Mocktails: Capturing the Memory Behaviour of Proprietary Mobile Architectures

Mario Badr
University of Toronto
Toronto, ON, Canada
mario.badr@utoronto.ca

Carlo Delconte
Arm
Cambridge, UK
carlo.delconte@arm.com

Isak Edo
University of Toronto
Toronto, ON, Canada
isakedo@gmail.com

Radhika Jagtap
Arm
Cambridge, UK
radhika.jagtap@arm.com

Matteo Andreozzi
Arm
Cambridge, UK
matteo.andreozzi@arm.com

Natalie Enright Jerger
University of Toronto
Toronto, ON, Canada
enright@ece.utoronto.ca

***Abstract*—Computation demands on mobile and edge devices are increasing dramatically. Mobile devices, such as smart phones, incorporate a large number of dedicated accelerators and fixed-function hardware blocks to deliver the required performance and power efficiency. Due to the heterogeneous nature of these devices, they feature vastly larger design spaces than traditional systems featuring only a CPU. Currently, academia struggles to fully evaluate such heterogeneous systems on chip due to the limited access and availability of proprietary workloads. To address these challenges, we propose Mocktails: a methodology to synthetically recreate the varying spatio-temporal memory access behaviour of proprietary heterogeneous compute devices. We focus on capturing the interspersed address streams of the workload and the burstiness of the injection process for proprietary compute devices commonly found in mobile systems. We evaluate Mocktails in simulation with proprietary memory traces of IP blocks. Mocktails accurately recreates the dynamic behaviour of memory access scheduling for memory controller metrics including read row hits (at most 7.3% error) and write row hits (at most 2.8% error). Architects can use Mocktails in their simulations as a substitute for a proprietary compute device, making the tool a useful conduit between industry and academia.**

***Index Terms*—Simulation, Systems-on-Chip, Memory Systems**

I. Introduction

The heterogeneity available on today's systems-on-chip (SoCs) is staggering. Currently, mobile SoCs allocate less than a third of their area to general-purpose cores with the rest dedicated to specialized intellectual property (IP) blocks [1]. These IP blocks perform tasks from video encoding to immersive graphics (e.g., augmented reality) to networking [19]. However, much less is known about both the underlying hardware of these IP blocks and their workloads compared to CPUs; both the IP and workloads are largely considered proprietary by the company that has designed them. Without access to how and what IP blocks execute, evaluating SoCs in academic research is difficult; only 1% of papers published in top architecture conferences focus on mobile SoCs [35]. This gap between academia and industry will continue to widen with the growing number of IP blocks found in future mobile systems [38].

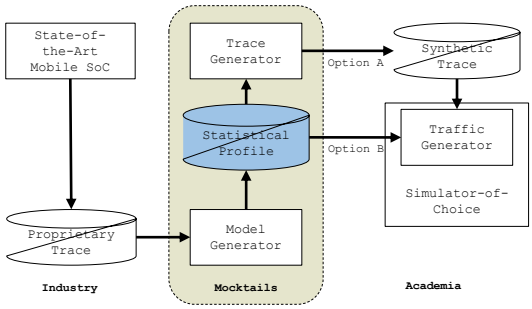


Fig. 1: Two use cases for Mocktails.

A key challenge in designing heterogeneous SoCs is ensuring that the memory system can efficiently deliver data to these compute devices [19], [35]. Heterogeneous IPs place varying demands on the memory hierarchy–they access vastly different volumes of data, have different access patterns and experience different latency and bandwidth sensitivities. But how can academia explore memory hierarchy designs when the workloads and devices are proprietary? While there is a plethora of workloads addressing general-purpose compute [8], [42], [43], [47], there is a dearth of workloads and open source models for proprietary hardware IP blocks used in heterogeneous mobile SoCs. This is problematic for the architecture community which relies primarily on simulation to evaluate ideas [5].

A promising approach that addresses the lack of access to models and workloads is statistical simulation. In statistical simulation, a profile of a workload running on a device is created from the characteristics of its execution and then used to synthesize the workload's behaviour. Fig. 1 shows how Mocktails uses statistical simulation to bridge the gap between industry and academia. Industry collects traces of memory requests from their state-of-the-art IP and uses Mocktails' model generator to create a statistical profile. Using a black-box modeling approach, the Mocktails profile hides details that are contained in the original trace. This allows industry to distribute profiles freely without revealing proprietary secrets.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00046

Academic researchers can augment their simulations with synthetic requests generated by these profiles to recreate the behaviour of proprietary workloads and devices.

Prior statistical simulation techniques focus on modeling general-purpose CPUs [3], [4], [6], [28], [32]. However, these models are only applicable to CPUs (Sec. II), which necessitates further research on how to model heterogeneous devices. Architects have begun exploring how to model GPUs [20], [21], [23], [30], [33], [48]. But developing a different model for every generation or type of compute devices is intractable. Next-generation systems-on-chip (SoCs) are developed at a rapid pace and range from low-end to high-end with different hardware characteristics [19]. Ideally, a single technique should be robust enough to encompass multiple compute devices that interact with the memory hierarchy in different ways.

In this paper, we develop Mocktails, a statistical simulation technique that is both accurate and robust enough to apply to multiple devices. The most commonly used compute devices in a standard SoC include CPUs, GPUs, display processing units (DPUs) for controlling screen output, and video processing units (VPUs) for streaming video. Mocktails uses a divide-and-conquer approach, grouping memory requests temporally and spatially so that each partition can be modeled independently. Our spatial partitions are dynamically-sized, adapting to the memory access behaviour of the compute device and workload being modeled. We use these models to synthesize memory requests, taking special care to recreate the time-varying behaviour of the original memory access pattern. Specifically, we make the following contributions:

- A novel scheme for dynamically partitioning requests based on their spatial behaviour.
- A novel injection process model that recreates the time-varying behaviour of multiple, concurrent address streams.
- Mocktails accurately captures the memory access behaviour of CPUs, GPUs, DPUs, and VPUs, without any assumptions of the underlying compute device.
- An open source implementation of Mocktails and the profiles generated for this paper available at: https://github.com/mariobadr/statistical-simulation.

II. Related Work

In this section, we present three areas of related research. First, we discuss statistical simulation for CPUs. Second, we review current GPU modeling approaches. Finally, we present literature related to modeling domain-specific accelerators.

Statistical Simulation for CPUs. Reuse distance models are pervasive in statistical simulation for modeling temporal locality in CPU workloads [3], [6], [15], [28], [31], [32], [45], [46]. Reuse distance is the number of unique addresses referenced between consecutive requests to the same address [7], [29], [50]. In general, a reuse distance profile is a discrete distribution of the relative frequency of observed reuse distances. WEST observes that building a reuse distance model at the global level (i.e., all requests) does not accurately capture the temporal locality of cache sets [6]. Instead, WEST builds multiple reuse distance models for each cache set. MeToo extends WEST to explore the design space of main memory [46], emphasizing the importance of request timing by modeling an application's instruction dependencies and instruction count intervals.

While reuse distance captures temporal locality, it does not capture spatial locality. To overcome this issue, HRD creates reuse distance profiles for multiple block granularities (e.g., 64B and 4KB) [28]. Stride pattern models can also capture spatial locality [3], [31], [32]. The main difficulty in modeling strides is that several address streams are interleaved in time [22]. STM overcomes this problem with a novel stride pattern table (similar to a Markov chain) that predicts the next stride for a history of strides [3]. SLAB groups requests by PC to uncover sequences that can be modeled with a single stride value [32]. HALO groups requests into 4KB blocks and dynamically determines how much history to use when modeling the stride pattern [31].

Statistical simulation techniques are well-tuned and accurate for mimicking CPU behaviour [3], [6], [28], [31], [32], [46]. However, they are not appropriate for modeling the proprietary IP blocks that we target. The main reason is specialized hardware accelerators do not behave like general-purpose CPUs. For example, non-CPU devices have very different reuse behaviour [19], [30]. Moreover, prior CPU techniques exploit several CPU-specific features such as the PC [32], instruction set [28], basic blocks [22], or the existence of virtual memory (e.g., 4KB page sizes) [31]. In contrast, Mocktails makes no assumptions on the underlying compute device or workload.

