency buckets.

This distribution fits into a statistical or machine learning model very well: the majority of the cases are predictable while the corner cases are there to optimize. With a statistical or machine learning model, we cannot handle 100% of the requests accurately like cycle accurate simulator. However, if we can accurately predict, for instance, 90% of the requests at the cost of a fraction of simulation time, then the trade-off may be worth the accuracy loss in scenarios where simulation speed is a limiting factor.

4 PROPOSED MODEL

4.1 Classification

It is clear now that the latency distribution for most memory requests is concentrated in a very small range. But there can still be tens of numeric values in that small range. These numeric values create noises to prevent the model from converging. For example, some requests have the latency of 20 cycles, which is exactly the minimum cycles it takes to complete a row buffer hit. But requests of 21, 22, 23, and all the way to 30 cycles also represents row buffer hit conditions, because if it is not a row buffer hit, then the minimum latency will be 20 + $tRCD$, which is well over 30 cycles. All the variations of 20+ cycles are caused by reasons such as bus contention, or rank switching, but they are still essentially row buffer hits, and therefore they should all be classified as one category instead of 10 individual numbers.

As we stated in Section 3, the dominating factor of the latency of a memory request is the DRAM states. For instance, a row buffer hit results in 20 cycles latency; a request to an idled bank takes 35 cycles; a row buffer miss takes 50 cycles; a request blocked by refresh operations can take 400 cycles. These are far more influential than one or two cycles of bus contention. Plus, these smaller numbers are very specific to the DRAM protocol and are thus not portable/universal. Therefore we propose to classify requests into these collective categories as opposed to individual values.

Based on how DRAM works, we propose the following latency classes and their corresponding latency number in DRAM timing parameters:

- **idle**: this class of latency occurs when the memory request goes to an idle or precharged DRAM bank, requires an activation (ACT), and then read/write.
- **row-hit**: this class of latency occurs when the memory request happens to goes to a DRAM page that was left open by some previous memory requests.
- **row-miss** this class of latency occurs when the memory request goes to the a DRAM bank that has a different page opened by previous memory requests. Therefore, to complete this request, the controller must precharge the bank, then activate, and then read/write.
- **refresh** this class of latency occurs when the memory request is delayed by a refresh operation. Depending on whether the request comes before the refresh or during the refresh, the latency in this class may vary.

We do not seek to reproduce the exact latency as cycle accurate simulation, but extrapolate an appropriate latency number based on DRAM timing parameters. We will further explain this in Section 4.2.

4.2 Latency Recovery

Once we have latency classes in hand, combined with DRAM timing parameters, we can recover their latency into approximate DRAM cycles. By doing this we can avoid relying on any specific numbers but rather have a portable generic model. For example, we can simply plug in a DRAM profile with timing parameters to obtain latency numbers for that profile, and if we want latency numbers for a different profile, we simply plug in another DRAM profile without having to retrain the whole model.

We specify how we recover a latency cycle number from each latency class as follows:

- **idle**: the minimum latency of this class is a row access followed by a column access. In DRAM parameters it is typically $tRCD + tCL + BL/2$. Note there are some variances. For instance, read or write operations may have different $tRCD$, tCL values, and for GDDR the burst cycles can be $BL/4$ as well.
- **row-hit**: the minimum latency of this class is simply the time of a column access. In DRAM parameters it is typically $tCMD + tCL + BL/2$. Again, there are protocol specific variances like the **idle** class.
- **row-miss**: the minimum latency of this class is a full row cycle. In DRAM parameters it is typically $tRP + tRCD + tCL + BL/2$.
- **refresh**: We use a refresh counter similar to the refresh counters in DRAM controller, to provide timestamps of when each rank should refresh. We only use the timestamps as references to determine whether a request arrives right before a refresh or during a refresh. If the request comes right before the refresh, then we estimate the latency as $tRFC + tRCD + tCL + BL/2$. If the request arrives during the refresh cycle, e.g. n cycles after the reference refresh clock ($n < tRFC$), then we estimate the latency as $tRFC - n + tRCD + tCL + BL/2$. For example, the refresh counter marks cycle 7200, 14400, *.etc* as refresh cycles for rank 0, and if a request arrives at cycle 7201, then it will be regarded as arriving during a refresh. Now by no means our reference refresh timestamps can matches precisely the real refresh cycle in a cycle accurate simulation, but it is a good approximation for the impact of refresh.

