Learning Architectural Cache Simulator Behaviour

Pranjali Jain*, Meiru Han†, Zhizhou Zhang‡, Brandon Lee*, Jonathan Balkind*
*UC Santa Barbara †University of Pennsylvania ‡Uber Technologies
*{pranjali_jain, brandon_lee, jbalkind}@ucsb.edu, †meiru@seas.upenn.edu, ‡zzzhang@uber.com

Abstract

Modern applications exhibit memory access patterns with complex spatial and temporal relationships. Traditional architectural simulators utilized to evaluate these applications are highly sequential in nature, particularly for stateful components like caches. In this paper, we present an innovative approach to cache simulation by reframing the problem from a deep learning perspective. We exploit the fact that memory access traces in any part of a processor design can be represented as two-dimensional heatmaps. Our key insight is that the behaviour of a cache acts as a filter on these heatmap images which can be learned as a function using deep learning techniques. Leveraging this observation, we introduce CacheBox, a framework that employs a Generative Adversarial Network (GAN) to learn and replicate the filtering behaviour of caches using memory access heatmaps. We demonstrate that CacheBox effectively generalises across multiple state-of-the-art benchmarks, various cache configurations, different cache hierarchy levels, and even alternative microarchitectural structures with high accuracy. We also show that CacheBox enables highly parallelized inference, allowing for simultaneous processing of multiple memory access heatmaps.

1. Introduction

Modern processors generate complex patterns of memory accesses that reflect both program behaviour and architectural influences. For caches, these patterns exhibit temporal and spatial relationships that determine system performance. While traditionally analyzed through deterministic and sequential architectural simulators, these patterns share characteristics with other domains where deep learning has excelled at capturing complex relationships.

Specific contexts have driven bespoke solutions for architectural simulation, such as for manycore processors, where accuracy may be traded for simulation throughput due to the blow-up in instruction counts alongside core counts [10]. Such issues have also led to statistical techniques for simulation like SimPoints [25], which enable architects to focus only on applications' most characteristic phases. Even with these advanced techniques, the underlying mechanism remains sequential simulation.

In this paper, we propose a novel alternative to cache simulation, reconceptualized through the lens of deep learning, as shown in Figure 1. We exploit the fact that memory traces can be lossily represented as 2D heatmap images. Thus, the trace on any memory bus can be represented as a heatmap image. Our key insight is that since caches act as filters over memory access patterns (filtering an input stream of cache accesses into an output stream of cache misses), this filter behaviour can be represented as a transformation over memory access heatmap images. This opens up an intriguing possibility: can we leverage deep learning based image-to-image translation, to learn and reproduce cache behaviour?

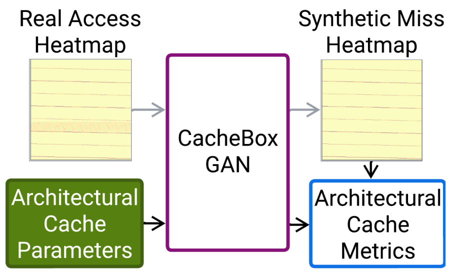


Figure 1: Design and workflow of CacheBox

To this end, we propose CacheBox (CBox), a framework which employs a Generative Adversarial Network (GAN) to learn cache simulator behaviour. To our knowledge, CacheBox is the first tool that uses 2D heatmaps to learn microarchitectural behaviour. By representing memory traces as heatmap images and cache specifications as model parameters, we enable CBox to learn complex temporal and spatial locality patterns that determine cache performance. CBox can easily learn complex filters that depend on relationships between pixels spaced far apart in a heatmap image. CBox learns long-term cache behaviour over heatmaps representing around 50,000 memory operations per image.

We show that CBox can be simultaneously trained on multiple cache configurations with high accuracy, enabling efficient, parallel evaluation of different cache configurations. CBox demonstrates remarkable versatility by effectively generalizing across diverse benchmark suites, accurately modelling multiple cache hierarchy levels (L1, L2 and L3), adapting to completely novel cache configurations, and can be easily extended to model other microarchitectural features such as prefetchers. In our evaluation, we ensure that all benchmarks seen during inference are entirely unseen in training. We deliberately adopt this conservative methodology of testing only on unfamiliar applications to emphasize CBox's generalizability.

CBox achieves high accuracy in predicting cache miss patterns, with average absolute percentage difference in hitrate prediction of 3.05% across diverse benchmark suites, including SPEC, Ligra, and Polybench. The framework demonstrates remarkable adaptability, successfully modelling behaviour for cache configurations absent from training data with average absolute hitrate differences of 1.26–3.28%. CBox also parallelizes through batching, achieving 2.4× speedup with batch size 32 compared to sequential inference.

Our approach demonstrates that deep learning models can effectively learn and generalize cache behaviour across different applications, cache configurations, and microarchitectural features. This work opens up new possibilities for architectural simulation by showing how modern machine learning techniques can be applied to computer architecture problems. Our contributions are as follows:

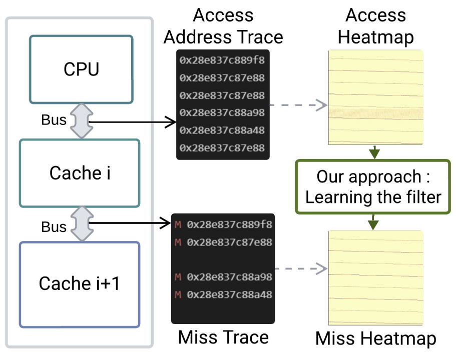
- CacheBox (CBox): a GAN-based framework that accurately models cache filtering behaviour utilizing heatmaps of cache memory access patterns.
- CB-GAN: a modified image-to-image GAN, that can incorporate cache parameters along with input heatmap images of memory access patterns.
- We show that CBox generalises across benchmark suites, cache configurations, cache hierarchy levels, and even to prefetching with high fidelity and without requiring retraining per variant.
- We show that CBox enables parallelized inference with 2.4× speedup at batch size 32, enabling rapid design space exploration.

2. Motivation and Key Idea

Memory address traces provide rich information about programs' behaviour. Architectural cache simulators often rely on such traces combined with their execution model to establish whether a given memory access was a hit or miss in a particular cache level. Both the memory access address traces (as input) and the hit-and-miss traces (as output) are represented sequentially as text files.

These traces often become cumbersome to use and analyse. As an alternative, heatmaps [6, 13] offer an intuitive visualization of trace data. For example, Hashemi et al. [13] use neural networks to identify memory access patterns and utilize heatmaps to understand the underlying decision of their neural network. Heatmaps have also proven valuable for architectural studies and analysis. Dangwal et al. [6] used heatmaps to represent and analyse memory access patterns for privacy-preserving trace wringing.

We observe (Figure 2) that buses between components can have their memory access traces encoded as heatmaps, capturing both their temporal evolution and behavioural characteristics. Our key insight is that a cache acts as a filter on the heatmap of the cache accesses (before the cache) to produce the heatmap of the cache misses (after the cache), as shown in Figures 2 and 3. This enables us to transform the problem of cache simulation from a sequential, text-based analysis to a visual, learning-based approach. We transform the memory address trace entering cache level i into an access heatmap, while converting the corresponding miss trace exiting the same cache level into a miss heatmap. Thus we fundamentally reframe the cache as a filter over heatmap images, enabling us to leverage neural networks to model these complex spatial and temporal filtering behaviours.

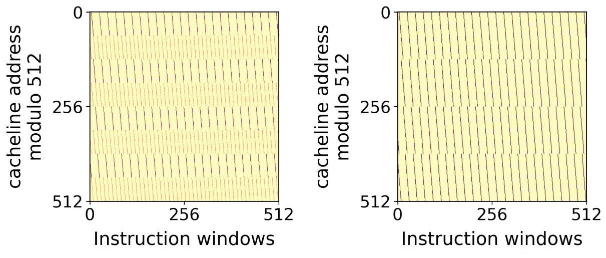


```
0x28e837c889f8
0x28e837c87e88
0x28e837c87e88
0x28e837c88a98
0x28e837c88a48
0x28e837c87e88
```

```
M 0x28e837c889f8
M 0x28e837c87e88

M 0x28e837c88a98
M 0x28e837c88a48
```

Figure 2: Caches act as filters over memory access traces to produce traces of memory accesses that miss in that particular cache level. CacheBox learns this filter behaviour.



