

Online Model Swapping for Architectural Simulation

Patrick Lavin
Georgia Tech, CSE
Atlanta, GA, USA
plavin3@gatech.edu

Jeffrey Young
Georgia Tech, CS
Atlanta, GA, USA
jyoung9@gatech.edu

Richard Vuduc
Georgia Tech, CSE
Atlanta, GA, USA
richie@cc.gatech.edu

Jonathan Beard
Arm Research
Austin, TX, USA
jonathan.beard@arm.com

Abstract

As systems and applications grow more complex, detailed computer architecture simulation takes an ever increasing amount of time. Longer simulation times result in slower design iterations which then force architects to use simpler models, such as spreadsheets, when they want to iterate quickly on a design. Simple models are not easy to work with though, as architects must rely on intuition to choose representative models, and the path from the simple models to a detailed hardware simulation is not always clear.

In this work, we present a method of bridging the gap between simple and detailed simulation by monitoring simulation behavior online and automatically swapping out detailed models with simpler statistical approximations. We demonstrate the potential of our methodology by implementing it in the open-source simulator *SVE-Cachesim* to swap out the level one data cache (*L1D*) within a memory hierarchy. This proof of concept demonstrates that our technique can train simple models to match real program behavior in the *L1D* and can swap them in without destructive side-effects for the performance of downstream models. Our models introduce only 8% error in the overall cycle count, while being used for over 90% of the simulation and using models that require two to eight times less computation per cache access.

CCS Concepts

• **Computer systems organization** → **Architectures**; • **Computing methodologies** → **Modeling methodologies**; **Discrete-event simulation**.

Keywords

Architectural simulation, Statistical simulation, Cache modeling, Model swapping, Phase detection

ACM Reference Format:

Patrick Lavin, Jeffrey Young, Richard Vuduc, and Jonathan Beard. 2021. Online Model Swapping for Architectural Simulation. In *Computing Frontiers Conference (CF '21)*, May 11–13, 2021, Virtual Conference, Italy. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3457388.3458670>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF '21, May 11–13, 2021, Virtual Conference, Italy
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8404-9/21/05...\$15.00
<https://doi.org/10.1145/3457388.3458670>

1 Introduction

With traditional lithography-scaling slowing and Dennard scaling effectively ending [7, 9, 18], performance increases will come increasingly from specialization and scaling ever upward and outward. Rapid prototyping tools that enable system architects to determine the best composition of algorithm, architecture, and scale are critical to enabling the development of next-generation systems. Computer architecture modeling today is largely accomplished through discrete event simulation, therefore the time to model a system is roughly proportional to the number and detail of the components modeled. To assess a given architectural configuration, modelers set up a simulation with a fixed architectural topology. Each model within this topology is typically fixed within the simulation infrastructure for the duration of simulation. However, applications are not fixed, and they typically have rich variation in activity from one point in time to the next, with some phases of relatively stable behavior. We intend to exploit this variation by dynamically swapping from complex to simple models for these phases of stable behavior. Detecting when, where, and how to select simpler models while ensuring equivalent model fidelity is the primary contribution of this work.

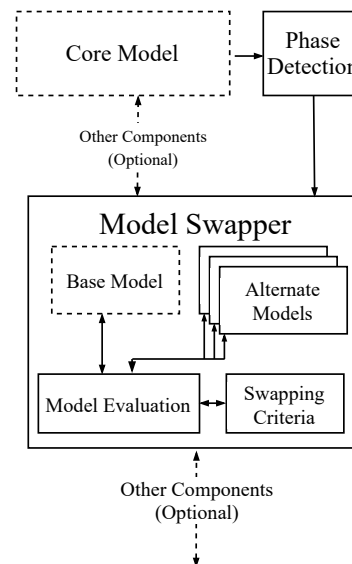


Figure 1: A diagram of the model swapping components. Dashed lines represent components that should already exist in a simulation and solid lines represent components covered in this work. In this diagram, *Other Components* means other simulation components that will remain intact and not be swapped out.

Computer architecture simulation (simply *simulation* from here on out for brevity) is a critical tool for determining how to shape future architecture. *Simulation* is largely synonymous with discrete event simulation of a computer system-on-chip [1], with each sub-component of the simulator representing some module within the target architecture. Early *simulators*, such as SimpleScalar [2], modeled every function or component of the system in some level of detail. This practice was fine for the age in which they lived, however simulation of a modern multi-core system-on-chip with a framework such as SimpleScalar would take an inordinate amount of time. More recent simulators such as gem5 [4] adopt a method known as “sim-points” [25] in order to speed simulation by adopting a statistical approximation for the behavior of entire phases across an entire simulation. While this approach speeds simulation, it is offline, requiring architects to generate the sim-points for a program before simulating it, and it doesn’t give architects the choice of which components are modeled in detail, and which are approximated. Our approach enables automated, online migration to simpler models during a single run of the simulator.

Modern computer architects have a myriad of computer simulation frameworks to choose from (e.g. [4, 14, 20, 22]). This work will apply to many of them, assuming that simulator framework is “plug-able” (e.g. [22]), that is, the simulator modules themselves have a defined interface. We assume that the plugin mechanism defines any state transfer between plugin modules and that state transfer only occurs at module boundaries through defined interfaces or ports (i.e., no global state). Within this framework, model swapping is trivially possible, that is the reader can surely imagine changing out components by re-connecting input and output ports as is depicted in Figure 1. This oversimplification glosses over necessary breakage of this pure model (e.g. synchronization primitives must exist, and often have state), but it provides a basis to build on. With this definition of a plug-able framework, we next need to decide how to swap modules while respecting the spatial and temporal output that downstream models expect. Our work examines swapping in the *LID* so that we can examine such effects. Given the importance of the memory system to so many applications, improvements here also have the potential to greatly speed up simulations.