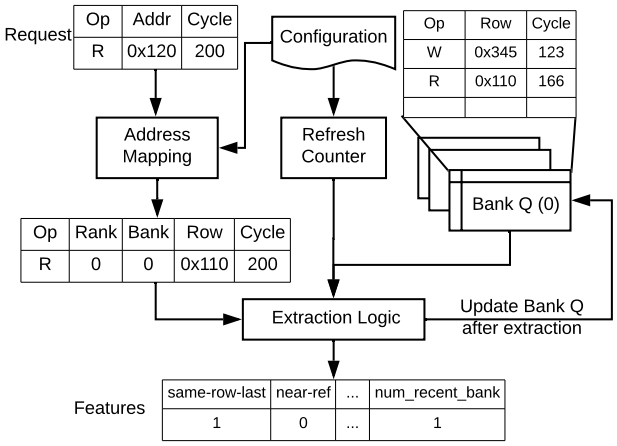

Figure 1: Feature extraction diagram. We use one request as an example to show how the features are extracted.

4.3 Dynamic Feature Extraction

To train a statistical or machine learning model, we need "features" that provides distinctive information about the latency classes.

As we know, the latency of each memory request is largely dependent on the DRAM states, which are the results of previous memory requests. For example, a previous request opened a page when the DRAM bank was idle, then a following request that goes to the same bank same row can take that advantage. So the features we are looking for here should come from the address streams, especially the previous requests, and we need to be able to extract features dynamically from these address streams.

There are two aspects of extracting features from address streams, temporal features and spatial features. Temporal features reflect the potential dependency between requests. For example, a previous request that is 5 cycles ahead should have more impact on the timing of the current request than another previous request 5000 cycles ahead. The difficulty is how to translate the timing intervals into useful features. Again we cannot rely on specific values because there would be too many features to be useful. But instead, we use generic DRAM parameters to classify how far or near is a previous request. For example, we consider a request "near" if it was arrived within tRC cycles, the intuition is that in tRC cycles, which represents the full row cycle, the DRAM can be activated and then precharged by a request, which renders that DRAM state unchanged to a following request outside of tRC cycles. Another line we draw here is the "far" line, which uses the number of $tRFC$, which is the number of cycles it takes to do a refresh. It may imply a reset state for the DRAM.

Table 1: Features with Descriptions

Feature	Values	Description	Intuition
same-row-last	0/1	whether the last request that goes to same bank has the same row (as this one)	key factor for the most recent bank state
is-last-recent	0/1	whether the last request to the same bank added recently (tRC)	relevancy of the last request to the same bank
is-last-far	0/1	whether the last request to the same bank added long ago (tRFC)	relevancy of the last request to the same bank
op	0/1	operation(read/write)	for potential R/W scheduling
last-op	0/1	operation of last request to the same bank	for potential R/W scheduling
ref-after-last	0/1	whether there is a refresh since last request to the same bank	refresh reset the bank to idle
near-ref	0/1	whether this cycle is near a refresh cycle	latency can be really high if it's near a refresh
same-row-prev	int	number of previous requests with same row to the same bank	if there is same row request then OOO may be possible
num-recent-bank	int	number of requests added recently to the same bank	contention/queuing in the bank
num-recent-rank	int	number of recent requests added recently to the same rank	contention
num-recent-all	int	number of recent requests added recently to all ranks	contention

Spatial features need to reflect the structures of DRAM, in particular, banks and rows, because the state of each bank is the most determining factor for the incoming DRAM request. For example, if we are trying to predict the timing of one request, the previous requests that go to the same bank weigh more than the previous requests go to any other banks. And same as temporal features, we don't need to identify each bank and row by their specific bank number, but instead we identify them by "same row", "same bank", "same rank"(but different bank), or "different rank". We can evaluate a request with previous requests on these fields easily once they have their physical addresses translated to DRAM addresses(rank, bank, row, column). And to simplify and facilitate feature extraction, we maintain a request queue for each bank and put requests into each bank queue after the address translation. Unlike the queues in DRAM controllers, this bank queue is not actively managed and is strictly FIFO with a maximum length imposed for performance optimization.

Combining the temporal features with spatial features, we can have features coded with both temporal feature and spatial feature. For instance, $num - recent - bank$ feature counts the number of previous requests that go to the same bank and that are recent. We propose a list of features in Table 1; these features can give hints on the possible state that the DRAM banks are in and how DRAM controllers can make scheduling decisions etc.

The feature extraction using one request as an example is shown in Figure 1.

4.4 Model Training

Having the features and classes ready, we now put pieces together and build the training flow shown in Figure 2. We use synthetic traces as training data, and use a cycle accurate DRAM simulator, DRAMsim3, to provide ground truth. The beauty of using synthetic traces is that we can use a small amount of synthetic traces to represent a wide range of real world workloads. For example, we can control the inter-arrival timings of the synthetic traces to reflect to intensity of workloads; we can also generate contiguous access streams and random access streams and interleave them to cover all types of memory access patterns of real workloads. Plus, we also don't have to worry about the contamination of testing dataset when we test the model with real workloads.