(a) Access Heatmap (b) Miss Heatmap

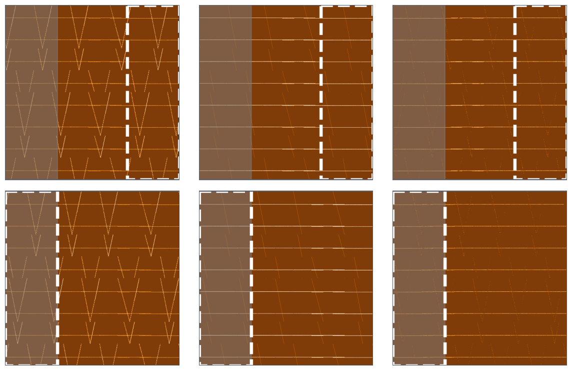
Figure 3: Access (a) and Miss (b) heatmaps for a benchmark from Polyhedral benchmark suite for L1D cache. Both have dimensions of 512×512 and 100 accesses per column, for a 51,200 memory access window. Miss heatmaps represent the L1D cache misses within the window.

3. Design

Here we detail the design of CBox, as shown in Figure 1. We describe the representation of 2D memory access heatmaps, which are utilized by CBox's Generative Adversarial Network (CB-GAN). CB-GAN predicts the hits and misses based on the memory accesses in the input heatmap, generating a miss heatmap which can be used to calculate architectural metrics like hit rate.

3.1. Heatmap Representation

We generate a heatmap from memory access addresses by projecting the memory access trace onto a fixed-size modulo-mapping of the memory space. The heatmap's x-axis is the instruction count over time, and the y-axis maps the memory addresses. We perform modulo operations on each memory address using the heatmap's vertical size. Likewise, we group the instructions over time into bins within windows of a specified number of instructions. Despite this binning, the horizontal dimension of the heatmap can rapidly expand with instruction count, making the heatmap very wide. We divide this larger heatmap into smaller ones, each capturing a smaller number of instructions. Finally, each pixel indicates the total number of memory accesses to that particular modulo-memory address during one instruction window.



(a) Real Access heatmap (b) Real Miss Heatmap (c) Synthetic Miss Heatmap

Figure 4: Consecutive (top and bottom) pairs of Real Access, Real Miss, and Synthetic Miss heatmaps. The first 30% of each heatmap in the second row overlaps with the last 30% of the previous one (in the first row).

An example heatmap is shown in Figure 3a, representing a subset of the memory accesses of a Polyhedral benchmark. The y-axis represents the memory addresses modulo 512, and the x-axis represents 51200 memory accesses, specifically 512 instruction windows of 100 instructions each. Each pixel represents memory accesses to that particular 512-modulo-memory address in a 100-instruction window.

We generate two sets of heatmaps for a cache. The first set, the **access heatmaps**, show the memory accesses input to the cache; the total of all the pixel values gives the total number of memory accesses. The second set, the **miss heatmaps**, show the memory accesses that incur misses when the cache is accessed; the sum of all pixel values gives the total number of misses. Figure 3b shows a miss heatmap generated after the L1 Data cache is accessed. The patterns in the heatmaps indicate memory access trends, including spatio-temporal locality information and the behaviour of the cache based on its configuration.

3.1.1. Utilizing Overlap in Heatmaps. To preserve information regarding local program behaviour, we maintain redundant overlap segments between each pair of heatmaps (Figure 4). This overlap retains spatio-temporal information when splitting the large heatmap, acting as a "warmup" in each heatmap image. We experimented with varying degrees of overlap between successive heatmaps and find that a 30% overlap yields the best results. In this setup, the initial 30% of each heatmap is duplicated in the heatmap immediately preceding it. Figure 4 shows consecutive pairs of Real access, Real miss, and Synthetic miss heatmaps (described further in Section 3.2) representing the overlapped regions. Considering the Real access heatmap pair, the gray-shaded area shows the overlap with the previous heatmap. The white-bordered box in the top heatmap shows the overlap with the subsequent (bottom) heatmap. Thus, the intersection of the gray-shaded area and the white-bordered box in the bottom heatmap shows the part overlapped with the top heatmap.

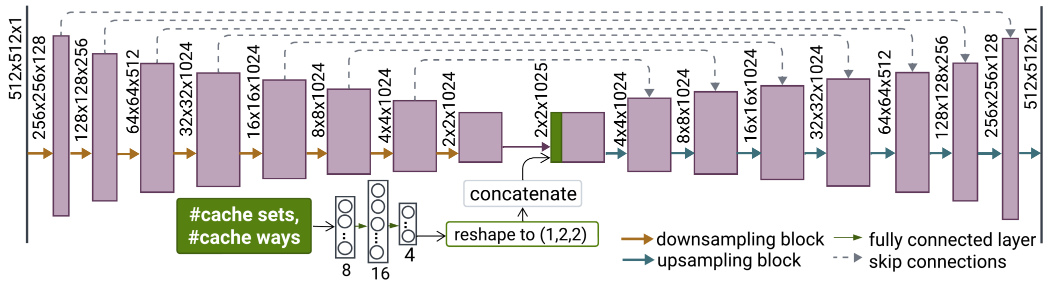
3.2. The CB-GAN Model

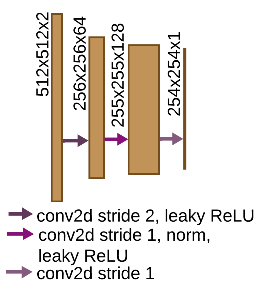
CBox employs a GAN model (CB-GAN) to predict the miss heatmap for a given access heatmap, as shown in Figure 1. CB-GAN is similar to a Pix2Pix model, which is a GAN-based deep learning model for image-to-image translation [12, 14], but is specialised for learning architectural cache simulator behaviour. CB-GAN has a generator and a discriminator that are trained concurrently. The generator produces realistic images, while the discriminator distinguishes between real and synthetic images. It is trained using supervised learning, and produces output images that look similar to the desired targets by learning selective features from the input image. We train CB-GAN on pairs of access heatmaps and miss heatmaps, and we expect the model to generate Synthetic miss heatmaps that closely resemble the Real miss heatmaps.

3.2.1. Model Structure. CB-GAN can recognize spatio-temporal structures in memory access heatmaps and selectively filters out memory accesses that hit in the cache to create the miss heatmap.

Generator. We employ an 8-layer U-Net [23] encoder-decoder as the generator model, with a modification that enables it to take numerical inputs for cache parameters, as shown in Figure 5. The U-Net model (Unet256) has eight down-sampling blocks and eight up-sampling blocks. Each down-sampling block progressively reduces the spatial dimensions of the input image using convolutions, and the up-sampling blocks reconstruct the high-resolution features, gradually restoring the spatial dimensions of the input image. Skip connections in the encoder-decoder structure connect corresponding encoder and decoder layers through which outputs from down-sampling blocks are concatenated to outputs of up-sampling blocks and fed into the next up-sampling block. We provide cache parameters (number of cache sets and ways) as inputs to the generator in CB-GAN along with the access heatmap, to provide additional context to the model regarding the cache architecture, further detailed in Section 3.2.3. They are passed through 3 fully-connected layers, and the reshaped output is appended to the output of the last down-sampling block and then fed into the first up-sampling block.

Discriminator. The discriminator is a PatchGAN [14, 15], which discriminates between real and synthetic images at the granularity of image patches. These patches have dimensions of N×N, where the receptive field N is smaller than the input image dimensions, which enhances the efficacy of the model in capturing local image characteristics. The discriminator takes two inputs: the concatenation of the input image and the target output image marked with a True ground truth label, and the concatenation of the input image and the synthetic image with a False ground truth label. It evaluates each patch in the concatenated inputs as real or synthetic and generates a truth map, which is further processed with the binary cross-entropy loss. The loss value provides feedback to train the discriminator.





512x512x2
256x256x64
255x255x128
254x254x1
conv2d stride 2, leaky ReLU
conv2d stride 1, norm, leaky ReLU
conv2d stride 1

(a) Generator (b) Discriminator

Figure 5: (a) CB-GAN Generator: An 8-layer UNet, modified to take cache parameters as inputs. (b) CB-GAN Discriminator: A 16×16 convolutional PatchGAN. ReLU indicates the Rectified Linear Unit activation function.