Our primary contributions, to be detailed within this work, are summarized by the following: 1) We present a proof of concept for online model swapping within an architectural simulation. 2) We demonstrate multiple statistical models that can approximate a functional cycle-accurate model 3) We detail a model selection methodology that can be used for online model selection.

2 Online Model Swapping

The scope of an idea like model swapping is quite broad and needs to be narrowed down before we can make headway with an implementation. Let us first take a high-level look at the four components of our scheme, which, in general, are answers to the four following challenges:

- **How do we find sections of execution that are simple enough to model?** Our goal of replacing components of a simulation with easy-to-compute statistical models relies on us breaking up the computation into chunks which are

easier to model than the program as a whole. This will be addressed by *Phase Detection*.

- **What do we swap in place of the base, detailed model?** We need to choose statistical models to provide to our model swapping algorithm. Criteria for such models will be addressed in *Alternative Models*.
- **How do we evaluate alternative models?** As we will have multiple statistical models available to us at simulation time, we must identify criteria for ranking the relative performance of the models. Things like model speed and performance must be compared. The question of evaluating models will be addressed in *Model Evaluation*.
- **How do we decide to swap out a model?** At simulation time, we need criteria for deciding when it will be beneficial to swap out the base model for one we have trained ourselves. Deciding to swap the models means considering the criteria previously mentioned and deciding if it will be worth it to make the swap. Making the decision between models is covered in *Model Swapping*.

We will answer these four questions in the following subsections.

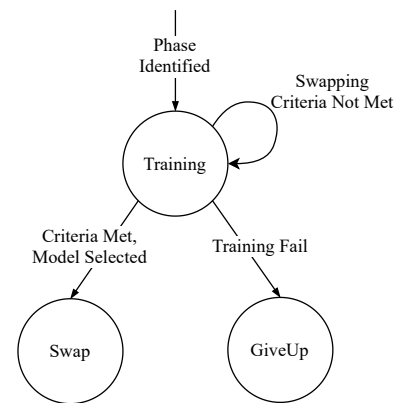


Figure 2: A state diagram for the abstract model swapping algorithm. State transitions happen upon interval end, as defined by the Phase Detector. Alternate models are trained per-phase, and once swapping criteria have been met, that phase will move into the *Swap* state, where a single alternate model is chosen for that phase for the rest of the simulation. If training seems too hard for a phase, or takes too long, a *GiveUp* state can be entered, so that we can stop spending computational resources on a phase deemed too complex.

2.1 Phase Detection

In order to train simple statistical models to predict behavior, we need to break up the computation into chunks which themselves exhibit simple behavior, at least for the component we want to swap out. This can be accomplished with the help of phase detection[15], which, as the name implies, detects phases within a program. Phases represent portions of the program that display similar characteristics, such as having a consistent number or branch misses or the same set of instruction pointers. Many algorithms exist in this field,

but we will only use one in this work, which is laid out in Section 3.1. Abstractly, it is important for whatever phase detector that is chosen to be able to identify a phase identifier (*ID*) for the most recent interval, and for it to share that *ID* with the other components of the simulation, so that they know what phase has just run, and can use that info to pick models.

2.2 Alternative Models

An obvious need for model swapping is to have multiple models to choose from. The only requirement for these models is that they statistically approximate the original base model responses given a time series of inputs. For instance, a cache model that predicts a miss at time *T* sends a response to a downstream cache level (e.g. the level two data cache). If the model receives any sort of coherence traffic that needs a response, the model must support this too.

2.3 Model Evaluation

Various criteria exist for choosing between different statistical models based on things like model size and the number of required sample points. For this work, we highlight the four parameters that need to be considered in any model selection methodology:

- *Accuracy* - the statistical model should accurately represent the behavior of the base model it is replacing,
- *Side Effects* - the statistical model should not disturb the statistics of other parts of the simulation,
- *Model Size* - the model should ideally be smaller than the base model given that memory access is likely dominant in the simulation,
- *Model Complexity* - the statistical models should ideally require less time to perform prediction than the base model.

These four criteria will enable us to limit the choice of models thereby giving us a method of choosing one online during simulation.

2.4 Model Swapping

The final part part of any model swapping is the algorithm that decides when it is time to stop training a statistical model and to swap one in. This process is depicted in a state diagram in Figure 2. This algorithm is run per-phase, and the transitions between states occur on interval boundaries.

To instantiate such an algorithm, all of the other components described so far must be in place, and swapping criteria must be defined so that this algorithm can eventually terminate. Optionally, a time limit can be set so that training is permitted to fail, at which point the swapping algorithm gives up on trying to train a model for that particular phase.

3 Proof of Concept: Swapping the L1 Data Cache

Now that we’ve explained all the components of our model swapping methodology, it’s time for us to implement it. To accomplish this, we implement model swapping for the L1 data cache (*L1D*). We choose the *L1D* specifically because it is central to many simulations, and changes to this model have real consequences for

downstream models consuming its output. In this section, we’ll discuss the phase detection, alternate models, and model swapping algorithm that we have chosen, and in Section 4, we’ll discuss how well our method works.

3.1 Phase Detection

In order to reduce the degrees of freedom for our proof of concept, we opt for a simple solution to the phase detection problem, based on working sets, which was first described in [10]. We define an interval as a continuous block of 10,000 instructions. In our case, instructions consist of only memory references, as this is the information that will be available to the memory hierarchy. For each interval, the instruction pointers are hashed into a bit-vector, which serves as the signature for that interval. We can compare interval signatures with a simple similarity metric, and if we have enough consecutive similar intervals, we will classify this as a phase, and begin training models to fit that phase.