Modeling GPUs. A GPU's parallel execution model makes it difficult to use insights from reuse distance that have helped architects understand CPUs [30]. Instead, analytical and statistical GPU models take into account the GPU's fine-grained multi-threaded execution model. For example, G-MAP is a statistical simulation technique that uses the execution models of CUDA and OpenCL as a guide [33]. Specifically, G-MAP generates request sequences for each GPU core and then orders them based on a model of the scheduling policy. Like SLAB [32], G-MAP partitions requests by PC to find constant stride patterns. Analytical [20], [21], [30], [49] and statistical [33] models are useful for understanding the multi-threaded execution model of GPUs. Mocktails differs in that its black-box approach does not assume an execution model, making it useful for modeling other compute devices for statistical simulation.

Modeling IP Blocks. Modeling of domain-specific accelerators for mobile SoCs is less mature than for CPUs [35]. Aladdin estimates the performance, power, and area of various IP blocks using dynamic data dependence graphs [38], [39]. LogCA is a high-level model for identifying performance bottlenecks in accelerators [2]. Gables [19] finds bottlenecks by adapting the Roofline model. Unlike LogCA, Gables considers multiple IP blocks running concurrently on a mobile SoC. GemDroid [9] is a simulation framework for evaluating SoCs; it features high level models of IP blocks which may include simplications. Rather than a high-level system, Mocktails targets something completely different: allowing integration of more recent, closed-source IP into a simulation infrastructure

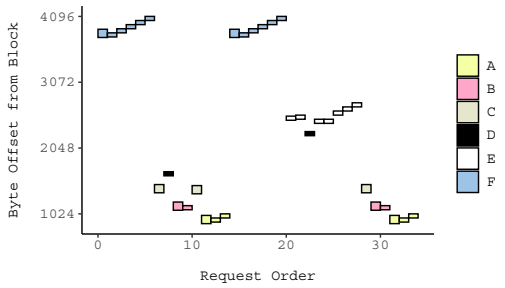


Fig. 2: Requests from a 4KB memory region of a VPU workload (HEVC1).

of the researcher's choice. Mocktails is a novel addition to the architect's toolbox that tackles a different challenge: the growing gap between academia and industry caused by increasing heterogeneity and proprietary IP blocks.

III. Mocktails

At a high level, Mocktails (1) deconstructs a sequence of memory requests into partitions, (2) models each partition independently, and (3) uses each model to synthetically reconstruct the behaviour of the original workload. Synthetically reconstructing the behaviour hides proprietary information; industry vendors are often unwilling to disclose memory traces as they reveal too much information about what the proprietary hardware design does, how the software is implemented and other aspects of the system.

In this section, we use Fig. 2 as an example to explain temporal and spatial partitioning and motivate the need for both. Fig. 2 shows over 30 memory requests (as rectangles) from HEVC1 that belong to a 4KB memory region. We focus on request features that are available at the interface between the compute device and memory. Specifically, we consider the timestamp, address, operation (read/write), and size (bytes requested) of memory requests. In Fig. 2, the height of a rectangle indicates the size of the memory request (e.g., 64 and 128 bytes). The figure plots the order requests to a specific block are sent to the memory system.

The goal of Mocktails is to not only find access patterns in the address streams, but also to capture temporal changes (i.e., burstiness, or lack thereof) at the global level. Capturing the spatio-temporal behaviour of the workload is not limited to one particular step in Mocktails. Partitioning along both spatial and temporal dimensions is crucial, but so is modeling the injection process of each partition (Sec. III-B) as well as combining the injection processes into one that resembles the original memory access behaviour (Sec. III-C).

A. Partitioning Memory Requests

There are multiple ways to partition memory requests (e.g., by size, operation, time, or address). We focus on the temporal (time) and spatial (address) dimensions because temporal and spatial locality are well known properties of applications [13]. Fig. 2, our example, partitions in both the spatial and temporal dimensions. The requests shown are from the first 100,000

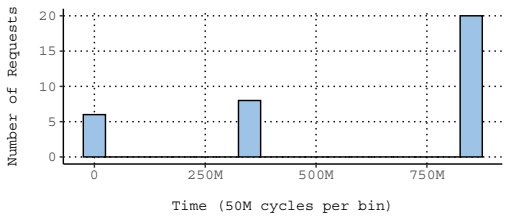


Fig. 3: The timing of the requests from Fig. 2.

requests (temporal) in HEVC1 and belong to the same 4KB block (spatial). First, we review the different approaches to temporal partitioning then present our novel approach for creating spatial partitions. After introducing temporal and spatial partitioning, we show how to combine both dimensions hierarchically to uncover simple patterns within partitions.

Temporal Phases. Dividing requests temporally in a way that is agnostic of the compute device is challenging. Prior work uses the instruction stream to identify phase behaviour [24], [25], [40], [41]. However, this assumes the existence of a PC or basic blocks, which is not applicable to all compute devices.

To address this challenge, prior art relies on fixed-size temporal partitions (i.e., intervals). STM uses fixed-size intervals containing at most 100,000 requests (i.e., request_count) for single-core processors [3]. Using request_count for temporal partitions helps bound the number of requests being modeled at a time but it ignores the time the requests were injected. In Fig. 3, we plot the number of requests sent per 50M cycle interval. Clusters of requests are separated in time by hundreds of millions of cycles. SynFull uses fixed-size cycle_count intervals at two granularities: macro (100,000s of cycles) and micro (100s of cycles) [4]. This captures bursty and idle phases, but does not bound the number of requests in an interval. Mocktails is compatible with both approaches for creating temporal partitions to capture time-varying behaviour.

Dynamic Memory Regions. Prior work divides requests spatially using fixed-size blocks. For example, HALO divides requests into 4KB memory regions that are modeled independently [31]. These regions may be merged if two contiguous regions have similar models. Our proposed spatial partitioning scheme dynamically uncovers variable-sized memory regions *before* modeling. The dynamic memory regions vary in size and are not multiples of an input block size (e.g., 4KB).

Dynamic spatial partitioning merges requests that access overlapping or adjacent memory regions (Alg. 1). Fig. 2 shows the six dynamic partitions created by our approach (A through F). It is possible that multiple requests, spread out over time, belong to the same memory region (e.g., due to reuse). For example, the dynamic partition F is made up of two sets of six requests that access the same memory region. Dynamic spatial partitioning may find a request that does not overlap and is not spatially adjacent to any other requests. Because the objective of partitioning is to generate a subset of requests to model, it does not make sense to model a single request. We merge *lonely* requests with other lonely requests (e.g., dynamic partition D). If there are multiple lonely requests that are equally spaced

out in memory (i.e., they have the same stride between them), we group them into a single partition (not shown in Fig. 2).

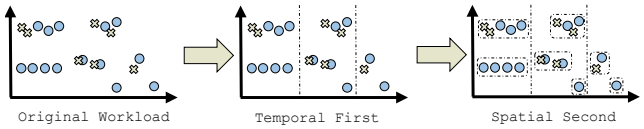
Algorithm 1: Pseudocode for Dynamic Spatial Partitioning

```
Data: Sequence of memory requests.
Result: Non-overlapping spatial partitions.
ranges ← [];
foreach Memory Request r ∈ requests do
    ranges.append([r.address, r.address + r.size]);
end
ranges.sort();
partitions ← [];
group ← ranges[0];
i ← 1;
while i < ranges.length() do
    if ranges[i].intersects(group) then
        group.expand(ranges[i]);
    else
        partitions.append(group);
        group ← ranges[i];
    end
    i ← i + 1;
end
partitions.append(group);
```

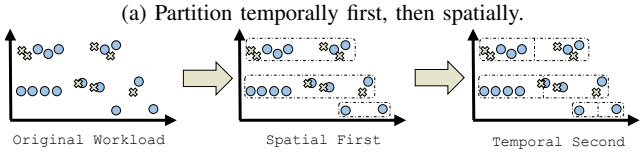
The requests in Fig. 2 are sparse and irregular, accessing only a small subset of the 4KB block. But there are patterns among the requests within the 4KB block (e.g., partitions A, B, F). Considering these requests as a single partition would increase the variance of the memory request features that need to be modeled. We reduce this variability with dynamic spatial partitioning, which finds fine- and coarse-grained memory regions. Our hypothesis is that requests within a memory region behave similarly and that dynamic spatial partitioning adapts to the memory access behaviour.