We run the synthetic traces through DRAMsim3 with a DRAM configuration file as usual. To mark the ground truth, we modified DRAMsim3 so that it generates a trace output that can be used for training. Because the DRAMsim3 knows exactly what happens to each request inside its controller, it can precisely classify the requests into any of the categories we proposed in Section 4.1. And once the requests are classified, we run them through the feature extraction to obtain features. Finally, we run the features along with the classes into a model to obtain a model.

There are many machine learning models that can potentially handle this particular classification problem and we are not going to test every one of them as it is out of the scope of this thesis. In

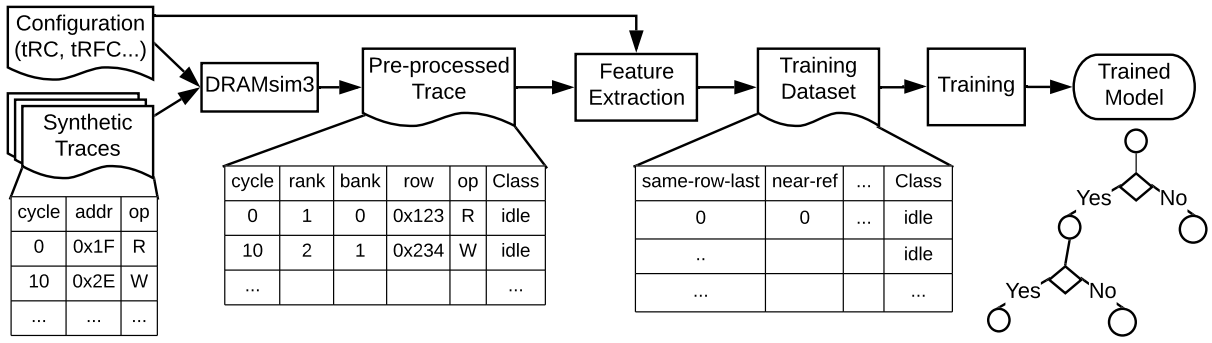

Figure 2: Model Training Flow Diagram

this study we start with simple and efficient models like decision tree[15] and random forest[11] for this study:

- These models are simple, intuitive, and explainable, which is quite important for prototyping work like this: it helps to be able to look at the model to examine and debug.
- The an ensemble tree model mimics how a DRAM controller works naturally, but instead of doing it in a series of cycles, the decision tree makes the decision instantly.
- The simplicity of these models makes both training and inference fast, the later is crucial for simulations performance.

As far as hyperparameter tuning goes, while decision tree model is relatively simple, there are still many hyper-parameters to tune. Luckily the model is not hard to train, and it does not take much time to go through many parameters. We tried several different approaches(including brute force), and they all work decently. But what we have found that produced best accuracy is Stratified K-fold Cross Validation[10]. Stratification samples help reduce the imbalance of classes in our training dataset, especially the **refresh** class that is much rarer than other classes. K-fold Cross Validation divides the training data sets into k folds, and for each fold, uses it as test set and the rest $k-1$ folds as training sets.. This further reduces the bias and overfitting of the model. The details of hyperparameter training can be found later in Section 5.1.

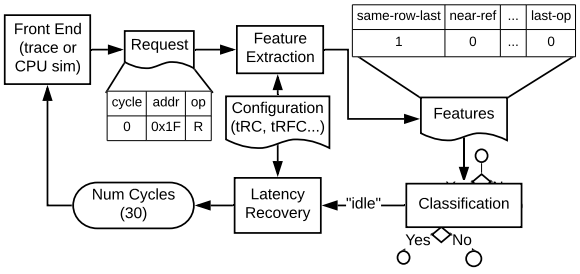

Figure 3: Model Inference Flow Diagram

4.5 Inference

Inference is relatively straightforward, as shown in Figure 3. However, one thing to note is that, if we are to compare the inference results to the results of cycle accurate simulation, we have to use the same DRAM configuration profile as the cycle accurate simulation. Otherwise we are not required to use the same DRAM configuration profiles.

In implementation, the entire inference process only takes one function call combining the request cycle, address, and operation (read/write), and the inference function returns the number of cycles that the request is going to take to complete.

This is great relief from the cycle accurate interface where the frontend has to always stay synchronized with the DRAM model. It allows much more flexible integration into other frameworks.