We have included the parameters (such as the signature size and minimum number of similar signatures required to declare a new phase) as well as pseudo-code for this phase detection algorithm in Appendix A. While simple, this algorithm faithfully reproduces the known phases for the program. Thus, we deem it suitable for our proof of concept.

3.2 Alternate Cache Models

We need a selection of models to use in place of the *L1D* cache when we swap out the base model. In this work, we present three simple models. As we are working to replace a cache, the behavior we need to replace is that of the *hit check* where we ask a cache whether or not it has a line in residence. Thus, a cache model needs to take information about a request (such as the instruction pointer and whether it is a read or a write) and use this to predict whether the access is a hit or a miss.

	RH	RM	WH	WM		RH	RM		WH	WM
RH	0.4	0.4	0.1	0.1	RH	0.5	0.5	RH	0.5	0.5
RM	0.3	0.1	0.3	0.1	RM	0.75	0.25	RM	0.75	0.25
WH	0.1	0.1	0.1	0.7	WH	0.5	0.5	WH	0.125	0.875
WM	0.4	0.3	0.2	0.1	WM	0.57	0.43	WM	.67	.33

Figure 3: Markov-Chain Based Models: Our Markov-chain based models restrict the transition probabilities based on information regarding the next access. If the next access is a read, we will only allow the model to predict a transition to a ReadHit (RH) or ReadMiss (RM) state. This image depicts the 4-state Markov model, but the 8-state model works similarly.

3.2.1 Fixed Hit Rate The first and most basic model is the *Fixed Hit Rate Cache Model*, also referred to as the *Fixed Rate Cache Model*. In this model, we count the number of hits and misses during a phase’s execution, and calculate a hit rate for the phase. When using

Table 1: Model Complexity

	Size (Bytes)	Hit Check (comparisons)	Training (comparisons)
Base	8192	16	16
Fixed Rate	16	1	1
Markov 4	384	1	$O(N^2)$
Markov 8	1536	2	$O(N^2)$

this model for prediction, we generate a random number uniformly spaced between zero and one, and check whether that number is below the hit rate or not. If it is, we say the access is a hit, otherwise, we classify it as a miss, and pass it along to the *L2*.

3.2.2 4-State Markov Model The fixed rate model is simple, which likely means it won't be able to represent the behavior of most real programs. To add information on access history to our prediction, we use a Markov-chain based model. Our first Markov model includes four states, *ReadHit*, *ReadMiss*, *WriteHit*, and *WriteMiss*. During training, we learn the transition probabilities from one state to the next. During prediction, we generate a uniform random number and move to the state indicated by the transition probability. However, we have one caveat: if the next request to the cache is a read, we do not want our Markov model to predict that we move to a *WriteHit* or *WriteMiss* state. Thus, during prediction, we restrict the model to moving to states corresponding to the current cache access. This behavior is depicted in Figure 3, and the pseudo-code is shown in Appendix B.

3.2.3 8-State Markov Model While the 4-State Markov model adds some history, neither it nor the Fixed Hit Rate model capture one of the most important performance features of a cache, namely spatial and temporal locality. To remedy this, we have an 8-state Markov-chain model that adds a small amount of locality information. It has the same states as the above 4-state Markov model, but it has both *Near* and *Far* versions of each of the four states. A *Near* state is defined as cache access occurring on the same cache line as the last one, and a *Far* access as anything else. Note that to be consistent, we used 64B as the boundary for a cache line, which means two cache lines are *Near* for the L1 model and one cache line is near for the L2 model. Similar to the 4-state model, we must restrict the states we allow the model to transition to during prediction based on both whether or not the next access is a read or a write, and also whether the access is considered *Near* or *Far*. This means that prediction is only able to transition to one of two states, so the entire prediction takes only two comparisons (one to determine if the access is near or far, and one to determine if it is a hit or a miss).

3.3 Model Evaluation Criteria

As mentioned in section 2.3, we have identified four areas to evaluate models on. For a cache component, we have defined them as follows:

3.3.1 Accuracy For our purposes, accuracy is defined as the percentage of hits correctly predicted. This is measured during simulation by training the statistical models as soon as we have any data on the phase (i.e., as soon as it is identified by the phase analysis

component), and running the partially trained models alongside the base model until we have a good idea of the accuracy.

3.3.2 Side Effects In the cache, the most important side effect to consider is the locality of the references that miss in the *L1D*, as these are the ones that go to the *L2*. Many metrics for locality are computationally intensive, and since our goal is to speed up simulation, we have chosen a simple proxy for locality, the *near miss count*. With *near* being defined as in the 8-state Markov model: if two accesses are on the same cache line, they are considered near, and otherwise classified as far. Thus, the metric is simply the number of misses that are classified as near. We can compare this with the number of near misses from the base model to get an idea of the locality properties of the accesses going to *L2*.

3.3.3 Model Size Aside from metrics derived at runtime, there are some static information we can use in model selection, the first of which is the model size.

- The base model is a 32 KiB, 8-way set associative cache, with 4 B words. This gives us 128 sets with 8 tags each, and stored as 8 B integers. This is a total size of 8 KiB.
- The Fixed Hit Rate model requires storing an 8 B hit count, and an 8 B hit rate, meaning the full size is only 16 B.
- The Markov models each require storing an $N \times N$ transition count matrix, and an $N \times N$ transition probability matrix, where N is the number of states. They also memoize the restricted transition matrices (depicted in Figure 3). This amounts to another $\frac{N}{2}$ matrices of size $N \times 2$ which brings the total cost to $3N^2$. Assuming that these are all stored with 8-byte values, we end up with 384 B for the 4-state Markov model, and 1536 B for the 8-state model.