Hierarchical Partitioning. Capturing the spatio-temporal behaviour of the requests sent by a compute device requires partitioning in both dimensions. Hierarchical partitioning is not only important at the global level (i.e., when considering all requests), but also within a partition. Mocktails accepts a hierarchical configuration as input. The configuration specifies the number of layers in the hierarchy. For each layer, the type of partitioning (e.g., spatial or temporal) is specified. For spatial partitioning, Mocktails supports fixed-size and dynamic schemes. For temporal partitioning, Mocktails supports `request_count` and `cycle_count`. The leaves of the hierarchy represent the final partitions of requests.

Fig. 4a shows an example hierarchy where requests are partitioned temporally (three intervals) then spatially (dynamic). Here we illustrate reads with a circle and writes with an x. In the first temporal partition, there are two concurrent address streams that are each assigned to a spatial partition, which we model separately. In the top spatial partition, there is a mixture of reads and writes while in the bottom, there are only reads. In addition, read requests in the bottom spatial partition are evenly spaced out in time and access the same memory location. Relying only on temporal partitioning requires the model to account for variability in the operation, timestamp,



(a) Partition temporally first, then spatially.



(b) Partition spatially first, then temporally.

Fig. 4: Partitioning requests in a two-level hierarchy. Reads are circles and writes are x's.

TABLE I: Requests from Partition F in Fig. 2.

Address	*1 Temporal Partition*		*2 Temporal Partitions*	
	Stride	*Size*	*Stride*	*Size*
81002**EB8**	N/A	128	N/A	128
81002**EC0**	8	64	8	64
81002**F00**	64	64	64	64
81002**F40**	64	64	64	64
81002**F80**	64	64	64	64
81002**FC0**	64	64	64	64
81002**EB8**	-264	128	N/A	128
81002**EC0**	8	64	8	64
81002**F00**	64	64	64	64
81002**F40**	64	64	64	64
81002**F80**	64	64	64	64
81002**FC0**	64	64	64	64

and address features because requests in both spatial partitions would be interleaved. The second temporal partition is further divided into three spatial partitions, but these spatial partitions have different start times. By allowing spatial partitions to begin at different times, we capture burstiness of requests and of addresses being accessed.

Fig. 4b shows an example where requests are partitioned spatially (dynamic) then temporally (two intervals). This approach finds the time-varying behaviour within a spatial partition, unlike the previous approach which finds the spatial behaviour within a time interval. Table I shows the requests for the dynamic partition F (Fig. 2). In the *1 Temporal Partition* column, we show the sequence of strides and sizes for all requests in F. In the *2 Temporal Partitions* column, we further divide the requests in F into two temporal partitions (using an `interval_count` of 2). If we only partition spatially (*1 Temporal Partition*), then the leaf contains all 12 requests in the table. If we partition spatially then temporally (*2 Temporal Partitions*), then there are two leaves, each containing six requests (the second leaf in Table I is shaded). With two temporal partitions, the stride and size features can be modeled with 100% accuracy using Markov chains (e.g., a stride of 64 is always followed by the same stride). But with only one temporal partition, a Markov chain does not capture the sequence perfectly (e.g., a stride of 64 can be followed by either 64 or -264).

We discuss how to model each leaf independently in

Sec. III-B. Table I shows that the partitioning scheme has an impact on model accuracy. For example, modeling each of the two temporal partitions separately with a Markov chain, we capture the *Stride* and *Size* columns perfectly. In Sec. III-C, we demonstrate how to reconstruct a sequence of memory requests that mimics the behaviour of the workload by combining the independent models. The hierarchical configuration may increase the meta-data overhead of the model; we model the trade-off between accuracy and meta-data in Sec. IV-C.

B. Modeling the Leaves

Each leaf in the hierarchy consists of a sequence of requests. In Mocktails, our statistical profile is a collection of models, one for each leaf in the hierarchy. In this section, we describe our approach to recreate the behaviour of each request feature (i.e., timestamps, address, operation, and size). For timestamps and addresses, we consider the difference between subsequent values (i.e., delta values) to capture relative changes.

We model each feature in isolation, making the assumption that they are independent (we evaluate the impact this has on accuracy in Sec. IV). If the feature shows no variability in the leaf node, then the sequence of values produced for that feature can be generated from a single value (e.g., a constant stride captures a linear pattern). If there is variability, we generate the sequence from an initial state and a Markov chain. Markov chains of each feature allow us to capture both regular and irregular patterns in a leaf. We call our approach, choosing between a **M**arkov **c**hain or **C**onstant value, the McC model.

All requests in Fig. 2 are read requests. Thus, all dynamic partitions have a McC model with a constant value (read) for the operation feature. However, this is not the case for all features. Table I shows the requests for dynamic partition F (focusing on *1 Temporal Partition* column). Consider the sequence of sizes: Our Markov chain generates a size of 64 with 100% probability if the last size was 128. However, if the last size was 64, there is a 89% probability of generating a size of 64 and an 11% chance of generating 128. The stride pattern is similiar, with a 14% chance of generating a stride of -264 when the last stride was 64. Our Markov chain does not capture the exact sequence of values (e.g., strides or sizes), but accurately recreates the overall behaviour of the leaf node. Because each feature is modeled independently, there is a chance of producing a 128B size for an address that is not 81002**EB8**.

Capturing the exact behaviour of requests in a partition would reveal proprietary details of the overall execution flow of the IP workload. Instead our goal is to accurately approximate the behaviour of the original workload. Approximating the behaviour is lower overhead and helps obfuscate details of the workload, which is beneficial when the details are proprietary. We obfuscate in two ways: (1) Using Markov chains and (2) modeling features independently. A more accurate model with dependent variables would leak more information by revealing correlations between memory features as compared to our independent modeling approach. To minimize error, we save the address range and starting address of each leaf node to aid in generating the sequence. We also save the time that each partition should begin sending requests, which we use in conjunction with our McC model for delta times. Next, we show how to use this data with the McC models to minimize error in the access pattern and recreate the injection process.

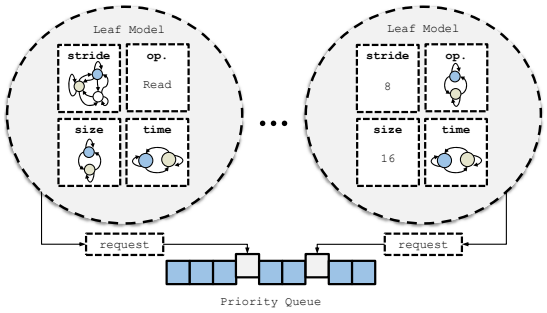


Fig. 5: Generated requests are pushed into a priority queue.

C. Synthesizing Requests

Each model in our statistical profile generates a partial order of requests. Each model contains only a partial order because requests from different spatial partitions are interleaved in time. The synthesis step must both generate the requests and reconstruct their total order. Synthesis must also capture the bursts and idle phases of the injection process. In this section, we first discuss how to generate a request, then how to reconstruct a total order that has time-varying behaviour.

Generating a Request. Each McC model captures the behaviour of a particular feature (delta time, stride, operation, and size). We use each McC model to generate the feature of the next request, either by using a constant value or based on the initial state and transition probabilities of the Markov chain. When using the Markov chain for a feature, we ensure *strict convergence*, following a similar methodology to prior work [3], [6]. When using strict convergence with Markov chains, each time we transition to a particular state we adjust (lower) the probability of doing so again the next time (down to a minimum of 0%). For example, for Table I, strict convergence ensures that only two 128 sizes and ten 64 sizes are generated.