4.6 Other Potential Models

The innovation of this work is to translate what is an essentially time-series problem into a non-time-series problems. We are aware of that there are models that work on time-series problem. Some of the temporal features in the data are easy to extract, whereas if we use models to automatically extract features, it will be costly when it comes to training. Additionally, our approach preserves portability and model reusability when it comes to different DRAM profiles, which we believe is not easy to preserve in other models. That being said, we certainly look forward to other efficient implementations of this problem.

5 EXPERIMENTS & EVALUATION

5.1 Hyperparameters

We use the *Scikit-learn*[14] package to train our model, which contains a set of tools and models that are readily available. The hyperparameters we use for training the decision tree model is shown in Table 2. We use the $StratifiedShuffleSplit$ module to conduct a grid search on the these hyperparameters to find the best fitting model.

On top of these hyperparameters, we also use $k-fold=5$ for K-fold cross validation. The end result is that there are a total of $5400 \times 5 = 27000$ models to train. Fortunately, each model trains quickly and the training can be distributed to multiple cores/threads

Table 2: Hyperparameters explored for the decision tree model.

Hyperparameter	Values	Explanation
max-depth	None, 3, 5, 8, 10	Max depth of any path in the decision tree ("None" means unlimited)
min-samples-leaf	5, 20, 20, 30, 0.1, 0.2	Min number of samples needed to create a leaf node in the tree. Float number means ratio.
min-samples-split	5, 10, 20, 0.05, 0.1	Min number of samples to create a split in the tree. Float number means ratio.
min-weight-fraction-leaf	0, 0.05, 0.1	Min weighted fraction of the sum total of weights required to be at a leaf node.
max-features	auto, 0.2, 0.5, 0.8	Max number of features to consider. Auto means square root of number of features.
random state	1, 3, 5	Help train reproducible model.

in parallel. It takes less than a minute to train and evaluate all 27000 models using a 4-core desktop CPU.

The best hyperparameters are automatically selected based on accuracy. The values of the "best" hyperparameters are listed in Table 3. The best accuracy is **96.76%** (for all cross-validation data).

Table 3: Hyperparameter Values of Best Accuracy

Hyperparameter	Value
max-depth	None
min-samples-leaf	20
min-samples-split	5
min-weight-fraction-leaf	0
max-features	0.8
random state	3

As for the hyperparameters of random forest model, the default parameters provided by Scikit-learn package works out of the box, trains in seconds, and produces an accuracy as good as the decision tree model. Therefore we did not explore the hyperparameters of random forest model.

5.2 Feature Importance

We analyze the trained models and see how they treat the features differently. To better visualize the importance of features, we clip the least important features into one category("others"), and plot the pie chart as Figure 4.

The two models handles different features differently, with the two most important features the same: $same-row-last$ and $ref-afterr-last$, contributing to more than 50% combined. It can also be seen that the distribution of importance is more balanced in the random forest model than the decision tree model.

5.3 Benchmarks

To comprehensively test the trained model, we evaluate our model against cycle-accurate DRAMsim3 with memory traces of real-world benchmarks. We use a subset of SPEC CPU2017 benchmarks [2] that are most representative according to [13], this allows us to

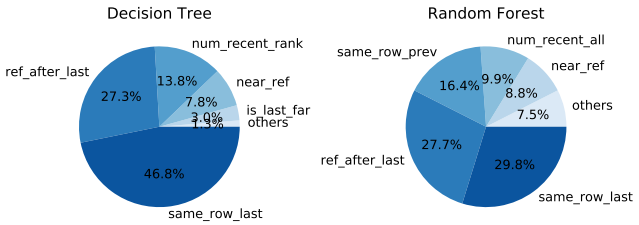

Figure 4: Feature importance in percentage for decision tree and random forest

test benchmarks with different memory characteristics. These SPEC benchmarks are *bwaves_r, cactuBSSN_r, deepsjeng_r, fotonik3d_r, gcc_r, lbm_r, nab_r, mcf_r, x264_r*, and *xalancbmk_r*. We also include *STREAM*, which is very bandwidth sensitive, and *ram_lat*, an LMBench-like memory benchmark that is latency sensitive. These benchmarks will show us the full spectrum of memory characteristics and behaviors.

5.4 Accuracy

We run all the benchmarks mentioned in Section 5.3 with cycle accurate, out-of-order Gem5 CPU along with DRAMsim3, and this will provide us the golden standard for our accuracy tests. A address trace for each of the benchmark is generated as the input to the statistical models, this allows the statistical model and cycle accurate model to have exactly the same inputs to work with. For each request we also record its latency class and latency value in cycles labeled by DRAMsim3 so that we can use it for comparison with the statistical models.