3.3.4 Model Complexity We measure model complexity as the number of comparisons required to make a hit check. For the base model, this takes 16 comparisons, as it does a linear search over the tags present in the set, which is 8-way associative. Another search will be needed for eviction. For the Fixed Hit Rate Model, a single comparison is needed to see if the random number generated is less than the hit rate.

Due to the way the Markov models restrict the states they can move to, we actually only need a single comparison for the 4-state model. The 8-state model needs an additional comparison, to check if the current address is near or far. This was explained in depth in Section 3.2.2. The numbers for the model size and complexity are also listed in Table 1.

3.3.5 Model Score We compile the above into a length-4 vector representing the model's performance on the phase. The first number is the model accuracy, which of course is a number between zero and one. The second value is count of near misses predicted by the statistical model, divided by the number of near misses that the base model emitted. The third number is the size of the model as a fraction of base model, and the final number is the number of comparisons required as a fraction of the base model. The score, then, is the distance of this vector from the ideal, which would be, $[1, 1, 0, 0]$. A lower score will indicate a superior ranking.

3.4 Model Swapping

In Figure 1, we showed an abstract picture of what model training and swapping should look like. For this work, due to the simplicity of the models chosen, we have used a reduced version of that diagram for our algorithm. We simply train for two intervals, and do not make use of a *GiveUp* state. We found that this time period was long enough for the learned parameters in our models to stabilize, and that the resulting accuracy was reasonable for the total run.

Now that we've seen the components of our simulation, it is time we take a look at how well this methodology works. In our tests, the time spent with the base model is five intervals per phase to identify, plus two intervals per phase to train, plus the number of any uncategorized phases. This means that for the results presented in Section 4, the swapped models are used for over 90% of the simulation.

4 Results

4.1 Methodology

4.1.1 Simulator Our technique for model swapping was implemented in *SVE-Cachesim*, which is an in-order, Python-based cache simulator developed for [8]. While a simple model such as this ignores some parts of the cache system, such as the effects of pre-fetching, by focusing on a component in the middle of the system, the *L1D*, we were able to study the downstream effects introduced by our models.

4.1.2 Simulated System The system under examination is the a 3-level cache hierarchy 4B words and 8 words per line in the *L1D* and 16 words per line in the *L2* and *L3*. It has a 16 KiB 8-way set associative *L1D*, a 256 KiB 8-way set associative *L2*, and a 1 MiB 32-way set associative *L3*. All levels use a least recently used replacement policy. Computational cores are not modeled in this work; we merely take the memory addresses from the trace and send them to the L1 cache.

4.1.3 Trace To test our phase-analysis methodology, we used a micro-benchmark called Meabo¹. Meabo allows us to run a program with pre-identified phases, which makes it easy for us to visually confirm that our phase analysis is working. We chose three phases from Meabo, *Phase 1: Floating-point & integer computations with good data locality*, *Phase 4: Vector addition*, and *Phase 10: Random memory accesses*. These were chosen as we expected them to vary in their utilization of the cache hierarchy.

We made a few changes to the code, which are intended as markers to ensure that we have correctly identified each specific phase. First, we added a loop to Meabo so that all phases will be run three times instead of just once, and second, we added a marker phase between each phase so we can see which phases represent computation and which belong to other program artifacts artifacts, such as data initialization.

The trace is collected by running Meabo with a single thread on an Intel Xeon Haswell E52690-v3 and capturing all memory references with DynamoRIO *memtrace* [6]. The trace is roughly three million references long.

¹<https://github.com/ARM-software/meabo>

4.1.4 Model Training As mentioned in Section 3.4, we use a simple model swapping criterion in this paper: one or more models are trained for the first two intervals in a phase, and then the best one is selected. We observed that the training parameters for all of our selected models stabilized quickly, and thus we did not need to train for more time. We do not use a criteria to declare training a failure in this work.

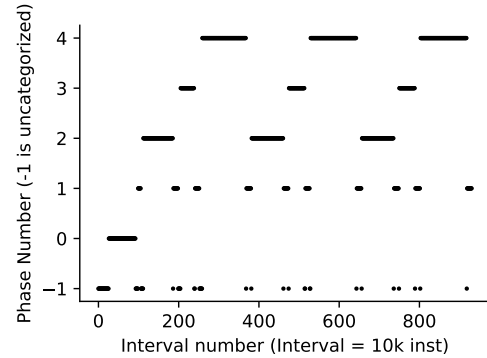


Figure 4: Meabo phases: Here we have plotted the interval number of the program along with its corresponding identifier assigned to it through phase detection. These images shows our marker phase, phase one, being run after every Meabo computational phase, phases two, three, and four.

4.2 Phase Analysis

First, we should examine that our phase analysis code is working as intended. In Figure 4, we plot the phase identifier (*id*) for every interval in the simulation. At the end of every interval, which are each 10,000 instructions, the phase detector will attempt to identify the interval by comparing the signature to those it has encountered previously. If it is successful, an *id* is assigned, otherwise the interval is labeled as (-1).

The trace we collected ran each Meabo kernel three times. Looking at Figure 4, it is clear then that phases two, three, and four must be the phases from Meabo, and that phase one must be our marker phase that runs between Meabo computational phases. For the rest of the paper, we will refer to these phases by the number assigned to them by our phase detection algorithm, but we will include a description of the computation (e.g. *High Locality*, *Vector Add*, *Random*, etc.) so that it will be clear to which we refer.

Now that we've demonstrated empirically that our phase detection methodology is sound, and capable of driving our online model selection process. We can now evaluate how accurate each model is.