We treat address generation as a special case. Once a stride is generated and added to the last address we synthesized, we check if the new address lies within the address range of the model's memory region. If it does not, we modulo the address back into the range to preserve spatial locality.

The Injection Process. Each model provides a start time (cycle count) and a McC model for the delta time, and pushes their generated requests onto a priority queue (Fig. 5). The priority queue is sorted by timestamp to create a total order of requests. Bursts in the injection process are captured because each model generates requests that may overlap in time. For example, multiple models may have similar start times to guarantee a burst. It is important to capture these bursts for two reasons. First, a burst puts strain on buffers in the memory hierarchy (e.g., the read and write queues found in memory controllers). Second, these bursts contain requests from different spatial partitions which can stress the memory system. For

example, the requests may need to go to different memory controllers, putting strain on the interconnection network [4]. Or many requests in the burst need to go to the same memory controller, offering additional opportunities to the memory scheduler [36]. We discuss metrics related to queue length and memory scheduling further in Sec. IV.

The use of start times and delta times, in conjunction with a priority queue, differs from previous approaches. We experimented with modeling the transitions between models, as done in prior art, but this leads to random behaviour. In addition, it added complexity to the statistical profile by requiring each parent to capture a transition model for its children. The use of a priority queue helps keep all models completely independent and allows concurrent address streams to be synthesized without having to track which address stream generates the next request.

Simulator Feedback. The timestamps of the requests in the priority queue give the relative time between memory requests. During a simulation, however, it may not be possible to inject these requests due to backpressure. In our simulations, we accumulate backpressure delay and add the latency to the timestamps of requests in the priority queue. This allows our simulations to adapt to the varying amount of contention found in the interconnect and memory hierarchy.

D. Summary

Hierarchical partitioning and dynamic spatial partitioning reveal underlying patterns that are difficult to model when considering all requests at once. This allows us to use a comparatively simple model, McC, to effectively recreate the behaviour of requests within a partition. In general, we recommend partitioning temporally before spatially. Spatial partitions have different durations; it is difficult to select a universal number that temporally partitions variable-sized time intervals of fine- and coarse-grained memory regions. In this paper, we focus on two-level hierarchies that partition temporally first. We use insights from prior art to appropriately size our time intervals [3], [4] and explore our model's sensitivity to length of these time intervals in Sec. IV-C. We validate that our two-level hierarchical model recreates the behaviour of CPU, DPU, GPU, and VPU devices. Sec. IV models requests as seen in the network before arriving at the memory controller. Sec. V models requests between the CPU and L1 cache, stressing our dynamic spatial partitioning approach to uncover memory regions that have not yet been filtered by any memory component.

IV. Validating Mocktails

The goal of Mocktails is to dynamically generate a request sequence to plug into your simulator in lieu of a detailed simulator model of a proprietary IP block. In this section, we validate Mocktails against proprietary traces from SoC compute devices. To validate Mocktails, we examine the interaction of our generated requests from each IP block with the memory controller and compare this against statistics obtained from running real applications on the real device. In addition, we show that we can incorporate previous models with Mocktails.

TABLE II: Proprietary traces.

Name	*Device*	*Description*
Crypto	CPU	A cryptography workload (2 traces).
CPU-D	CPU	A workload that interacts with a DPU.
CPU-G	CPU	A workload that interacts with a GPU.
CPU-V	CPU	A workload that interacts with a VPU.
FBC-Linear	DPU	Display compressed frames (linear mode, 2 traces).
FBC-Tiled	DPU	Display compressed frames (tiled mode, 2 traces).
Multi-layer	DPU	Display multiple VGA layers.
T-Rex	GPU	T-Rex from GFXBench (2 traces) [11].
Manhattan	GPU	Manhattan from GFXBench [11].
OpenCL	GPU	An OpenCL stress test (2 traces).
HEVC	VPU	Decoding compressed video (3 traces).

Specifically, we use STM in place of our McC models for the address and operation features [3]. Finally, in Section IV-C, we explore how sensitive Mocktails is to the hierarchical configuration used during modeling.

A. Methodology

To evaluate our technique with proprietary IP blocks, we used memory traces of CPU, DPU, GPU, and VPU devices for state-of-the-art SoC platforms from an industry partner. Table II describes these proprietary traces. The traces were collected through RTL emulation with probes that monitored the memory requests injected into the interconnect.[1] In the case of the CPU and GPU, these requests come from multiple cores after being filtered by the cache hierarchy. The CPU and GPU are connected to the memory system via a cache coherent interconnect, while the DPU and VPU are connected with a different, non-coherent interconnect. Note that RTL emulation is expensive and time consuming; some steps were taken to reduce run time (e.g., VPU traces had their inputs down-scaled). Our main goal is to validate that Mocktails recreates the different behaviour exhibited by compute devices found on SoCs; this can be effectively achieved with down-scaled inputs and/or shortened traces.

To create a common platform for validation, we simulate these traces in gem5 to obtain reference memory statistics.[2] The simulation connects a traffic generator (that parses trace files) to main memory through a crossbar and does a cycle-accurate simulation of the network and memory system. Mocktails also uses the traces to generate its statistical profiles for a two-level hierarchy that partitions temporally first, then spatially. Temporally, we use 500,000 cycles per execution phase as found in SynFull [4]. Spatially, we use our novel dynamic partitioning scheme. We call this configuration `2L-TS`..

[1]Trace collection is orthogonal to Mocktails. For example, traces can be obtained through hardware performance counters, RTL or simulation.

[2] Note: Our use of Mocktails in this section corresponds to Option A in Fig. 1 where our trace generator takes the Mocktails profile and makes a synthetic trace that gets fed into gem5. Alternatively, one could feed the statistical profile directly into the simulator to allow a more tightly coupled feedback mechanism between address generation and memory system and interconnect backpressure.

TABLE III: Memory configuration.

Parameter	*Value*
Number of Channels	4
Ranks per Channel & Banks per Rank	1 & 8
Burst Size	32 bytes
Read & Write Queue Size	32 & 64 bursts
High & Low Write Threshold	85% & 50%

Leaf nodes are modeled using McC for each feature (`2L-TS (McC)`), which we also compare with an STM model for the operation and stride features (`2L-TS (STM)`). We compare to STM using the same hierarchical configuration, allowing us to explore the importance of capturing the stride history when using dynamic spatial partitioning and how best to model reads and writes. As our hierarchical partitioning results in smaller subsets of requests in each leaf than considered in the original paper, we use smaller tables. Specifically, we use 32 rows for the stack distance table and consider, at most, the last 8 strides. Note that strict convergence ensures that both McC and STM models produce the exact number of reads and writes as found in the baseline trace. In Sec. IV-B, we compare the accuracy of McC and STM models for memory controller metrics.

Table III show the main memory configuration we use in gem5. We focus on memory controller metrics because they are sensitive not only to the pattern of memory accesses but also to the time memory accesses arrive [10], [12], [17], [36]. In gem5, a memory controller is made up of a read and write queue for incoming requests [17]. Large memory requests are divided into smaller packets to match the DRAM interface (Read Bursts, Write Bursts). Scheduling determines when requests are serviced, which impacts the row hits and the length of the queues. Our evaluations use first ready, first come first serve scheduling (FR-FCFS) with an open adaptive page policy. The open adaptive policy exploits row locality, but also dynamically decides when to close a page (a non-adaptive policy waits until there is a bank conflict) [17]. Our simulations uses a write drain model; writes are only serviced at certain thresholds to increase scheduler's opportunity to exploit parallelism. Our evaluation considers several metrics to validate that we recreate the spatio-temporal behaviour of the proprietary workloads.[3]

B. Results and Analysis

In this section, we show that Mocktails accurately generates requests from heterogeneous devices. Our validation looks at multiple memory controller metrics that demonstrate how well Mocktails captures the spatio-temporal behaviour of each memory request feature. For example, the size feature impacts the number of DRAM bursts created. Most metrics are impacts by multiple aspects of memory access behaviour and validate our Mocktails approach. For example, the length of the read and write queues for each memory channel is impacted by all four features of a memory request. Overall, the McC models combined with hierarchical partitioning accurately adapt memory request generation to the workload and device. Sec. IV-C analyzes how sensitive Mocktails is to different hierarchical configurations.