Note that there are two aspects of accuracy, classification accuracy, which represents how many requests the statistical model can correctly classify according to the cycle accurate models; and latency accuracy, which is the numeric latency values of the request produced by the statistical models.

First we look at **classification accuracy** for decision tree and random forest models, as shown in Figure 5 and Figure 6.

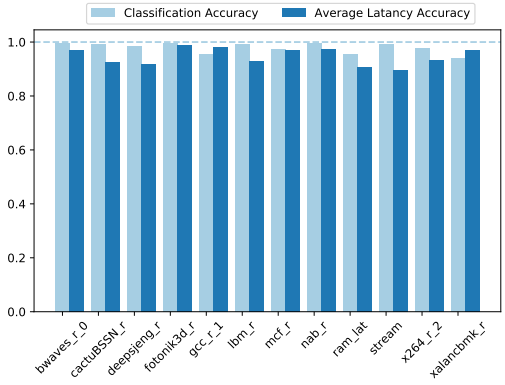

Figure 5: Classification accuracy and average latency accuracy for decision tree model on various benchmarks.

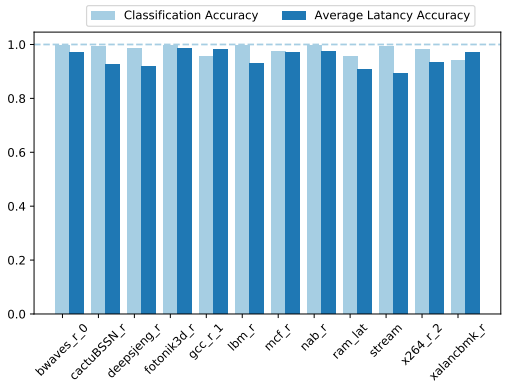

Figure 6: Classification accuracy and average latency accuracy for random forest model on various benchmarks.

As can be seen in Figure 5 and Figure 6, overall the predictors produces great classification accuracy across all benchmarks for both models. The average classification accuracy is 97.9% for decision tree and 98.0% for random forest. In most cases the classification accuracy even exceeds our training accuracy. This is because our training traces generally contain more address patterns than most real-world workloads, and is thus harder to work with. Also note that the accuracy between random forest and decision tree models is almost indistinguishable, the largest difference being a mere 0.4% for *lbm* benchmark, in all other cases the difference is often 0.1% to 0% difference. This shows signs of our model converging.

Being able to correctly classify the latency categories is the important first step. The next step is to verify that our latency recovery model can also reproduce the latency value in cycles according to the DRAM configuration profile. Our model translate the latency class for each memory request to a numeric value in DRAM cycles, and we compare these numeric latency values against our cycle

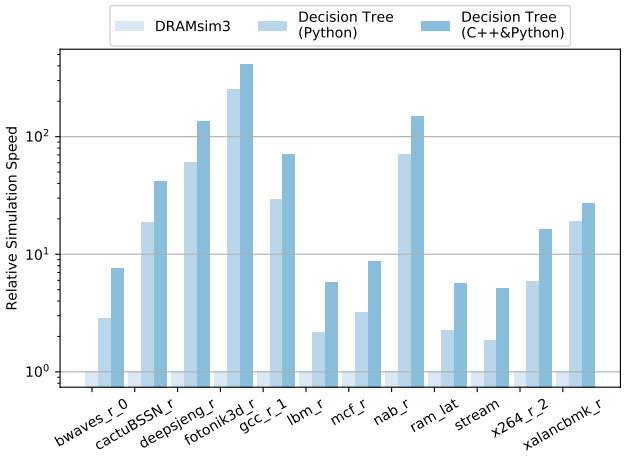

Figure 7: Simulation speed relative to cycle accurate model, y-axis is log scale.

accurate model baseline. To put into perspective of the classification accuracy, we again use the cycle accurate numbers as the baseline and plot the average latency value from our statistical models normalized to the cycle accurate model as the accuracy measurement. We call this measurement "*Average Latency Accuracy*". The average latency accuracy are plotted side by side with classification accuracy in both Figure 5 and Figure 6. The average relative latency for both decision tree model and random forest model are **0.969** comparing to cycle accurate base with worst case 0.94 for *stream* and *ram_lat* benchmarks. We also have a discussion on how to further reduce the latency accuracy in Section 5.6.

5.5 Performance

We now compare the simulation performance. The simulation time of each model is measured as the total time it takes to process all input memory requests. We use the inverse of simulation time as simulation speed, and plot the simulation speed normalized to the cycle accurate simulations, as shown in Figure 7. Note that we have 2 implementations of our statistical models, the first one being a pure *Python* implementation from end to end, and the other one is a hybrid *Python* and *C++* implementation where the feature extraction part of the program is implemented in *C++* and the rest of the model is implemented in *Python*. These two implementations produces the same results in terms of accuracy but only differs in simulation speed. Therefore, there are 3 datapoints for each benchmark in Figure 7, with one representing cycle-accurate DRAMsim3, the other 2 representing 2 implementations of the statistical model.