4.3 Prediction Accuracy

We would like to compare the accuracy of the each model we have chosen so that we can get an idea of how they measure up against each other. In Figure 5, we measure the accuracy of each model for each phase. Accuracy is defined as the number of properly classified accesses, which means the number of accesses correctly predicted

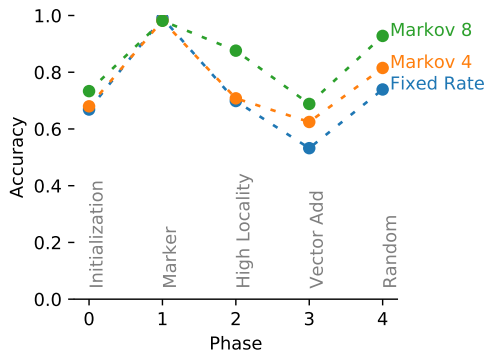


Figure 5: Accuracy per phase: We run our simulation but restrict it to using a single alternate model (as opposed to letting the simulation choose between them), so that we can view their relative performance. We average the accuracy over 10 runs. These values are also shown in Table 4, along with the standard deviation in Table 5, which was too small to plot.

to be hits or misses, divided by the total number of accesses to the *L1D*. We consider the truth to be the standard cache model running with no model swapping.

This plot shows us that the 8-State Markov model is the best for all but the marker phase, Phase 1. The marker phase is essentially all read hits, and thus trivial to predict. The gaps between the model performance on the Meabo phases (Phases 2, 3, and 4) give us hints as to what type of information is important for modeling each phase. In phase 2, the tiny gap between Markov 4 and the Fixed Hit Rate model shows us that knowledge of whether the access was read or write (and whether the last access was read or write) is not important, at least not on its own. This information, however, is useful in both Phases 3 and 4. As we see from the performance of the 8-state Markov model, even the relatively small amount of locality information is enough to achieve 90% accuracy in Phases 2 and 4.

While promising, this data is an aggregate over the entire run, with multiple runs of each phase aggregated together. We will break this data down to examine it further.

4.4 Prediction Accuracy Over Time

While it is useful to look at data aggregated per-phase, it leaves us unable to ask question about how the accuracy changes over time, particularly at phase boundaries. Figure 6 displays the accuracy per-interval, as opposed to the per-phase data we looked at in the last section.

From the data for Phases 1, 2, and 4, it is clear that the accuracy does change over time. If we reference the phase *id* plot in Figure 4, we can see that these drops in accuracy occur when phases are re-entered. This is perhaps the opposite of the expected behavior - we would expect accuracy to drop at the end of a phase, because the phase detector will not be able to tell the model swapping algorithm that a phase has ended until it identifies an interval that is not a part of the phase. Thus that one interval will have been run with

Table 2: Model Scores (lower is better)

	Phase 0	Phase 1	Phase 2	Phase 3	Phase 4
Fixed Rate	0.6875	0.0626	0.6035	0.9273	0.5188
Markov 4	0.6519	0.0781	0.6070	0.7479	0.3842
Markov 8	0.5802	0.2253	0.3313	0.6659	0.2668

a model trained for a different phase. However, as we only see 2 spikes on Phase 2, it must be that this is happening on phase re-entry, not phase exit. It is likely that phases exhibit different memory behavior upon re-entry, such as a higher number of cache misses. This behavior is likely not captured by our models that are trained on data from several intervals into a phase.

On the other hand, it is quite promising that these lines appear to be flat. This means that the accuracy is not getting worse as time passes. In other words, deleterious state doesn't accumulate or compound over time.

Now that we have looked at how the models work in regards to the *L1D* statistics, we should take a look at how the rest of the simulation is affected.

4.5 Locality of Misses

In Figure 7 we have plotted the reuse distance histograms for the cache accesses that miss in the *L1D* cache and thus go to the *L2* cache, which has not been swapped out for a simpler model. It will be important that these match the base model so that we do not disturb the *L2* behavior too much. Across the top of the plot is the reuse data for the base cache. These are the shapes that we would like our statistical models to learn.

The locality of phases 0 and 1, program initialization and the marker phase, seem difficult to capture with our models. Phase 0 has a large spike that no models capture, and phase 1 is particularly hard for the Fixed Rate Model. Thankfully, these do not take up a large percentage of the simulation. Moving to the Meabo phases, we see that the 8-State Markov model shows a strong ability to mimic the reuse distances of the base model, especially compared to the other two models. In Phase 2, both the Fixed Hit Rate and the 4-State Markov produce large spikes in the histogram that should not be there, whereas the 8-State Markov model does not. In Phase 3, the 8-State model seems to do best at reproducing the right side of the distribution, while they all seem to have trouble with the left. And finally, in Phase 4, it again seems that the 8-State model is best at reproducing the qualitative properties of the reuse distance distribution.

The takeaway here is that in cases where the locality of the references going to the *L2* does not matter, for instance in some phase with a very high *L1D* hit rate or a phase where *L2* and *L3* stats are not important to the simulation designer, it may be alright to use a model like the Fixed Hit Rate or 4-State Markov model. However, in cases where the spatial and temporal locality matters, as is typically the case, the 8-State Markov model will work best (out of the models we have to select from).

Now that we have examined how the individual models perform, we should examine how they are ranked.