[3]Note: the goal is not to evaluate the goodness of this configuration but to validate that our synthetic traffic is an accurate recreation of the IP applications.

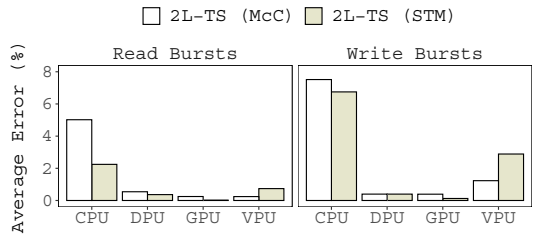


Fig. 6: Average error per device for the number of DRAM bursts.

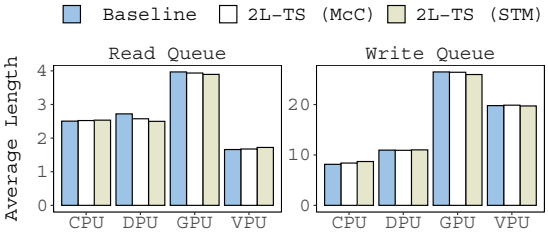


Fig. 7: Average read and write queue length for each SoC device.

DRAM Bursts. The memory controller divides requests into packets to match the DRAM's burst length [17]. The size feature of memory requests impacts the number of bursts; this feature is modeled the same way for McC and STM. Fig. 6 shows the geometric mean error of read and write bursts for each SoC device. The differences between STM and McC stem from how operations are modeled. Mocktails with McC has low error across multiple devices with the highest average error of only 7.5% for write bursts from the CPU.

The error stems from modeling features independently. The operation type and size features each have their own McC model; therefore, there is a small probability that Mocktails synthesizes requests with an operation-size pair that did not exist in the original workload. The probability is higher on CPU and VPU workloads because memory regions are more likely to have both read and write operations, impacting the read and write bursts in Fig. 6. In contrast, both the read-write and size patterns for DPU and GPU memory regions are more structured and easily captured with a Markov chain. The error could be further reduced by building a more detailed model of read and write interleaving; such a model might reveal correlations that the vendor would prefer to obscure and is likely unnecessary given the already low error.

Queue Length. At the memory controller, DRAM bursts go into their respective read or write queue. The queue lengths validate how well our injection process recreates burstiness in terms of the number of requests as well as how those requests are distributed spatially; spatial behaviour exercises different memory channels. Fig. 7 shows the average read and write queue length of each SoC device. On average, the read queue contains approximately 3-4 requests. The write queue is typically longer: ∼18 requests on average; write drain mode waits until certain thresholds are met before servicing writes,

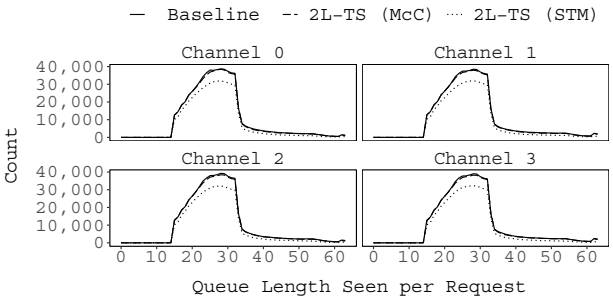


Fig. 8: Per memory controller distribution of write queue lengths observed by arriving requests for T-Rex1 GPU workload.

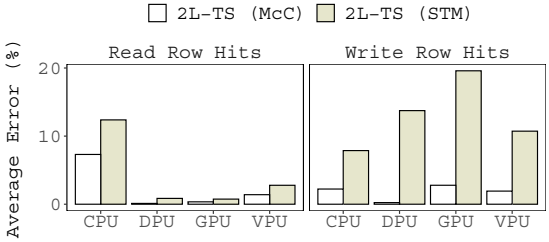


Fig. 9: Average error for read and write row hits for each SoC device.

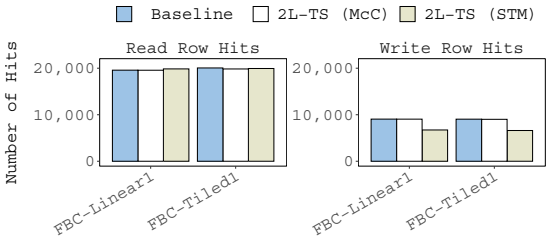


Fig. 10: Number of row hits when decompressing frame buffers on DPU.

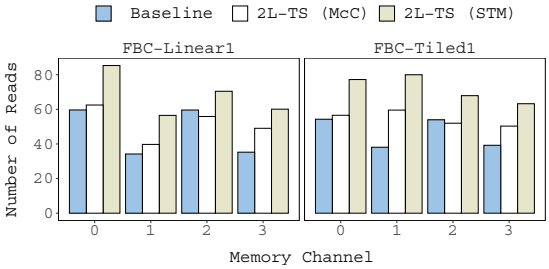


Fig. 11: The average number of reads sent to DRAM before switching to writes for each memory channel.

which we discuss in more detail later in this section.

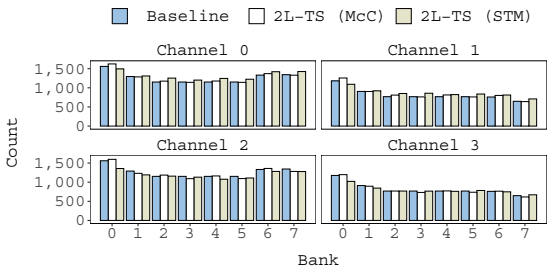
GPU workloads have longer average queue lengths because large requests are issued in short time intervals (i.e., burstiness). When a request arrives at the memory controller, packets are likely to already be in the queue. Fig. 8 shows the distribution of write queue lengths observed by a new request at each memory channel. The McC model captures the distribution of queue lengths well, but the STM model is slightly worse, which we explain further when we discuss read and write row hits. Fig. 8 also shows that Mocktails captures the spatio-temporal behaviour of the workload. Write requests must arrive at one of four possible destinations at the right time in order for the distributions to match (the same is true for read requests).

Row Hits. Open adaptive page policies try to maximize row hits by exploiting the row locality of requests in the read and write queues [17]. Fig. 9 shows the geometric mean error of read and write row hits for each SoC device. Overall, the McC model is more accurate than the STM model, implying that the need for modeling stride history is diminished thanks to dynamic spatial partitioning. Dynamic spatial partitions greatly reduces the variance in observed strides. STM's error stems comes from the fact that the operation is modeled based on one probability value, which does not capture patterns in the order of reads and writes. A memoryless Markov chain, used in McC, is enough to capture the performance of write row hits (a maximum error of 2.8% for GPU workloads).

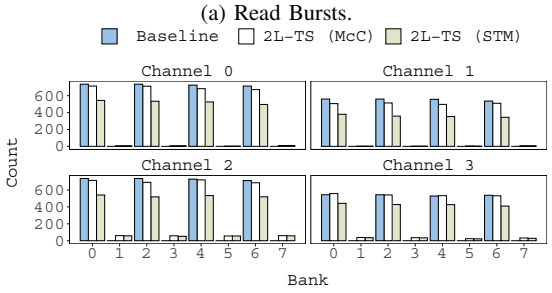
We show the importance of capturing read and write patterns by comparing DPU workloads. Fig. 10 shows the total number of row hits for a linear and tiled approach to accessing memory. The linear approach differs from the tiled approach in the address access sequence, which impacts the sequence of strides being modeled by McC and STM. Both McC and STM capture the stride pattern well; the addresses arriving at the memory controller mimic the behaviour of the original workload. However, the operation synthesized by STM does not accurately mimic the behaviour of the original workload because the write requests belong to different rows, reducing row locality and degrading the hit rate (over 25% error on write row hits). McC is more accurate (less than 1% error) because both the sequence of addresses and operations being synthesized closely match the original workload.