3.2.2. Objective Function. The loss function or objective function of CB-GAN is a weighted sum of two components: adversarial loss $\mathcal{L}_{cGAN}(G,D)$ and reconstruction loss $\mathcal{L}_{L1}(G)$. The adversarial loss quantifies the ability of the generator to create realistic images and measures the discriminator's capability to distinguish between real and synthetic images. Reconstruction loss measures how closely the generated image resembles the target output image in terms of the content of the image. The hyperparameter λ is utilized to tune the weights between the two components of the loss function. The generator minimizes the reconstruction loss as well as the adversarial loss, while the discriminator aims to maximize adversarial loss. The objective function G^* can be expressed as follows:

$$G^* = \mathrm{argmin}_G \mathrm{max}_D \mathcal{L}_{cGAN}(G,D) + \lambda \mathcal{L}_{L1}(G) \tag{1}$$

The adversarial loss can be further expressed as follows:

$$\begin{aligned}\mathcal{L}_{cGAN}(G,D) =& \mathbb{E}_{x,y}[\log D(x,y)] \\ &+ \mathbb{E}_{x,z}[\log(1-D(x,G(x,z)))]\end{aligned} \tag{2}$$

where x is the input image, y is the target image and z is random noise. The discriminator makes decisions based on the concatenation of the target output and the input image as well as the synthetic output and input image.

3.2.3. Inputs. CB-GAN takes a pair of images as input while training: the input (access heatmap) and the target output (`Real` miss heatmap) as well as numerical inputs (cache parameters). Each `Real` access-miss heatmap pair represents the same chunk of the program's memory address trace before and after the cache is accessed. Once trained, CB-GAN can take a `Real` access heatmap and cache parameters as input and generate (`Synthetic`) miss heatmaps after a particular cache level is accessed.

Cache Parameters. These are the number of sets and ways in the cache, which provide additional information to the CB-GAN model regarding the size and associativity of the cache. We provide cache parameters during both training and inference. We only show results with these parameters, but others can easily be added.

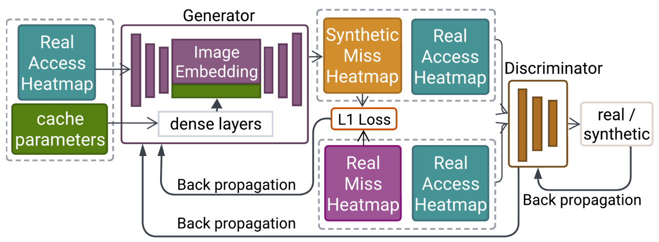


Figure 6: CBox training procedure. The input (`Real` access), output (`Synthetic` miss), and ground truth (`Real` miss) heatmaps are shown along with the cache parameter inputs.

3.2.4. Outputs. CB-GAN generates a synthetic image during both training and inference. The synthetic image resembles the target output image provided as input to the model. The synthetic image is a miss heatmap, further referred to as the `Synthetic` miss heatmap.

4. Methodology

In this section, we outline the implementation details of CBox, including benchmarks employed, heatmap generation, running CB-GAN, and calculating architectural metrics.

4.1. Dataset

For our dataset, we transform memory access traces from *SPEC* (comprises SPEC 2006 [28] and SPEC 2017 [29] from the Third Data Prefetching Championship (DPC3) [7]), *Ligra* [3, 27] and Polyhedral Benchmark suite (*Polybench*) [22]. All traces are collected using Pin [18]. We utilize 189 SPEC, 100 Ligra, and 32 Polybench benchmarks, respectively. We split each benchmark suite 80-20 for training and inference, resulting in 255 training (SPEC: 150, Polybench: 25, Ligra: 80) and 66 testing (SPEC: 39, Polybench: 7, Ligra: 20) benchmarks. Most of our experiments are run on SPEC due to high volume of data for this suite.

Importantly, we always maintain strict separation between training and testing datasets to ensure robust evaluation of the generalizability of CBox. When multiple traces of the same benchmark exist in a suite, we allocate all traces from that benchmark exclusively to either the training or testing set, never distributing across both. During inference, CBox only encounters programs it has never seen in training. Thus, **our reported results reflect the performance of CBox on completely unseen benchmarks** rather than merely unseen inputs to familiar programs, providing a strong assessment of its ability to capture fundamental cache behaviours.

4.2. Heatmap Generation

We generate `Real` (ground-truth) access heatmaps for the benchmark suites described above. We then leverage the hit/miss information for each cache memory access operation of the benchmark traces from Champsim [11] to construct `Real` miss heatmaps, establishing the paired training dataset. We tested various modulo mapping strategies and find that using a modulo of 512 gives the highest prediction accuracy. Similarly, we find that window size of 100 provides a compact, lossy representation, while still retaining key information about cache state and access patterns. Thus, we fix our heatmaps to 512×512 with 100-instruction windows, and 30% overlapping between consecutive heatmaps. We simulate different cache configurations to collect diverse miss heatmaps for training. Each cache configuration is indicated by the number of cache sets and ways, with a fixed 64 byte block size.

We use a bimodal branch predictor and LRU replacement policy in ChampSim to generate the training miss heatmaps. We do not utilize prefetching for any cache level. The Champsim simulation runs 1 billion instructions without warm-up. Heatmap generation is performed single-threaded. However, for inference (most users' case), Pin can dump heatmaps faster than traces and heatmap generation from traces is highly parallelizable. The training data (`Real` access and miss heatmaps of 225 benchmarks from three suites) requires approximately 50 GB of space on disk (versus roughly 60GB for ChampSim traces). However, this is a one-time cost since, once trained, it results in a less than 1GB CBox model.

4.3. CBox GAN Model

We train CB-GAN with a per-pixel L1 reconstruction loss (Equation (1)) balanced by a λ value of 150. In the generator, we incorporate cache parameters after feeding them through 3 fully connected layers, reshaping and concatenating them to the output of the encoder, and finally feeding them into the first decoder block. As a result, in each iteration, the weights of the dense layers linked to the cache parameters are adjusted during backpropagation alongside other weights. Throughout our evaluations (excluding Section 5.4), we employ the Unet256 generator and a 16×16 PatchGAN discriminator, configured with 128 ngf (number of generator filters) and 64 ndf (number of discriminator filters). Finally, the 512×512 input access and miss heatmaps provided to the model have their original pixel values scaled by two. CB-GAN utilizes random batching during training.

4.4. Hit Rate Calculation in CBox

CBox generates `Synthetic` miss heatmaps for a given cache level which can be used to calculate miss rate. The sum of the pixel values in a `Synthetic` miss heatmap is equivalent to the total misses incurred in the window represented by the heatmap. Similarly, the sum of the pixel values in the `Real` access heatmaps is the total cache accesses. With these, we can calculate the hit or miss rate for a given `Real` access and `Synthetic` miss heatmap pair. The overlapped region should be accounted for only once to calculate the correct miss rate for the entire trace.

To measure accuracy of CB-GAN for a given benchmark, we measure the hit rate of the generated `Synthetic` miss heatmaps and compare it with the hit rate of ground truth access-miss heatmap pairs. The **average absolute percentage difference** represents the mean of the absolute percentage differences between the *predicted* and *true* hit rates for the specified benchmarks. This captures the architectural significance of prediction errors across all hit rate regimes — where a 5% deviation has consistent meaning whether the actual hit rate is 10% or 90%. This metric ensures balanced evaluation across diverse cache behaviours and configurations.

5. Evaluating the CacheBox

Here we showcase the capability of CBox to learn cache behaviour in different settings. We address the following key research questions:

1) **RQ1:** How well does CBox generalize to previously unseen benchmarks across benchmark suites?
2) **RQ2:** Can CBox accurately model diverse cache configurations?
3) **RQ3:** Does CBox effectively predict behaviour for cache configurations absent from training data?
4) **RQ4:** How does CBox perform across different cache hierarchy levels?
5) **RQ5:** What advantages does parallelized inference offer to CBox?
6) **RQ6:** How accurately does CBox reproduce cache response compared to traditional simulation?
7) **RQ7:** Can CBox capture the behaviour of other microarchitectural elements such as prefetchers?