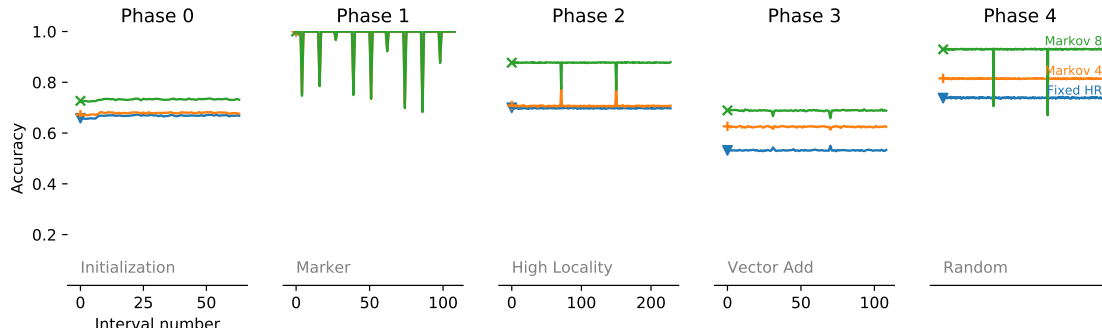


Figure 6: Accuracy per interval: We can view the accuracy that each model achieves on each interval of the trace. In actuality, the trace moves from phase to phase, but for this plot, we show each execution of each phase as a single line, to demonstrate what happens to the accuracy of each model as time moves on. We observe that that accuracy does not deteriorate over time, but that there are dips in accuracy occurring upon phase re-entry. For example, Phase 2 has two reentry points (as shown in Figure 4), and we see corresponding dips when accuracy is plotted sequentially here

Table 3: Percent Change in Cache Stats (10 Runs)

	L1 Hits	L2 Hits	L3 Hits	Cycles
Base	7.69e+06	7.78e+05	2.53e+05	1.37e+08
Fixed Hit Rate	-0.07%	54.11%	-71.11%	-27.91%
Markov 4	-0.37%	46.10%	-52.14%	-23.13%
Markov 8	-0.19%	12.13%	-4.18%	-7.59%
ALL	0.07%	10.12%	-4.65%	-7.99%

4.6 Model Selection

We described our model selection criteria in Section 3.3. In Table 2 we see which models actually won out, based on the criteria of accuracy, locality, complexity, and size. Due to the ability of the 8-state Markov model to represent locality, as well as its high accuracy, we see that it wins for the all but one phase. However, due to the simplicity of phase 1 and the size of the Fixed Hit Rate model, the 8-state model loses here. Thus, during a run of the simulation where the simulator is allowed to choose the best ranking model, the 8-state Markov model is always chosen except in phase 1. In subsequent sections, we refer to this as *ALL*, as the simulator trains all 3 models and chooses the best for each phase.

4.7 Overall Simulation Statistics

In the last section, we saw that due to the performance of the 8-State Markov model, it is chosen by the model selection algorithm for every phase except for the marker phase, Phase 1. In this section, we'll look at how the overall simulation statistics change when using each model, including the *ALL* model, as just described. This data is in Table 3.

First off, we see that every model is able to accurately match the hit count of the *L1D* cache. This is expected, as each model need only produce the same number of hit and misses as the base model. This is obviously the case of the Fixed Hit Rate model. This is also expected of the Markov models, as we expect them to spend the same amount of time in each state as the training data, so they are expected to produce the proper number of hits and misses.

It is good, however, to know that our slightly modified Markov model, which is able to restrict the state it moves to in order to predict locality, does not break this behavior. As we move to look at the rest of the stats, things start looking grim for the Fixed Hit Rate and 4-State Markov models. These models do not encode any locality information, and thus the stream of misses that go to the *L2* have far different locality properties than the base detailed model. Thankfully, it seems easy enough to correct for this. The 8-State Markov model greatly reduces the error in the *L2*, and *L3* hit counts, and correspondingly the total number of execution cycles as well.

As for the simulation where we automatically selected models, *ALL*, the stats are not too different from the 8-State Markov model. This is to be expected, as we saw in Section 4.6, the 8-State Markov model is chosen for every phase except for Phase 1, which was trivial to predict. While this choice does introduce extra error, it will be up to future simulation designers to decide if error like this is acceptable for their purposes. The fact that the Fixed Rate Model was used at all, and did not disturb overall simulation statistics too much should be seen as a win.

Overall, our accuracy in cache hits and simulation cycles indicate that this model swapping methodology has merit, and shows promise for speeding up larger simulations.

5 Related Work

The topic of how to choose between models is a well studied topic with an established body of literature [13, 16, 26]. Our work builds on these classic works by using known modeling techniques such as the Markov model [11]. A model selection survey by [21] describes more recent methods of model selection, in which they describe a posterior predictive criteria method related to our distance metric. We differ in that we used summarized criteria in the selection process as well as raw model performance.

As model validation is closely related to model selection, our work is loosely related to empirical response surface evaluation [5]. Our work, instead of relying on complex multi-dimensional, time-series surfaces, uses a summary statistic over each phase which is compared to an ideal using a distance metric (§ 2). Multiple