Another way to evaluate how well the operation feature is captured is by monitoring the memory controller's read/write switching policy. Writing to DRAM requires switching the direction of the bus used for data transfer; this takes time and frequent switching is undesirable. One solution is a write drain mode where writes are delayed until a certain threshold [17]. We saw evidence of this earlier when discussing queue lengths, where the average write queue length was longer than the read queue length. Here, we are interested in the number of reads that are sent to DRAM by the memory controller before switching to write requests (i.e., reads per turn around). Reads per turn around is impacted by both the number of requests in the read queue that map to the same row and the number of requests in the write queue.

Fig. 11 shows the average number of reads per turnaround for each memory controller using the same DPU workloads. Overall, McC (4% to 56% error) is always more accurate than STM (18% to 110% error), which underscores the importance of capturing read-write behaviour. While McC is more accurate than STM, the injection process is still a source of error. The injection process attempts to order requests from all leaf nodes based on the node's start time and model of delta time. This was done so that each node can be modeled independently, but it approximates the total order of requests. The approximate total order impacts when events, such as dynamically switching



(a) Read Bursts.



(b) Write Bursts.

Fig. 12: The number of read or write bursts arriving at each bank for the FBC-linear1 DPU workload.

to writes, occur. Despite this approximation, Mocktails captures the overall trend in complex memory controller metrics that are sensitive to the scheduling policy.

Per Bank Accesses. Thus far we have discussed the importance of capturing row locality. A good model should synthesize addresses that map to banks in the same manner as the original workload and device. Accurately capturing bank accesses is critical as bank conflicts (or lack thereof) will have a significant impact on DRAM performance. Each bank has its own open row and allows the memory controller to request data concurrently to improve performance [10]. Fig. 12 shows the number of read and write bursts arriving at each bank. Overall, McC accurately captures the memory access pattern with respect to different banks. Fig. 12b shows that the baseline does not issue any writes to certain banks in any of the memory controllers. The error for McC and STM here is due to modeling the operation and address feature separately.

Summary. In Mocktails, there are two ways in which the order of requests arriving at the memory controller may differ from the original workload. The first is caused by concurrent leaf nodes, where the injection process intersperses requests from different partitions. These requests enter the read or write queue, allowing the memory controller to exploit any parallelism it may find. The second is caused from the individual requests synthesized within a partition. Probabilistic models do not guarantee that a synthetically generated sequence matches the original sequence. In addition, the models for each memory request feature are independent, which creates a wider spectrum of memory requests than originally found in the workload.

Mocktails is highly accurate despite these subtle changes in the total order of requests. Hierarchical partitioning uncovers patterns that are recreated well by Markov chains, which

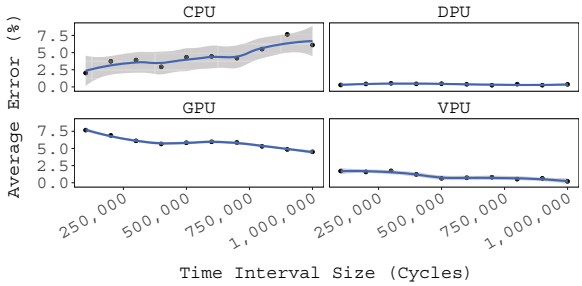


Fig. 13: Memory access latency sensitivity (variance shown in gray) to different sizes of time intervals.

results in accurate partial orders for many leaf nodes. This translates into recreating the very different behaviours of heterogeneous SoC devices without making any assumptions about the hardware or the workload.

C. Memory Access Latency and Sensitivity Analysis

Mocktails dynamically partitions requests *spatially*, but we rely largely on prior art to select the length of *temporal* execution phases. We use 500,000 cycles based on SynFull [4]. In this section, we study the sensitivity of Mocktails to the temporal partition length. Fig. 13 plots the average error for the average memory access latency. We use the `2L-TS` configuration and sweep the temporal partition size from 100,000 to 1,000,000 cycles. We plot the trends for each of the four SoC devices. The shaded region shows the variance in error (there are multiple traces for each device). Overall, the error is low (less than 8%) for all cycle counts. Error for CPU traces increases as we consider larger temporal partitions because more requests are grouped together in the dynamic spatial partitions. In different execution phases, a CPU application may use a certain memory region differently than in an earlier phase. This contributes to the error because the requests synthesized within a partition are not as accurate as for smaller execution phases. Other devices do not exhibit the same behaviour as the CPU (the main motivation for this work). Finally, while this section focused on low-level memory controller metrics, Fig. 13 summarizes the overall validity of Mocktails by demonstrating low error on overall memory access latency.

V. Comparing to Prior Work

This section focuses on the CPU to facilitate comparison with prior work. Specifically, we compare against HRD, which includes only a reuse distance model but at multiple block granularities to capture the spatial behaviour [28].

A. Methodology

We follow the methodology used by HRD to accurately recreate their results. Using Pin [27], we collect traces for 23 SPEC CPU2006 benchmarks with the reference input set. We fast-forward for 10 billion instructions before collecting memory requests. We stop trace-collection once 250 million

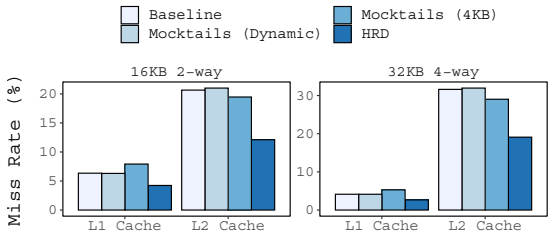


Fig. 14: Cache miss rates (geo. mean) for two cache configs.

additional instructions are executed. Our traces contain the following features: running instruction count, operation, memory address, and request size.

We configure gem5 for atomic mode simulation (which disregards the timestamp feature, focusing only on the order requests arrive). We use a least-recently used replacement policy, a 256KB 8-way L2 cache, and a 64B block size while varying the configurations (size and associativity) of a write-back L1 cache. We selected this cache configuration to align our results with those generated by prior art in this area.[4] HRD was evaluated with traces of memory requests from the port between the CPU and L1 cache; we do the same here for consistency. In Mocktails, we create a two-level hierarchy that partitions temporally first, then spatially. Temporally, we use 100,000 requests per execution phase as found in STM [3]. Spatially, we compare our novel dynamic partitioning scheme (`Mocktails (Dynamic)`) with fixed-size 4KB partitions (`Mocktails (4KB)`). Our HRD model configuration matches the original paper: reuse is modeled at the 64B granularity first and, in the event of a cold miss (i.e., reuse distance of infinity), reuse is then modeled at the 4KB granularity [28]. In addition, we do not divide requests into different phases to remain consistent with the original work.

B. Results and Analysis

Fig. 14 compares the miss rates of each technique against the baseline for two cache configurations: a 16KB 2-way and 32KB 4-way L1 cache. We see that `Mocktails (Dynamic)` closely matches the baseline and `Mocktails (4KB)` is slightly worse. The main reason for Mocktails out-performing HRD is temporal partitioning. HRD considers all requests in the entire trace, while Mocktails uses fixed-size intervals of 100,000 requests as done in prior art [3], [31].

`Mocktails (Dynamic)` has the lowest error across all the cache metrics evaluated. `Mocktails (4KB)` has more error due to the looser bounds on the address range of spatial partitions. For example, requests within a 4KB memory region may not touch the entire address range, but the McC model for strides generates addresses anywhere within the 4KB block. Conversely, requests within a dynamic memory region are guaranteed to touch the entire address range; the McC model for strides only generates addresses within a very tight address range. Overall, this results in the lowest error for the cache footprint (2.7%), L1 miss rates (5.6%), and L2 miss rates (2.6%) across all benchmarks and cache configurations.