5.1. RQ1: Adapting to unseen applications

Accurately modelling cache behaviour across diverse application domains is challenging. We investigate CBox's ability to capture cache dynamics across heterogeneous benchmark suites and learn generalizable cache access patterns. We train first on a L1 Data cache with 64set-12way cache configuration. Each training batch contains a mixture of heatmap images from the SPEC, Ligra, and Polybench suites. Our experiments reveal that CBox can generate `Synthetic` miss heatmaps with an average absolute percentage difference of 3.05% between *predicted* and *true* hit rates, as shown in Figure 7.

Key Takeaway: CBox can extract and generalize cache behaviour across fundamentally different computational workloads, from scientific computing and graph algorithms to standard performance benchmarks.

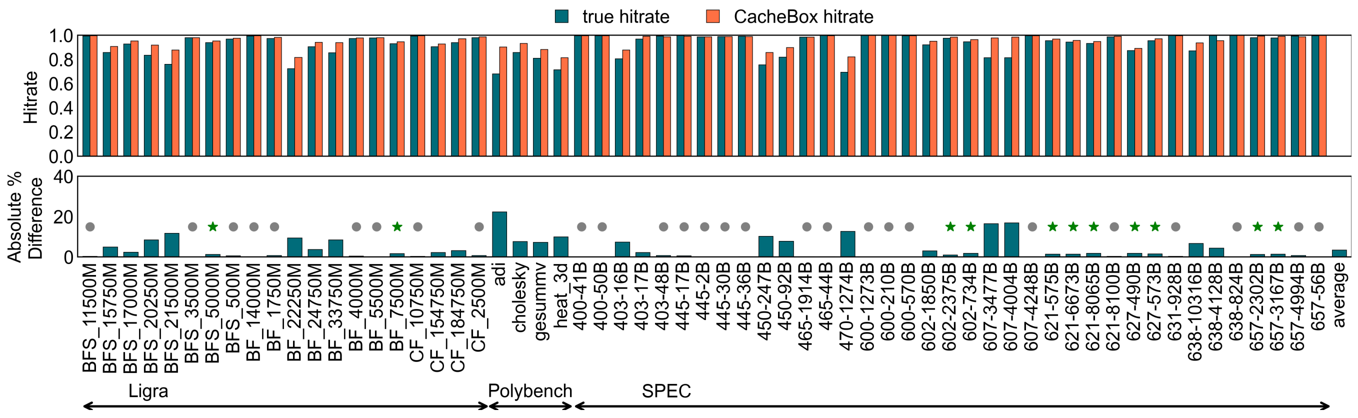


Figure 7: ***(top)*** *True* and *predicted* hit rates for CB-GAN trained on SPEC, Ligra, and Polybench for a 64set-12way L1 cache. ***(bottom)*** Absolute percentage difference in hit rates for the same. Black dots and green stars on the bars indicate absolute percentage differences of $<1\%$ and 1–2%, respectively. The average case absolute percentage difference is 3.05%.

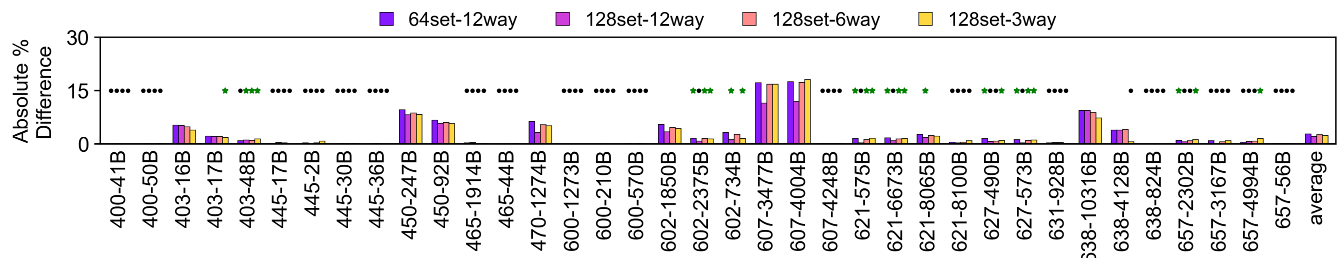


Figure 8: Absolute percentage difference in hit rates for SPEC on four L1 cache configurations. The average is 2.79%, 2.06%, 2.59%, and 2.46% for 64set-12way, 128set-12way, 128set-6way, and 128set-3way cache configurations, respectively. Black dots and green stars on the bars indicate absolute percentage differences of $<1\%$ and 1–2%, respectively.

5.2. RQ2: Supporting diverse cache configs

Cache parameters significantly impact memory system behaviour, but CBox generalizes across parameters. We train a single CB-GAN model on SPEC benchmarks for four different L1 Data cache configurations with the access-miss heatmap pairs for all cache configurations batched together. The CB-GAN model thus learns to distinguish between heatmaps of different cache configurations via the cache parameter inputs. Results are shown in Figure 8. *True* hit rates differ slightly across cache configurations for a given benchmark trace, and *predicted* hit rates follow similarly. The average case exhibits consistently low absolute percentage differences of 2.79%, 2.06%, 2.59%, and 2.46% for 64set-12way, 128set-12way, 128set-6way, and 128set-3way L1 cache configurations, respectively.

Key Takeaway: We do not need to train n independent CB-GAN models for n different cache configurations. A single CB-GAN model trained on different cache configurations can accurately predict `Synthetic` miss heatmaps for each cache configuration in the set.

5.3. RQ3: Generalizing to unseen cache configs

Predicting cache behaviour for configurations entirely absent from training data is critical for efficient design space exploration. To demonstrate CBox's ability to generalize to previously unseen cache configurations, we use the CB-GAN model trained on four L1 Data cache configurations from Section 5.2. We test this model's accuracy on three entirely novel L1 cache configurations: 256set-6way, 256set-12way, and 32set-12way. As shown in Figure 9, the average case absolute percentage difference between *true* and *predicted* hit rates is 1.96%, 1.26% and 3.28% for 256set-6way, 256set-12way, and 32set-12way L1 cache configurations respectively. Thus, CB-GAN models trained on diverse cache configurations can successfully predict miss heatmaps for entirely unseen cache configurations with high fidelity.

Key Takeaway: We can generalize across cache configurations without requiring retraining for each new configuration, making CBox well-suited for early-stage design space exploration.

5.4. RQ4: Adapting across multiple cache levels

Modern architectures employ cache hierarchies with distinct size, associativity, latency, and miss rate profiles for each level. This presents a significant modelling challenge, as each cache level exhibits unique filtering behaviours based on its position in the memory hierarchy and its architectural parameters. To evaluate CBox's multi-level cache modelling capabilities, we examine a 64set-12way L1 Data cache, a substantially larger 1024set-8way L2 cache and a 2048set-16way L3 cache using SPEC benchmarks.

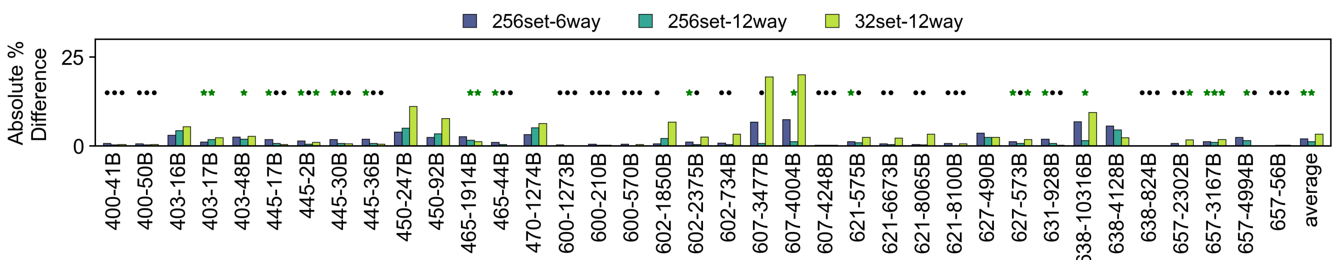


Figure 9: Absolute percentage difference in hit rates for SPEC, trained on four L1 configurations and tested on three unseen L1 configurations. The average is 1.96%, 1.26% and 3.28% for 256set-6way, 256set-12way, and 32set-12way respectively. Black dots and green stars on the bars indicate absolute percentage differences of $<1\%$ and 1–2%, respectively.