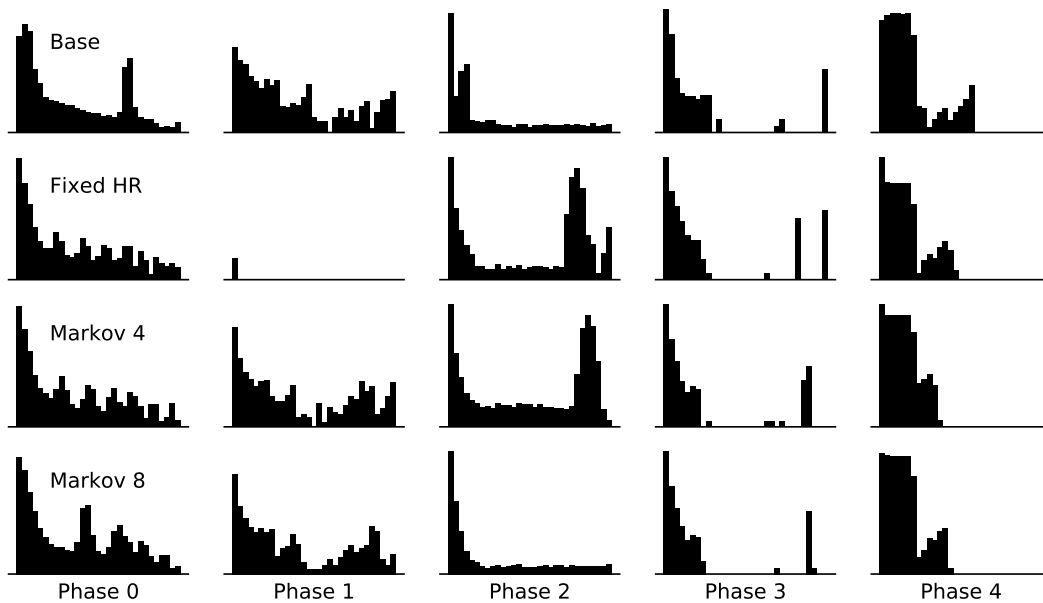


Figure 7: Reuse distance histograms: We calculate the reuse distance for every access to the $L2$ cache and plot this for a single run of the simulator. We cut the x -axis at reuse distance = 500 and the y -axis at count = 125,000 as this contains the overwhelming majority of the volume of the chart. Log y -axis. This shows us that the 8-state Markov model is able to model the locality properties of the base cache much better than the other two statistical models.

related fields from Operations Research [17] and agriculture [12] rely on model validation, typically through physical or empirical observation. Model validation and simulation [23, 24] are often discussed in terms of model cost (e.g. execution time) and model confidence (i.e., does the model work). We differ from the model validation described by [24] in that due to our intended automated approach, we rely almost exclusively on “scoring” vs. other more intensive approaches. Our thesis here is that we can constrain the impact of error by modeling behavior by phase.

While prior works provided the building blocks to enable our online model swapping proof-of-concept, we believe we are the first to demonstrate all such features combined, that is: phase detection, model selection, and model swapping.

6 Conclusions and Future Work

This work demonstrates the potential for online model swapping to speed up simulation while minimizing loss in accuracy. Our proof of concept shows that simple statistical models can faithfully represent phases of execution based on the realistic kernels in Meabo. As Figure 8 shows for the Meabo kernels, we can provide computer architects a new tool to quickly explore trade-offs between accuracy and the size or complexity costs of different models. We are excited by this result, but further work is yet to be done:

- **Swapping criteria and more statistical models** We have preliminary data for a subset of the applications in the PARSEC benchmark suite [3] that show the potential for this technique to apply to more datasets than just Meabo. Our models achieve greater than 90% accuracy in predicting $L1D$ hits on many of the phases, but produce overall cycle errors in the range of 15% to 35%. We believe this will be remedied by a combination of (1) choosing not to swap models in low accuracy phases and (2) the addition of more statistical models, such as [19].
- **SST Integration** This project was inspired by a desire to speed up simulations using the Structural Simulation Toolkit (SST) [22]. As such, an important next step will be the implementing our work there. This will allow our methodology to be used in more realistic simulations that include prefetching and cache coherency, and bring this work to a wide audience.
- **More components** We used the $L1D$ as a starting point for us to explore model swapping. However, we designed the methodology so that it would be general enough to apply to other components in a simulation. We aim to swap out components such as other parts of the cache, parts of the network, or even pieces of the core model.

The methods in this paper will serve as a starting point for future work in model swapping. With the main components the technique

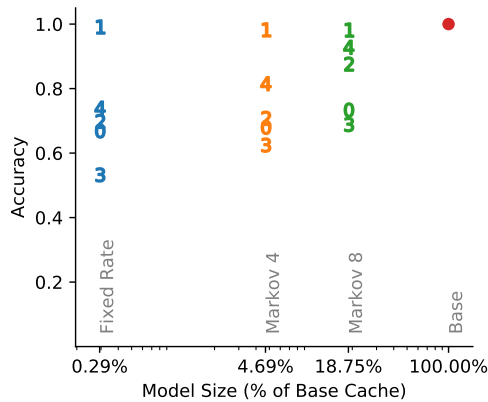


Figure 8: Accuracy as a function of model size: Here we visualize the trade-off between accuracy and model size, with each simpler model giving us a broader range of performance for the phases studied in this work. Our research enables simulation designers to explore this space, and other related spaces, for the purposes of faster iteration in design.

described here, and an implementation to serve as a starting point, we know that further exploration will yield useful tools to accelerate the architectural simulation.

Acknowledgements

This work used the Hive cluster, which is supported by the National Science Foundation under grant number 1828187. This research was supported in part through research cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology, Atlanta, Georgia, USA.

Additionally, this work was partially supported by the National Science Foundation under Grant No. 1710371, and was partially completed while the first author was an intern at Arm Research.