It can be seen that our prediction model runs 5x to 400x faster than DRAMsim3 (with the *C++&Python* implementation). The simulation speed of an inference model is solely dependent on the number of requests, because the work to predict each request is the same. In contrast, there are many more factors for cycle accurate simulations: first off, each cycle has to be simulated even if there is no memory request at all; the memory address patterns, which alter the behavior of scheduler, also impact the simulation performance.

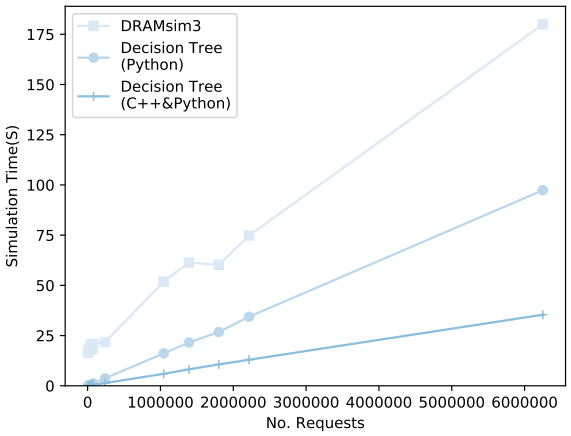

Figure 8: Simulation time vs. number of memory requests per simulation.

To demonstrate linearity of inference models, we sort the benchmarks by the number of memory requests they generate, and plot the simulation time over the the number of requests as in Figure 8. Both implementations of the decision tree model exhibit almost strictly linear performance. We can conclude that the time complexity of our model is $O(n)$ where n is the number of requests, and hence $O(1)$ for each request.

5.6 Multi-core Workloads

Previous experiments show our proposed model can successfully model DRAM timings for single core workloads, no matter the memory activity intensity. The success is based on the premise that the high accuracy classification can translate to high accuracy latency prediction because the variances are low in each class. While this might be true for single core workloads, multi-core workloads may break the assumption.

Table 4: Randomly mixed multi-workloads.

Mix	Benchmarks
0	*stream, xalancbmk_r, lbm_r, bwaves_r*
1	*bwaves_r, mcf_r, lbm_r, fotonik3d_r*
2	*deepsjeng_r, bwaves_r, gcc_r, fotonik3d_r*
3	*xalancbmk_r, x264_r, bwaves_r, gcc_r*
4	*mcf_r, stream, ram_lat, lbm_r*

To validate how well our model holds against scaling workloads, we amplify the workload by randomly mixing 4 traces of different workloads together to reflect intensive multi-core memory activities, and we use the same methodology to evaluate the accuracy. The mix of benchmarks is shown in Table 4.

It can be seen, in Figure 9, that our model still demonstrate very high classification accuracy with 0.99 for each mix, but the average latency sees a decrease down to 0.88 in the worst case (mix 4). The accuracy disparity between classification accuracy and latency

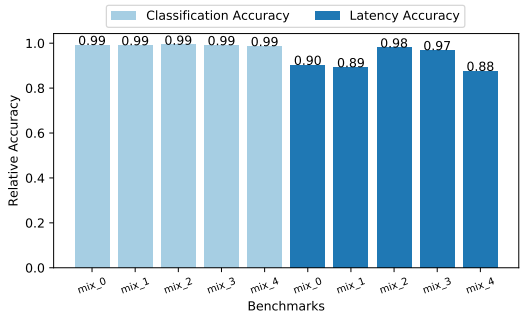

Figure 9: Classification accuracy and average latency accuracy for randomly mixed multi-workloads.

accuracy is due to the long gap between the latency category edges. For instance, in the DDR4 configuration we tested, the $row-hit$ class is 22 cycles while the next near class, *idle*, is 39 cycles, leaving 17 cycles in between.