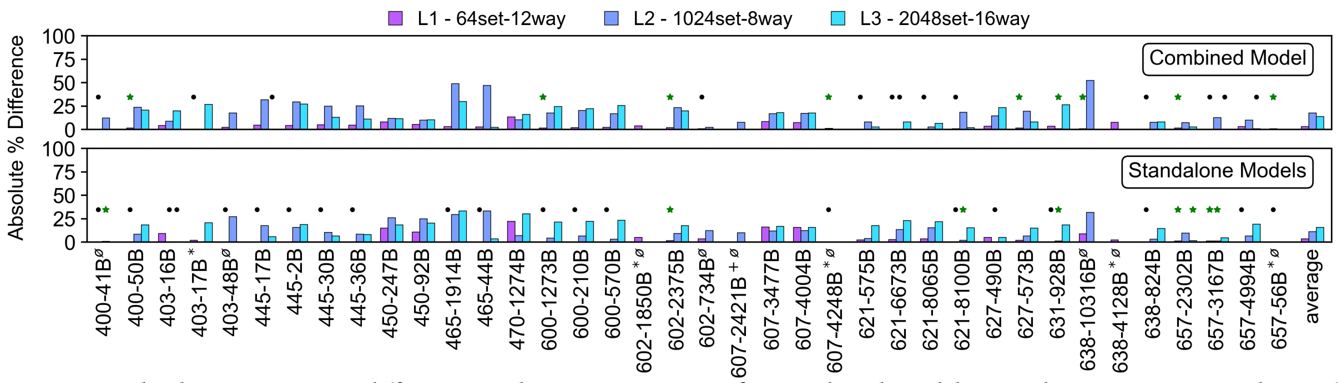


Figure 10: Absolute percentage difference in hit rates on SPEC for *combined model* trained on L1+L2+L3 cache, and *standalone models* trained exclusively on L1, L2 or L3 caches. Benchmarks with hit rates in low data regime (indicated by + for L1, $*$ for L2, and ø for L3 cache) are excluded. The average absolute percentage difference in hit rates for the combined model is 3.23%, 17.63%, and 14.06% for the L1, L2 and L3 caches. For standalone, it is 3.70% for L1, 11.40% for L2, and 15.89% for L3 caches. Black dots and green stars on the bars indicate absolute percentage differences of $<1\%$ and 1–2%, respectively.

We explore two training paradigms to assess whether hierarchical cache behaviour is better captured through integrated or level-specific models. The first is a combined model trained simultaneously on the L1, L2 and L3 cache configurations. The second is specialized standalone models trained exclusively on individual cache hierarchy levels.

The standalone models are given explicit set and way cache parameters. In contrast, the combined model is trained without any cache parameters, specifically to evaluate CB-GAN's ability to generalize without explicit architectural context. CB-GAN uses a Unet512 generator for the combined, standalone-L2, and standalone-L3 models to accommodate their highly variable hit rate patterns, while a Unet256 generator suffices for the standalone-L1 model. Both standalone and combined models employ a larger 142×142 PatchGAN discriminator model.

As shown in Figure 10, the combined model achieves average absolute percentage differences of 3.23% for L1 caches, 17.63% for L2 caches, and 14.06% for L3 caches. On the other hand, the absolute percentage difference in the average case is 3.70% for the standalone-L1 model, 11.40% for the standalone-L2 model and 15.89% for the standalone-L3 model. Benchmarks with hit rates in the low data regime were excluded from the inference evaluation as explained in Section 6.1. In Figure 10, symbols next to benchmark names indicate excluded benchmarks at different cache levels, namely (+) for L1, ($*$) for L2, and (ø) for L3.

Both the combined model and specialized standalone models achieve comparable hit rate predictions. However, both models exhibit higher prediction accuracy for L1 compared to L2 and L3 caches, which can be attributed to the fundamentally different memory access patterns present across levels. We anticipate that with additional training data representing a broader spectrum of hit rates, both models would attain higher accuracy across all cache levels.

Key Takeaway: CBox can generalize across cache hierarchy levels, offering flexibility to choose between a unified model or specialized models. The combined model effectively predicts cache behaviour without explicit cache-specific parameters, indicating that CBox can identify distinctive filtering behaviours across the cache hierarchy from the access and hit patterns alone.

5.5. RQ5: Parallelizing CBox inference

Traditional architectural simulation approaches are inherently sequential, limiting their scalability. We explore CBox's potential for parallelized inference, leveraging batch processing to evaluate multiple heatmaps simultaneously.

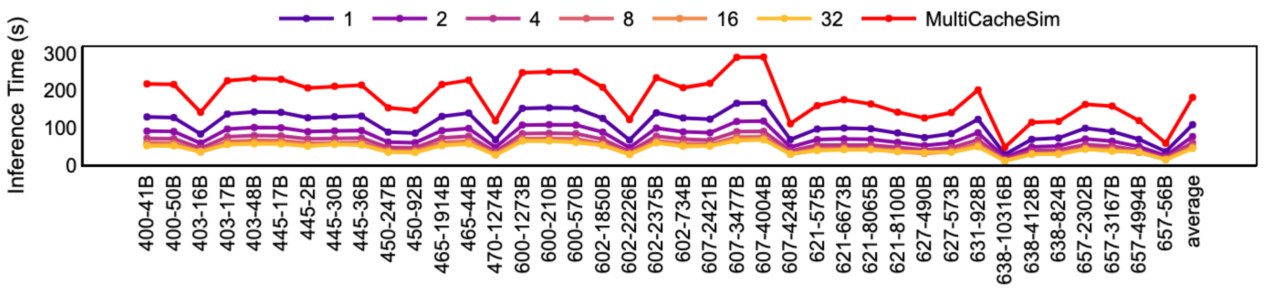


Figure 11: CB-GAN inference time across batch sizes (1 - 32) on an NVIDIA RTX A6000 GPU for model trained on 64set-12way L1 with SPEC benchmarks. Batch size of 32 gives 2.4× speedup in average inference time over batch size of 1.

Utilizing different batch sizes, we calculate CBox's inference time for each benchmark, which is the time required to generate `Synthetic` miss heatmaps corresponding to all `Real` access heatmaps of the benchmark during inference.

The results of parallelizing inference on CB-GAN model across batch sizes on a system with an NVIDIA RTX A6000 GPU (with Intel Xeon E5-2603 CPU and 32 GB memory) are shown in Figure 11. The CB-GAN model is trained on four L1 Data cache configurations (from Section 5.2), and the parallelized inference results correspond to the 64set-12way cache configuration. For sequential inference (batch size 1), the average time to predict all `Synthetic` miss heatmaps is 108.98 seconds. As we increase the parallelization via batch size, the inference time decreases significantly, reaching just 45.01 seconds for batch size 32 which is a 2.4× reduction in average inference time.

While a true apples-to-apples comparison with traditional simulation is not possible (since CBox is only modelling simulator behaviour, not simulating), we do provide a comparison in Figure 11 to MultiCacheSim [17], a high throughput cache-only simulator. Given the imprecise nature of the comparison, we do not claim a specific contribution but note 1.61×-1.81× speedups with an average of 1.67× for sequential CBox compared to MultiCacheSim.

Key Takeaway: By processing multiple `Real` access heatmaps simultaneously, CBox can significantly reduce time for architectural analysis. This is particularly valuable for large-scale architectural studies and CBox provides a distinct advantage over traditionally sequential cache simulation techniques in this case.