References

- [1] Ayaz Akram and Lina Sawalha. 2019. A survey of computer architecture simulation techniques and tools. *Ieee Access* 7 (2019), 78120–78145.
- [2] Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 35, 2 (2002), 59–67.
- [3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.
- [4] 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.
- [5] George EP Box and Norman Richard Draper. 1987. *Empirical model-building and response surfaces*. Vol. 424. Wiley New York.
- [6] Derek L. Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. MIT, USA. AAI0807735.
- [7] Andrew A Chien and Vijay Karamcheti. 2013. Moore’s law: The first ending and a new beginning. *Computer* 46, 12 (2013), 48–53.
- [8] Miguel Cruz, Daniel Ruiz, and Roxana Rusitoru. 2019. Asvie: A Timing-Agnostic SVE Optimization Methodology. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. 9–16. <https://doi.org/10.1109/ProTools49597.2019.00007>
- [9] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. 1974. Design of ion-implanted MOSFET’s with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268.
- [10] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (Anchorage, Alaska) (ISCA ’02)*. IEEE Computer Society, USA, 233–244.
- [11] Paul A Gagnic. 2017. *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons.
- [12] Hugh G Gauch Jr. 1988. Model selection and validation for yield trials with interaction. *Biometrics* (1988), 705–715.
- [13] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.
- [14] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. 2012. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of ?, 91–94.
- [15] Michael Hind, V. T. Rajan, and Peter F. Sweeney. 2003. *Phase Shift Detection: A Problem Classification*. Technical Report. IBM Thomas J. Watson Research Division.
- [16] Raj Jain. 1991. *The art of computer systems performance analysis*. Vol. 182. John Wiley & Sons Chichester.
- [17] Maurice Landry, Jean-Louis Malouin, and Muhittin Oral. 1983. Model validation in operations research. *European Journal of Operational Research* 14, 3 (1983), 207–220.
- [18] Gordon E Moore. 1965. Cramming more components onto integrated circuits. *Electronics* 38, 8 (1965).
- [19] K. O’Neal and P. Brisk. 2018. Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 763–768.
- [20] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC’11)*.
- [21] Juho Piironen and Aki Vehtari. 2017. Comparison of Bayesian predictive methods for model selection. *Statistics and Computing* 27, 3 (2017), 711–735.
- [22] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 37–42. <https://doi.org/10.1145/1964218.1964225>
- [23] Robert G Sargent. 1979. Validation of simulation models. In *Proceedings of the 11th conference on Winter simulation-Volume 2*. IEEE Press, 497–503.
- [24] Robert G Sargent. 2005. Verification and validation of simulation models. In *Proceedings of the 37th conference on Winter simulation*. winter simulation conference, 130–143.
- [25] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices* 37, 10 (2002), 45–57.
- [26] William J Stewart. 2009. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, Princeton, NJ.

A Phase Detection Algorithm

A description of the phase detection algorithm that we use exists in [10], but it does not include an implementation. We include pseudo-code here to aid the reader in understanding the implementation.

```

threshold = 0.5
interval_len = 10000
sig_len = 1024
drop_btis = 3
sig = new_bitvector(sig_len)
last_sig = new_bitvector(sig_len)

def diff(sig1, sig2):
    tmp1 = sig1 ^ sig2
    tmp2 = sig1 | sig2
    return popcount(tmp1) / popcount(tmp2)

```

```

def hash(sig):
    tmp = sig >> drop_bits
    return builtin_hash(tmp) >> (64 - log2(sig_len))

def phase_detector(ip):
    sig[hash(ip)] = 1
    count += 1

    # If we are on an interval boundary, determine
    # the phase and notify listeners
    if count % interval_len == 0:
        if diff(sig, last_sig) < threshold:
            stable += 1
            if stable >= stable_min and phase == -1:
                phase_table.append(sig)
                self.phase = len(phase_table) - 1
        else:
            stable = 0
            phase = -1

        # Check if we have entered a phase we
        # have seen before
        if len(phase_table) > 0:
            similar = []
            for s in phase_table:
                similar.append(diff(sig, s))
            best = imax(similar)
            if similar[best] < threshold:
                phase = best

    last_sig = sig
    sig = new_bitvector(sig_len)

    notify_listeners(phase)

```

Listing 1: The pseudo-code for our phase detector implementation. On every memory reference, the IP is shared with the phase_detector function, which updates a signature. At the end of an interval, this signature is compared with the previous one to see if execution is stable.

B Markov Prediction Algorithm

This pseudo-code corresponds to the algorithm depicted in Figure 3.

```

last_state # the last state
counts    # (N x N) matrix where (i, j) is
          # the number of times state j followed
          # state i during training

# The prediction function returns the next state
# The restrict parameter is a tuple with allowed

```

```

# states
def predict(restrict):
    tmp = counts[:, restrict]
    tmp = normalize_rows(tmp)
    pred = prefixsum_rows(tmp)

    r = rand()

    for i in range(len(restrict)):
        if r < pred[last_state, i]:
            next_state = restrict[i]
            break

    last_state = next_state
    return next_state

```

Listing 2: The algorithm for prediction with the 4 state Markov model. Care must be taken to only return a state that corresponds to the incoming request. This is done with the restrict parameter, which is used to restrict the states that the model can move to. In practice, the restricted prediction matrices are memoized if the counts matrix has not changed.

C Accuracy Data

This data is plotted in Figure 5, and is included here for readers curious about the variance between runs.

Table 4: Mean Per-Phase Accuracy (10 Runs)

	Phase 0	Phase 1	Phase 2	Phase 3	Phase 4
Fixed Rate	0.67	0.99	0.70	0.53	0.74
Markov 4	0.68	0.98	0.71	0.62	0.82
Markov 8	0.73	0.98	0.88	0.69	0.93
All	0.73	0.99	0.88	0.69	0.93

Table 5: Std.Dev. in Per-Phase Accuracy (10 Runs)

	Phase 0	Phase 1	Phase 2	Phase 3	Phase 4
Fixed Rate	0.0004	0.0000	0.0002	0.0003	0.0002
Markov 4	0.0004	0.0001	0.0002	0.0004	0.0001
Markov 8	0.0006	0.0001	0.0001	0.0005	0.0001
All	0.0005	0.0000	0.0002	0.0004	0.0001