[4]Our goal is not to endorse a particular cache hierarchy; modern CPUs contain L3s, but as we recreate requests between the CPU and the L1, an L3 is irrelevant to our analysis.

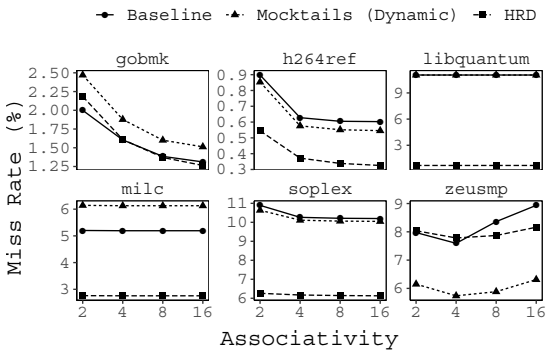


Fig. 15: Varying L1 cache associativities (2, 4, 8, and 16).

Cache Replacements. Prior work frequently uses reuse distance models because they capture temporal locality [3], [28], [31], [32], [46]. Mocktails is accurate for CPU devices even without a reuse distance model because it separates out interleaved requests from different address ranges. The variance in strides within a spatial partition is typically low (and sometimes constant), which minimizes the error for synthesized addresses. The priority queue combines the requests from multiple partitions into a total order of requests that faithfully captures cache performance metrics. Moreover, this is done without relying on any instruction-level information, such as the PC [32] or instruction dependencies [46].

To further demonstrate that Mocktails captures temporal behaviour, we explore L1 caches with different associativities and LRU replacement. Fig. 15 shows a 32KB L1 cache's miss rate for six SPEC benchmarks across four associativities: 2, 4, 8, and 16. Both HRD and Mocktails capture the three trends: First, increased associativity may decrease the miss rate (e.g., `gobmk`). Second, increased associativity may have no impact on the miss rate (e.g., `libquantum`). Third, increased associativity may increase the miss rate (e.g., `zeusmp`). Mocktails captures these trends because the dynamic spatial partitions have variable-sized time intervals with different start times. This results in 5.6% error on the number of L1 replacements across all benchmarks and cache configurations.

Cache Write-Backs. To accurately capture the behaviour of reads and writes, HRD employs a multi-state model for operations with explicit states for clean and dirty memory locations. Conversely, Mocktails only uses a McC model for operations. Fig. 16 shows the same cache configurations as Fig. 15, but measures the number of write-backs. Despite using a model that does not explicitly differentiate between clean and dirty memory locations, Mocktails captures the same trends as HRD.

Mocktails' fidelity (6.9% absolute error overall on L1 write-backs) stems from hierarchical partitioning: First, memory regions with different read-write behaviour are modeled independently (dynamic spatial partitioning). Second, read-only phases for a memory region are also modeled independently (temporal partitioning). The use of hierarchical partitioning means that Mocktails relies on the same, flexible McC model

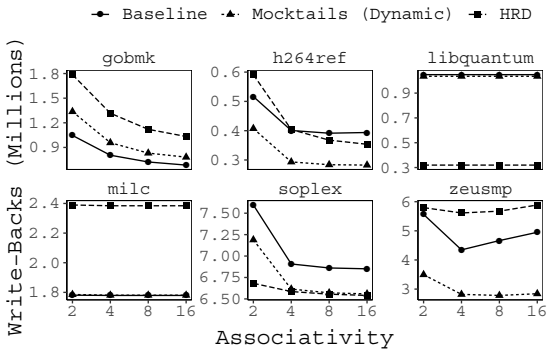


Fig. 16: The number of write-backs for a 32KB L1 cache.

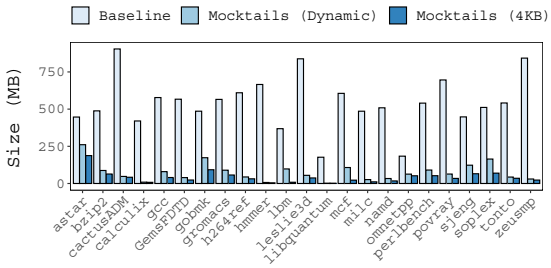


Fig. 17: The file sizes of traces and Mocktails models.

for operations as it does for strides. We find that this observation holds for other features, which is the reason Mocktails has high fidelity for CPU devices despite modeling them as a black-box.

Metadata Overhead. The metadata required by a Mocktails profile is a function of how many leaves it models. For each leaf, the overhead varies depending on whether constants or Markov chains are used for strides, operations, and sizes. Leaf nodes also consist of a starting time/address and request count. Figure 17 compares the amount of metadata required by Mocktails to the original trace sizes for SPEC CPU2006 benchmarks. We show the file sizes of profiles using both dynamic and fixed-size partitioning. Dynamic spatial partitioning results in larger profiles (1.5 to 261 MB, 50 MB on average) because it creates more fine-grained partitions. Fixed-size profiles (1.5 to 187 MB) are, on average, 20 MB less because they allow for sparse partitions, which reduces fidelity.

Both the trace and the model use Google's protobuf technology [16]; files are in an encoded binary format and compressed via gzip. Mocktails produces models that are smaller than a trace, sometimes by orders of magnitude (e.g., `calculix`, `hmmer`). The amount of metadata required for Mocktails is a trade-off between how many random variables are modeled with a constant versus how many requests each leaf node models. For example, in `calculix`, nearly half of the requests in the majority of temporal partitions belong to a single dynamic spatial partition, which significantly reduces the number of leaf nodes in the profile. In `hmmer`, many of the spatial partitions model a combination of strides, operations, and/or sizes with a constant value; the amount of metadata required increases if large (i.e., high number of states) Markov chains need to be stored. For example, `astar` has a large number of leaves, but the main reason it takes up more metadata than the other benchmarks is due to high variability in strides.

Overall, Mocktails profiles are 84% smaller than trace files. This size reduction is particularly important when considering larger and longer running applications that may produce very large traces that would be particularly cumbersome to store or distribute. HRD (not shown in the figure) requires the least amount of metadata; the model consists of only two histograms (64B and 4096B granularities) and a model for read/write operations. In contrast, Mocktails models additional features (e.g., request sizes, timestamps) that are not considered in HRD.

VI. Discussion

In this section, we discuss some use cases for Mocktails as well as its limitations. Mocktails enables the exploration of the memory hierarchy in heterogeneous SoCs. Our evaluation validates memory controller metrics related to the dynamic scheduling of requests to off-chip memory. Architects can use Mocktails to explore optimizations at the memory controller, such as: the scheduling policy, page policy, and read-write switching policy. For example, ChargeCache reduces the latency of off-chip requests by exploiting temporal row access locality [18]. ChargeCache is evaluated for CPU workloads, but Mocktails enables an evaluation with heterogeneous SoCs to determine if non-CPU devices also benefit from the design. Our evaluation also validates cache metrics, allowing research into appropriate cache sizes, the number of levels in a cache hierarchy, and replacement policies. Although Mocktails focuses on the memory system, it can provide insights to the IP block designers; for example, if the traces generated do not saturate the available memory bandwidth, than more parallelism can be introduced into the accelerator to fully exploit the memory capabilities. If row buffer locality is poor, IP designers may want to try and modify the access pattern of their designs. We envision the primary beneficiary of Mocktails to be academics who do not have access to proprietary IP blocks in SoCs; however, they can also be shared across companies so that one IP vendor can understand how their design will interact in the memory system with proprietary designs made by another company.

Mocktails models four features of a memory request: time, address, size, and operation. Another important feature not modeled by Mocktails is the data being communicated. Modeling data may give rise to privacy concerns; however we envision that techniques such as differential privacy [14] could be applied to obscure sensitive information while allowing patterns to be discerned. Modeling data would enable memory hierarchy research that exploits data value locality, such as: approximate computing [37], value prediction [26], and compression [34], [44]. Modeling the data feature in a way that hides proprietary information while enabling research in value locality is important. Mocktails' hierarchical partitioning can complement future models by uncovering patterns in the data feature once differential privacy has been applied. This is left for future work.