To further quantify this effect, we breakdown each latency class to those whose actual (cycle accurate simulated) latency matches exactly with their predicted latency; and those whose actual latency is more than their predicted latency, which we name as "$Class+$". For instance, the in the DDR4 configuration, $row-hit$ class translate to 22 cycles, while $row-hit+$ class represents those requests that are "row hit" situations but with more than 23 cycles due to contention. Figure 10 shows the breakdown of such classes for each mix. Each bar in the graph represents the percentage of the total requests for each class. Note that the predicted latency of $refresh$ classes is a variation itself so it does not accompany a "+" class like others. It can be seen that for mixes that have higher latency accuracy such as $Mix2$ and $Mix3$, the percentage of the "+" classes are much smaller, typically less than 10 percent combined. The opposite can be observed from other mixes such as 0, 1 and 4, where the "+" classes contribute to more than 20% of the total requests, resulting the inaccuracy in their latencies. Further looking into the specific benchmarks in each mix, we can confirm that the mixes with higher percentage of "+" all consist of more than 2 memory intensive benchmarks, whereas the mixes with lower percentage of "+" have at most 1 memory intensive benchmark.

One way to combat the extra latency introduced by contention is to train the model with more latency classes, i.e., filling the latency gap between current classes with more latency classes. This may increase the training efforts but should reduce the latency discrepancy between our statistical model and cycle accurate model.

6 DISCUSSION

6.1 Implications of Using Fewer Features

In the early stage of prototyping the machine learning model, we did not obtain results as good as Section 5.4. However, these results are still valuable in providing insights to the future improvement of the model. Therefore, we document the early prototype and results in this discussion.

One early prototype we had did not have the FIFO queue structure, but instead only keeping the latest previous memory request

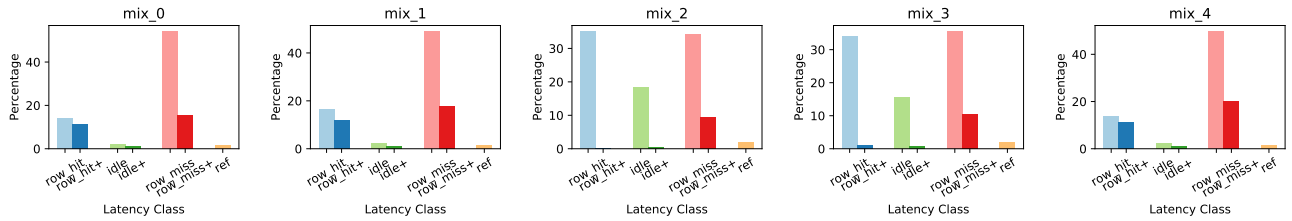

Figure 10: Request percentage breakdown of latency classes and their associated contention classes for randomly mixed multi-workloads. "+" classes are the contention classes apart from their base classes.

to the same bank, i.e. effectively a $depth = 1$ queue. This only allows us to extract features such as *same-row-last, is-last-recent, is-last-far, op,* and *last-op*. We only trained decision tree for for evaluation, and the classification accuracy and average latency accuracy for each of the benchmarks we tested are shown in Figure 11.

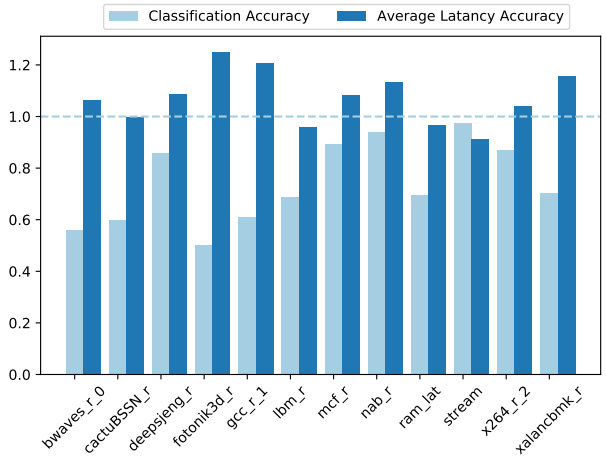

Figure 11: Classification accuracy vs average latency accuracy of an early prototype of a decision tree model.

As can be seen in Figure 11, the classification accuracy ranges from 0.5 to 0.94, with an average of 0.74; perhaps surprisingly, the average latency accuracy is better on numbers: ranging from 0.91 to 1.25, with an average accuracy of 1.07, or an absolute 10% error. In some benchmarks, classification accuracy can be 40 to 50 percent off while latency difference is much smaller. The reason behind this is that, with only the last request to the same bank being recorded, the model tends to predict more requests as *row-hit* or *row-miss* than it should, whereas in reality, a lot of these requests should be *idle*. Coincidentally, with the DDR4 DRAM parameters, the average latency of *row-hit, 22 cycles*, and *row-miss, 56 cycles*, is *39 cycles*, which is the *idle* latency. Therefore, while lots are latency classes are mis-predicted, the average latency numbers are not too far off. This presents a good reason that we should examine both classification accuracy and latency accuracy instead of focusing solely on one measurement.