5.6. RQ6: CBox cache response characteristics

To facilitate comprehensive architectural analysis, we demonstrate the cache response characteristics of CBox by analyzing *true* and *predicted* hit rates leveraging the CB-GAN model trained on four L1 cache configurations (from Section 5.2). As shown in Figure 12, CBox performs high-accuracy hit rate predictions as evidenced by the dense cluster of data points in the upper-right quadrant, which represents benchmark traces with *true* hit rates exceeding 90%. For benchmark traces with intermediate hit rates (70% to 90%), we observe a different pattern. *Predicted* hit rates trend higher than *true* hit rates, indicating a positive

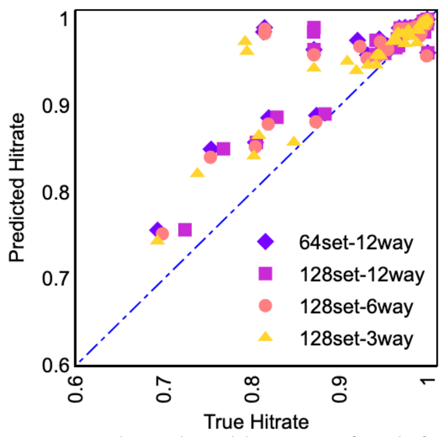


Figure 12: *true* and *predicted* hit rates for different cache configurations on SPEC benchmarks. Each datapoint is one benchmark running on a given cache configuration.

correlation bias. This bias is pronounced in two specific benchmark traces, `607-4004B` and `607-3477B`. We attribute this positive correlation to the composition of our training dataset, which contains a disproportionate number of high hit rate benchmark traces, as further discussed in Section 6.1. Note again that we strictly separate our training and testing benchmarks and thus all benchmarks are unseen, which could also contribute to the bias. In the real world, one may train on some inputs and test on other inputs from the same benchmark, which could improve the bias.

Key Takeaway: Despite the relative imbalance of low and high hit rate benchmark traces in our training data, CBox maintains robust prediction performance across a majority of inference workloads.

5.7. RQ7: Extending CBox to Cache Prefetching

We extend CBox to model microarchitectural components other than caches, specifically cache prefetchers. Modern processors employ sophisticated prefetchers to anticipate and mitigate cache misses. Given a memory address trace, prefetchers analyze access patterns and proactively fetch likely future memory addresses before they are explicitly requested. We utilize CBox to model this transformation exhibited by prefetchers by capturing

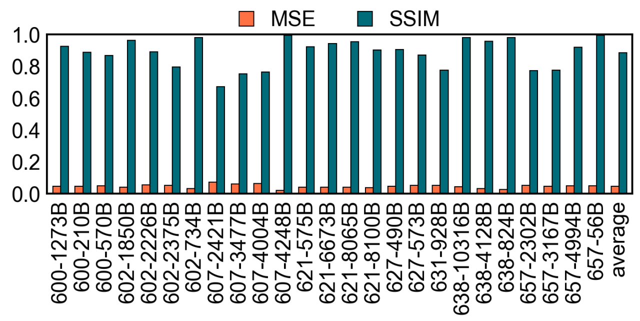


Figure 13: Mean Squared Error (MSE) and Structural Similarity (SSIM) of CB-GAN's predictions of Next Line Prefetcher behaviour for a 64set-12way L1 cache on SPEC 2017 benchmarks.

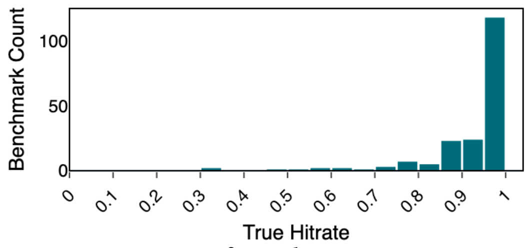


Figure 14: Histogram of *true* hit rates on a 64set-12way L1 cache of all SPEC 2006 and SPEC 2017 benchmarks.

both the address traces encountered by prefetchers and the resulting prefetched addresses as paired heatmaps.

We generate `Real` access heatmaps from address traces encountered by the prefetcher and corresponding `Real` prefetch heatmaps from the prefetched addresses. These heatmap pairs capture both spatial locality (relationship between addresses) and temporal locality (access timing patterns). The CB-GAN model is trained using these access-prefetch heatmap pairs to learn prefetcher behaviour. During inference, the `Synthetic` heatmaps produced by the trained CB-GAN model represent the memory addresses that would be prefetched by the cache prefetcher under given access address patterns. To evaluate prediction accuracy, we employ two complementary metrics: Mean Squared Error (MSE) and Structural Similarity Metric (SSIM). MSE measures the average squared per-pixel difference between `Real` and `Synthetic` prefetch heatmaps, with lower values indicating better performance. SSIM quantifies the structural information similarity between `Real` and `Synthetic` prefetch heatmaps, producing scores from -1 to 1, where 1 indicates perfect similarity.

We train CB-GAN on next-line prefetcher for a 64set-12way L1 Data cache using SPEC 2017 [29] benchmarks (only a subset of SPEC is used due to computational resource limitations). As shown in Figure 13, the consistently low MSE values coupled with high SSIM scores confirm that CBox can accurately model cache prefetcher behaviour.

Key Takeaway: CBox's heatmap-based approach is highly adaptable for modelling microarchitectural components beyond caches. We hypothesize that CBox can be extended to other prefetching algorithms and potentially other microarchitectural components.

6. Discussion, Future Work, and Limitations

6.1. Understanding the Data Ecosystem of CBox

Data quality and distribution are critical to the performance of the CB-GAN model. We utilize a dataset of 321 benchmarks comprising 189 SPEC, 100 Ligra, and 32 Polybench benchmarks, splitting each benchmark suite 80-20 for training and inference, as detailed in Section 4.1. Analysis of this dataset reveals patterns in hit rate distribution that directly impact model training and inference.

Figure 14 presents a histogram of *true* hit rates for all 189 SPEC benchmarks on a 64set-12way L1 Data cache. A majority of benchmarks exhibit very high hit rates, with over 95% of the SPEC benchmarks with hit rates exceeding 65%. Similarly, 70% of the 189 SPEC benchmarks have *true* hit rates higher than 40% on a 1024set-8way L2 cache, and 55% of the SPEC benchmarks have *true* hit rates higher than 35% on a 2048set-16way L3 cache. Even when considering all benchmark suites combined (including SPEC, Ligra and Polybench), more than 92% of the 321 benchmarks demonstrate *true* hit rates greater than 65% on a 64set-12way L1 cache. This stems primarily from SPEC being the largest benchmark suite in our dataset. Even though Ligra and Polybench exhibit more representative hit rate distributions, within SPEC, high-hit rate benchmarks significantly outnumber low-hit rate ones.

To ensure we are training and testing in the high data regime, based on the hit rate distribution of our dataset, we perform training and inference only utilizing benchmarks with hit rates greater than 65%, 40%, and 35% for L1, L2, and L3 caches. We apply these thresholds across all benchmarks in our dataset. All evaluations presented in Section 5 follow this approach. For instance, we do not present the absolute percentage difference for benchmarks in the low data regime which exhibit *true* hit rates below the threshold on L1, L2 and L3 caches in Figure 10, as indicated by the +, $*$ and ø symbols next to the benchmark names respectively. We observe that throughout our evaluations, CBox effectively learns cache filtering behaviour within the high data regime. However, limited data for low hit rate benchmarks prevents CBox from modelling those cases accurately, which manifests in part as the positive correlation bias visible in Figure 12. This limitation can be addressed by incorporating more applications with low hit rates into the training dataset. Nevertheless, since most real-world applications exhibit high cache hit rates, our current CB-GAN models remain capable of making accurate inferences on new applications. We conclude that CBox generalises well when sufficient data is available.

App	REaLTabFormer			Traditional		CBox		
	Tab-Base	Tab-RD	Tab-IC	HRD	STM	best	worst	average
600	23.90	5.20	23.70	5.90	18.20	0.00	0.15	0.07
602	56.60	27.30	19.20	19.70	29.00	1.53	5.27	3.49
607	11.20	15.10	11.00	15.20	9.70	0.29	16.23	10.83
631	81.50	81.60	43.50	0.00	0.20	0.15	0.15	0.15
638	38.40	8.20	27.70	33.20	8.90	0.01	8.96	3.85
avg % diff	42.32	27.48	25.02	14.80	13.20	0.39	6.15	**3.68**

TABLE 1: Absolute Percentage Difference comparison of **L1** cache miss rate prediction.