VII. Conclusion

Heterogeneous SoCs include many compute devices for general and special purpose computing. Research into SoC memory systems needs to model these devices to fully evaluate their designs. However, detailed architectural models of proprietary accelerators are not available in academia. In this paper, we propose Mocktails to create black-box models of these IP blocks. Mocktails recreates the memory access behaviour of heterogeneous compute devices. A hierarchy is created by dividing along both temporal and spatial dimensions, which reveals patterns in subsets of requests and adapts to the device being modeled. Using proprietary traces of IP blocks, Mocktails accurately captures the dynamic behaviour of memory access scheduling for many metrics including read row hits (at most 7.3% error) and write row hits (at most 2.8% error). We also show that Mocktails accurately captures the same trends as statistical simulation techniques for CPU devices (HRD)–capturing several cache metrics including: the miss rate, footprint, number of replacements, and number of write-backs. Despite making no assumptions about the underlying compute device, Mocktails captures the memory behaviour of CPUs, DPUs, GPUs, and VPUs. With the growing degree of heterogeneity in SoCs, Mocktails' black-box approach serves as a useful bridge between academia and industry.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback on this work. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, the University of Toronto and the Canada Research Chairs Program.

References

[1] "Enabling the next mobile computing revolution with highly integrated ARMv8-A based SoCs," Arm and Qualcomm, Tech. Rep., 2014.

[2] M. S. B. Altaf and D. A. Wood, "LogCA: A high-level performance model for hardware accelerators," in *Proceedings of the 44th International Symposium on Computer Architecture*. ACM, 2017, pp. 375–388.

[3] A. Awad and Y. Solihin, "STM: Cloning the Spatial and Temporal Memory Access Behavior," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*. IEEE, 2014, pp. 237–247.

[4] M. Badr and N. Enright Jerger, "SynFull: Synthetic traffic models capturing cache coherent behaviour," in *Proceedings of the International Symposium on Computer Architecture*. IEEE, 2014, pp. 109–120.

[5] M. Badr and N. Enright Jerger, "A look at computer architecture evaluation methodologies," in *The 2nd Workshop on Pioneering Processor Paradigms (WP3)*, 2018.

[6] G. Balakrishnan and Y. Solihin, "WEST: Cloning Data Cache Behavior Using Stochastic Traces," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture*. IEEE, 2012, pp. 1–12.

[7] B. T. Bennett and V. J. Kruskal, "LRU stack processing," *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, 1975.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008, p. 72.

[9] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "GemDroid: a framework to evaluate mobile platforms," in *ACM SIGMETRICS Performance Evaluation Review*, 2014.

[10] H. Choi, J. Lee, and W. Sung, "Memory access pattern-aware DRAM performance model for multi-core systems," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, 2011, pp. 66–75.

[11] "GFXBench," https://gfxbench.com/benchmark.jsp, CompuBench.

[12] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proceedings of the 26th International Symposium on Computer Architecture*. IEEE, 1999, pp. 222–233.

[13] P. J. Denning, "The locality principle," *Communications of the ACM*, vol. 48, no. 7, pp. 19–24, July 2005.

[14] C. Dwork, "Differential privacy: A survey of results," *Theory and Applications of Models of Computation*, pp. 1–19, 2008.

[15] K. Ganesan, J. Jo, and L. K. John, "Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and Implant-Bench Workloads," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, 2010, pp. 33–44.

[16] Google, "Protocol buffers," https://developers.google.com/protocol-buffers/?csw=1.

[17] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, "Simulating DRAM Controllers for Future System Architecture Exploration," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, 2014, pp. 201–210.

[18] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM latency by exploiting row access locality," in *International Symposium on High Performance Computer Architecture*, 2016.

[19] M. Hill and V. J. Reddi, "Gables: A roofline model for mobile socs," in *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, 2019, pp. 317–330.

[20] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the International Symposium on Computer Architecture*. ACM, 2009, pp. 152–163.

[21] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU performance modeling technique based on interval analysis," in *Proceedings of the 47th International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 268–279.

[22] A. Joshi, L. Eeckhout, R. H. Bell, and L. K. John, "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks," in *Proceedings of the International Symposium on Workload Characterization*. IEEE, 2006, pp. 105–115.

[23] M. Kiani and A. Rajabzadeh, "Efficient cache performance modeling in GPUs using reuse distance analysis," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 4, pp. 58:1–58:24, 2018.

[24] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for Variable Length Intervals and Hierarchical Phase Behavior," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, 2005, pp. 135–146.

[25] J. Lau, S. Schoenmackers, and B. Calder, "Transition Phase Classification and Prediction," in *Proceedings of the 11th International Symposium on High Performance Computer Architecture*. IEEE, 2005, pp. 278–289.

[26] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[27] 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," in *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 190–200.

[28] R. K. V. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, "Fast and Accurate Exploration of Multi-level Caches Using Hierarchical Reuse Distance," in *Proceedings of the 23rd International Symposium on High Performance Computer Architecture*. IEEE, 2017, pp. 145–156.

[29] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[30] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*. IEEE, 2014, pp. 37–48.

[31] R. Panda and L. K. John, "HALO: A hierarchical memory access locality modeling technique for memory system explorations," in *Proceedings of the International Conference on Supercomputing*. ACM, 2018.

[32] R. Panda, X. Zheng, and L. K. John, "Accurate Address Streams for LLC and Beyond (SLAB): A Methodology to Enable System Exploration," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, 2017, pp. 87–96.

[33] R. Panda, X. Zheng, J. Wang, A. Gerstlauer, and L. K. John, "Statistical Pattern Based Modeling of GPU Memory Access Streams," in *Proceedings of the 54th Annual Design Automation Conference*. ACM, 2017, pp. 1–6.

[34] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[35] V. J. Reddi, H. Yoon, and A. Knies, "Two billion devices and counting," *IEEE Micro*, vol. 38, no. 1, pp. 6–21, Jan. 2018.

[36] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th International Symposium on Computer Architecture*. ACM, 2000, pp. 128–138.

[37] J. San Miguel, M. Badr, and N. Enright Jerger, "Load value approximation," in *International Symposium on Microarchitecture*, 2014.

[38] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin : A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceedings of the International Symposium on Computer Architecture*. IEEE, 2014, pp. 97–108.

[39] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *Proceedings of the International Symposium on Microarchitecture*, 2016.

[40] X. Shen, Y. Zhong, and C. Ding, "Locality Phase Prediction," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2004, pp. 165–176.

[41] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," in *Proceedings of the 30th International Symposium on Computer Architecture*. ACM, 2003, pp. 336–349.

[42] "SPEC CPU2006," https://www.spec.org/cpu2006, Standard Performance Evaluation Corporation.

[43] "SPEC CPU2017," https://www.spec.org/cpu2017, Standard Performance Evaluation Corporation.

[44] Y. Tian, S. M. Khan, D. A. Jimenez, and G. H. Loh, "Last-level cache deduplication," in *International Conference on Supercomputing*, 2014.

[45] Q. Wang, X. Liu, and M. Chabbi, "Featherlight reuse-distance measurement," in *International Symposium on High Performance Computer Architecture*, 2019.

[46] Y. Wang, G. Balakrishnan, and Y. Solihin, "MeToo: Stochastic Modeling of Memory Traffic Timing Behavior," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2015, pp. 457–467.

[47] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*. IEEE, 1995, pp. 24–36.

[48] J. Yin, O. Kayiran, M. Poremba, N. Enright Jerger, and G. H. Loh, "Efficient synthetic traffic models for large, complex SoCs," in *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, 2016, pp. 297–308.

[49] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 382–393.

[50] Y. Zhong, X. Shen, and C. Ding, "Program Locality Analysis Using Reuse Distance," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 6, pp. 1–39, 2009.