The lack of tracking for previous requests beyond one entry, and no account for refresh operations are the primary reasons for low classification accuracy. Tracking for previous requests beyond one entry allows the scheduler to make out-of-order scheduling decisions. Another wildcard that we did not anticipate is the role of *refresh*. Although there are typically only less than 3% of memory requests are directly blocked by DRAM refresh operations, the subsequent impact of refresh is larger: each DRAM refresh operation resets the bank(s) to idle state, which leads to the next round of requests to these banks to have idle latency. When there are not many requests issued to the refresh-impacted banks in between two refreshes, the refresh operation will render a much larger impact.

6.2 Interface & Integration

Traditionally, the cycle accurate DRAM **simulator interface** is "asynchronous", meaning that the request and response are separated in time: the CPU simulator sends a request to the DRAM simulator without knowing at which cycle the response comes back; while waiting for the memory request to finish, the CPU simulators has to work on something else every cycle; finally, when the DRAM simulator finishes the request, it calls back the CPU simulator, who processes this memory request and its associated instructions. This asynchronous interface only works in cycle accurate simulator designs, as the CPU simulator has to check in with the DRAM simulator every cycle to get the correct timing of each memory request.

The statistical model, however, brings an "atomic" interface to the simulator design, meaning that upon the arrival of each request, the timing of this request can be provided back to the CPU simulator immediately with high fidelity. This will enable much easier integration into other models than cycle accurate models. For example, when integrated into an event-based simulator, the response memory event can be immediately scheduled to the future cycle provided by the statistical model, and no future event rearranging is needed.

Furthermore, the atomic interface provided by the statistical model will benefit parallel simulation framework. Because in a parallel simulation framework, simulated components interacting with each other generate synchronization events across the simulation framework, and frequent synchronization will negatively impact the simulation performance. The statistical model only needs to be accessed when needed, thus reducing the synchronization need to a minimum.

7 CONCLUSION & FUTURE WORK

In this paper, we proposed and implemented a novel machine learning based DRAM latency model. The model achieves highest accuracy among non-cycle-accurate models, and performs much faster than a cycle accurate model, making it a competitive offering for cycle accurate model replacement.

The model still has room to improve as future works. First off, if the entire flow can be implemented in *C++*, we can expect more performance gain without any impact on classification accuracy. Secondly, introducing more latency classes can bridge the gap between latency accuracy and classification accuracy for memory intensive workloads. Or rather, more latency classes can be constructed to model the working mechanisms of more sophisticated controller/scheduler designs beyond our currently modeled out-of-order open-page scheduler, providing more flexibility to the model. Finally, we only trained and tested decision tree and random forest models for the purpose of prototyping, and we realize that there are lots of alternative machine learning models that could also work for this problem, so it may be worth exploring other models in the future.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[2] James Bucek, Klaus-Dieter Lange, et al. 2018. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 41–42.

[3] Trevor E Carlson, Wim Heirmant, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[4] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep* (2012).

[5] Hyojin Choi, Jongbok Lee, and Wonyong Sung. 2011. Memory access pattern-aware DRAM performance model for multi-core systems. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 66–75.

[6] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. 2014. Simulating DRAM controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 201–210.

[7] Min Kyu Jeong, Doe Hyun Yoon, and Mattan Erez. 2012. DrSim: A platform for flexible DRAM system research. *Accessed in: http://lph. ece. utexas. edu/public/DrSim* (2012).

[8] Matthias Jung, Christian Weis, Norbert Wehn, and Karthik Chandrasekar. 2013. TLM modelling of 3D stacked wide I/O DRAM subsystems: a virtual platform for memory controller design space exploration. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 5.

[9] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.

[10] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Montreal, Canada, 1137–1145.

[11] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[12] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[13] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. 2018. Wait of a decade: Did spec cpu 2017 broaden the performance horizon?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 271–282.

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[15] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[16] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters* 10, 1 (2011), 16–19.

[17] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer architecture news*, Vol. 41. ACM, 475–486.

[18] Sadagopan Srinivasan, Li Zhao, Brinda Ganesh, Bruce Jacob, Mike Espig, and Ravi Iyer. 2009. CMP memory modeling: How much does accuracy matter? (2009).

[19] Vladimir Todorov, Daniel Mueller-Gritschneder, Helmut Reinig, and Ulf Schlichtmann. 2012. Automated construction of a cycle-approximate transaction level model of a memory controller. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 1066–1071.

[20] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News* 33, 4 (2005), 100–107.

[21] George L Yuan, Tor M Aamodt, et al. 2009. A hybrid analytical DRAM performance model. In *Proc. 5th Workshop on Modeling, Benchmarking and Simulation.*