6.2. Neural Networks for Architectural Analysis

Previous works have applied deep learning models to architectural contexts. Shi et. al. [26] utilize a transformer model, REaLTabFormer, for memory workload synthesis. They evaluate the synthesis quality based on the similarity between original and generated workloads using miss ratios. We compare CBox against REaLTabFormer [26] and traditional approaches for memory behaviour analysis, namely Hierarchicial Reuse Distance (HRD) [19], and Spatio-Temporal Memory (STM) model [2]. We report the absolute percentage difference between *true* and *predicted* hit rates for SPEC 2017 benchmarks on an L1 Data cache. Note that CBox employs Champsim for data collection and validation, while REaLTabFormer uses the Gem5 simulator.

The results are shown in Table 1. For CBox, there can be multiple phases for the same benchmark (i.e. both `602.gcc_s-734B` and `602.gcc_s-2375B` belong to tgcc), so we report the worst, best and average absolute percentage difference for each. CBox has the lowest average absolute percentage difference for the L1 cache among all methods. The 'best' column shows that if we choose the lowest absolute percentage difference phase for the five benchmarks, the overall average is below 0.39%. Even if we pick the 'worst' phase for each benchmark, the overall average is below 6.15%, which is the lowest among all methods. If we compute the 'average' absolute percentage difference of all phases of the same benchmark then the overall average absolute across the five benchmarks is 3.68%.

6.3. Future Work and Limitations

Caches feature many options, such as further prefetching strategies, presence/absence of victim caches, inclusion/exclusion, etc which we have not studied here. We leave more complete treatment to future work. To help modelling such features, we expect that model parameters could be added to enable/disable them at inference time. Our study assumes a cache block size of 64B; future work could evaluate its parameterisation.

7. Related Work

7.1. Computer Architecture Simulation

Architectural simulators are the most common tools that computer architects use, including for design space exploration, performance/power analysis, etc. Many recently published computer architecture papers are verified using popular simulators such as Gem5 [4], PTLsim [32], Sniper [5], and ZSim [24] to name a few [1]. Gober et al. built ChampSim [11], a highly modular trace-based microarchitecture simulator, that can model modern high-performance out-of-order (OoO) cores. We use ChampSim to collect cache hit-and-miss traces as ground truth data.

7.2. Accelerating the Simulation Process

Many methods and strategies have been proposed to accelerate notoriously slow architectural simulation.

Sampled Simulation. Sampling is a popular technique where instead of simulating the entire program, only a small number of samples are simulated. This reduces the simulation time and provides a good estimate of the benchmark performance. Two methods commonly used to select a subset of representative instructions are statistical sampling and target sampling. Statistical sampling is typically implemented by randomly selecting samples from the entire instruction stream or periodically from regular intervals [30, 31, 33]. Targeted sampling involves selecting sampling points after analyzing a program's behaviour. The program is first grouped into different phases, in which the instructions share similar behaviour, and then single sample units are picked from each phase [21, 25].

Statistical Simulation. Statistical simulation reduces overhead by combining detailed and analytical simulation [2, 9, 19, 20, 26]. A statistical profile of the program's characteristics is first generated with trace-based tools. The profile is then used to create an instruction trace that is fed into a trace-driven simulator. Simulation is made quicker because the synthetic traces are shorter.

Parallel Simulation. Parallel simulation reduces runtime by running different sampling points simultaneously, taking advantage of modern multi-core processors [5, 8, 24]. Although these methods can significantly speed up simulation, the use of instruction sequences is serial in nature, even with the help of deep learning [16]. Our alternative, learning-based mechanism, uses heatmaps as input, which also enables parallelization. CBox is not strictly a simulator, but can reduce the time required to model cache behaviour, and we expect future advances could incorporate more features, e.g. energy models.

8. Conclusion

CacheBox (CBox) is a novel framework that utilizes heatmap-based representations of microarchitectural components and a Generative Adversarial Network (GAN) to model architectural cache simulator behaviour. Our evaluation shows that the CBox achieves high accuracy in predicting cache hit patterns while significantly reducing computational time through parallelized inference, and demonstrates robust generalization capabilities across different cache configurations and cache hierarchy levels. This research shows the potential of neural networks for architectural analysis and opens up possibilities for further progress, such as modelling additional parameters like power and latency, investigating multicore architectures, and applying similar heatmap-based transformations to other performance-critical microarchitectural components.

Appendix

1. Abstract

This artifact provides the code, scripts, and a subset of the data used in our paper. Due to the large size of the full datasets, we include only representative portions.

Our workflow takes memory access traces from benchmarks (SPEC, Ligra, Polybench) generated with `ChampSim` and converts them into memory access heatmaps. Real access and miss heatmaps are generated as ground truth for training and evaluation of the `CacheBox_GAN` model. After training, the model produces synthetic miss heatmaps, which are then used with real access heatmaps to compute predicted cache hit rates. Comparison with true hit rates enables evaluation of model accuracy.

We provide a reference example demonstrating the workflow from raw traces to cache hit-rate evaluation, pre-trained model weights for RQ2, and scripts to reproduce a subset of the experiments. While full datasets are not included due to size, scripts and data snippets are provided for all steps.

1.1. Artifact contents. `CacheBox/`

- `HeatmapDataGenerator/`
  — `MemoryAccessTraces/` (memory access traces with hit/miss info generated via ChampSim `ChampSim`)
  — `HeatmapRepresentation/` (scripts to convert memory access traces into real access and real miss heatmaps)
- `Data/` (real access and miss heatmaps divided into Train and Test sets)
- `CacheBox_GAN/`
  — `models/` (CB-GAN implementations (with/without cache parameters, large models included))
  — `Experiments/` (training and inference scripts (RQ1–RQ4, RQ7, reference example))
  — `checkpoints/` (training checkpoints)
- `TrainedModels/` (pre-trained checkpoint for RQ2)
- `InferenceResults/` (synthetic miss heatmaps organized by experiment)
- `ArchitecturalMetricCalc/` (scripts to compute predicted and true cache hit rates)

2. Artifact check-list

- **Algorithm:** Cache hit-rate prediction using CB-GAN on cache access heatmaps
- **Program:** Heatmap generation, CB-GAN training/inference, hit-rate calculation
- **Transformations:** Memory access traces to heatmap conversion
- **Model:** `CacheBox_GAN` (PyTorch)
- **Data set:** Memory access traces generated by `ChampSim` (SPEC, Ligra, Polybench)
- **Run-time environment:** Python, PyTorch, CUDA optional
- **Hardware:** CPU supported, GPU recommended (tested with NVIDIA RTX A4000)
- **Run-time state:** Trained model checkpoints included
- **Execution:** Heatmap generation, training, inference, hit-rate calculation
- **Metrics:** True and predicted cache hit rates
- **Output:** Access/miss heatmaps, synthetic miss heatmaps, hit-rate statistics
- **Experiments:** Reference example reproduces end-to-end workflow; pre-trained model inference for RQ2; scripts for RQ1–RQ4, RQ7 included for inspection
- **Disk space required:** 9GB unzipped
- **Preparation time:** Workflow ready if Python and PyTorch installed; no automated installation provided
- **Experiment runtime:** Reference example inference: 10–30 min; full reference example: 1–1.5 hours
- **Publicly available:** Yes
- **Workflow automation framework:** Bash scripts
- **Archived DOI:** https://doi.org/10.5281/zenodo.16935883

3. Description

3.1. How to access.
Artifact available at: https://doi.org/10.5281/zenodo.16935883. Unpack and navigate to `CacheBox`. All code, scripts, example data, and pre-trained models are included.

3.2. Hardware dependencies.

- CPU-only execution supported (slower)
- GPU recommended for training (NVIDIA RTX A4000 tested)
- Minimum RAM: 16 GB

3.3. Software dependencies.

- Python 3.8, 3.9, 3.10, or 3.11 (PyTorch 2.0.1+cu117 officially supports these versions; tested with 3.8.10)
- PyTorch 2.0.1+cu117
- CUDA (optional, for GPU acceleration)

Note: The provided ‘requirements.txt‘ lists the correct PyTorch version, but we recommend installing PyTorch separately first (using the provided ‘requirements.sh‘) so the proper CUDA-enabled build is installed. PyTorch is only required if you plan to run training or inference. If you are only using the repository for heatmap generation or hit rate calculation, PyTorch is not needed.

3.4. Data sets.

- Memory traces from `ChampSim` for SPEC, Ligra, Polybench benchmarks
- Real access and miss heatmaps for SPEC, L1 64set-12way cache configuration
- Synthetic miss heatmaps generated after running inference on reference-model

3.5. Models.

- `CacheBox_GAN` implementation
- Pre-trained weights for RQ2
- Reference-model weights for quick testing

4. Installation

1) Download artifact from https://doi.org/10.5281/zenodo.16935883
2) Unzip the repository: `> unzip CacheBox.zip`
3) Install dependencies:
   Run the setup script: `> ./CacheBox/requirements.sh`

`requirements.sh` script will:

- Create a virtual environment
- Install PyTorch 2.0.1 + CUDA 11.7
- Install all other dependencies from requirements.txt (numpy, scipy, pandas)

5. Experiment workflow, Evaluation and Expected Results

5.1. Reference Example. This experiment generates real access and miss heatmaps for the training set of an L1D cache with a 64-set, 12-way configuration that contains three SPEC benchmarks. It then trains a CB-GAN model on these three benchmarks for 1 epoch. After training, the model runs inference on 10 SPEC benchmarks to evaluate performance. Finally, the hit rate calculation script calculates the true and predicted cache hit rates, along with the absolute percentage difference between them.

Convert memory access traces to heatmaps:.

- Run:
  ```
  > cd HeatmapDataGenerator/HeatmapRepresentation
  > ./run_heatmap_generation.sh
  ```
- Generated heatmaps are saved to `Data/SPEC/L1-64set-12way_reference_example/TRAIN`.
- These heatmaps are **identical to the ones in** `Data/SPEC/L1-64set-12way/TRAIN`.
- For training the CB-GAN model for the reference example, we use the heatmaps in `Data/SPEC/L1-64set-12way/TRAIN`, so users **do not need to generate the heatmaps** if they only want to run the model training, inference and hit rate calculation steps.

Train CB-GAN model and run inference:.

- Run:
  ```
  > cd CacheBox_GAN
  > ./Experiments/ReferenceExample.sh
  ```
- This script performs both training and inference for the reference example. Training data is located in `Data/SPEC/L1-64set-12way/TRAIN`, and access heatmaps of inference benchmarks are in `Data/SPEC/L1-64set-12way/TEST/FULL`.
- The trained model is saved to `CacheBox_GAN/checkpoints`, and inference results are stored in `InferenceResults/ReferenceExample/L1-64set-12way`.
- The trained model is identical to the one in `TrainedModels/ReferenceExample`, and the inference results match those in `InferenceResults/ReferenceExample/L1-64set-12way_expected`.
- **Users do not need to run training or inference if they only want to calculate hit rates.** Alternatively, to run only training, comment out the inference command in `ReferenceExample.sh`. To run only inference, comment out the training command in the same script.

Compute cache hit rates:.

- Run:
  ```
  > cd ArchitecturalMetricCalc
  > ./run_hitrate_calc.sh reference_example
  ```
- Calculates true hit rates using real access heatmaps from `Data/SPEC/L1-64set-12way/TEST/FULL` and real miss heatmaps from `Data/SPEC/L1-64set-12way/TEST/MISS`.
- Calculates predicted hit rates using synthetic miss heatmaps from `InferenceResults/ReferenceExample/L1-64set-12way` along with real access heatmaps from `Data/SPEC/L1-64set-12way/TEST/FULL`.
- All results are saved in `ArchitecturalMetricCalc/HitrateResults`.
- The current `run_hitrate_calc.sh` script has been updated to use synthetic heatmaps from `InferenceResults/ReferenceExample/L1-64set-12way_expected`, **allowing hit rate calculation without running previous steps**.
- Update the `run_hitrate_calc.sh` script to use synthetic heatmaps from `InferenceResults/ReferenceExample/L1-64set-12way` if needed.
- Expected results for hit rate calculation are available in `ArchitecturalMetricCalc/ExpectedResults`.

5.2. Pre-trained model for RQ2. This experiment uses the pre-trained RQ2 model, trained on four L1D cache configurations (64set-12way, 128set-12way, 128set-6way, 128set-3way) with 150 SPEC benchmarks. The experiment runs inference on 10 SPEC benchmarks from the 64set-12way configuration, a subset of the 39 SPEC benchmarks evaluated in the paper for this cache configuration. Finally, it calculates the true and predicted hit rates, producing results for a subset of benchmarks corresponding to 64set-12way configuration as shown in Figure 8 in the paper. Details to run provided in the Readme.

5.3. Other Experiments (RQ1, RQ4, RQ7). Scripts to run training and inference for all these experiments are present in `Experiments/`. The `Data/` folder contains placeholders for the various data files used in these experiments. These placeholders allow the experiment scripts to run without errors. **The folders do not contain actual data to avoid increasing repository size.**

References

[1] A. Akram and L. Sawalha, “A survey of computer architecture simulation techniques and tools,” *Ieee Access*, vol. 7, pp. 78 120–78 145, 2019.

[2] A. Awad and Y. Solihin, “Stm: Cloning the spatial and temporal memory access behavior,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 237–247.

[3] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, “Pythia: A customizable hardware prefetching framework using online reinforcement learning,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1121–1137.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[6] D. Dangwal, W. Cui, J. McMahan, and T. Sherwood, “Safer program behavior sharing through trace wringing,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1059–1072.

[7] “DPC3,” https://dpc3.compas.cs.stonybrook.edu/, (Accessed on 11/14/2022).

[8] L. Eeckhout, *Computer architecture performance evaluation methods*. Morgan & Claypool Publishers, 2010.

[9] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere, “Statistical simulation: Adding efficiency to the computer designer’s toolbox,” *Ieee Micro*, vol. 23, no. 5, pp. 26–38, 2003.

[10] Y. Fu and D. Wentzlaff, “PriME: A parallel and distributed simulator for thousand-core chips,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 116–125.

[11] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, “The championship simulator: Architectural simulation for education and competition,” *arXiv preprint arXiv:2210.14324*, 2022.

[12] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” 2014.

[13] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, “Learning memory access patterns,” in *International Conference on Machine Learning*. PMLR, 2018, pp. 1919–1928.

[14] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,” *CVPR*, 2017.

[15] C. Li and M. Wand, “Precomputed real-time texture synthesis with markovian generative adversarial networks,” in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part III 14*. Springer, 2016, pp. 702–716.

[16] L. Li, S. Pandey, T. Flynn, H. Liu, N. Wheeler, and A. Hoisie, “Simnet: Accurate and high-performance computer architecture simulation using deep learning,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–24, 2022.

[17] B. Lucia. MultiCacheSim: A coherent multiprocessor cache simulator. [Online]. Available: https://github.com/blucia0a/MultiCacheSim

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[19] R. K. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, “Fast and accurate exploration of multi-level caches using hierarchical reuse distance,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 145–156.

[20] S. Nussbaum and J. E. Smith, “Modeling superscalar processors via statistical simulation,” in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2001, pp. 15–24.

[21] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, “Using simpoint for accurate and efficient simulation,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[22] “Polybench/c – homepage of louis-noël pouchet,” https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, (Accessed on 11/14/2022).

[23] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer, 2015, pp. 234–241.

[24] D. Sanchez and C. Kozyrakis, “Zsim: Fast and accurate microarchitectural simulation of thousand-core systems,” *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: Association for Computing Machinery, 2002, p. 45–57. [Online]. Available: https://doi.org/10.1145/605397.605403

[26] C. Shi, F. Jiang, Z. Liu, C. Ding, and J. Xu, “Memory workload synthesis using generative ai,” in *Proceedings of the International Symposium on Memory Systems*, 2023, pp. 1–7.

[27] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[28] “Spec cpu® 2006,” https://www.spec.org/cpu2006/, (Accessed on 11/14/2022).

[29] “Spec cpu® 2017,” https://www.spec.org/cpu2017/, (Accessed on 11/14/2022).

[30] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, “Simflex: statistical sampling of computer system simulation,” *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[31] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, “Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling,” in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003, pp. 84–97.

[32] M. T. Yourst, “Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator,” in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2007, pp. 23–34.

[33] Z. Zhang, C. Ye, R. Lavaee, N. Gu, and C. Ding, “Fine-grained data usage analysis by access sampling: seeing the data that is not there,” in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 221–231.