Efficient Synthetic Traffic Models for Large, Complex SoCs

Jieming Yin* jieming.yin@amd.com
Onur Kayiran* onur.kayiran@amd.com
Matthew Poremba* matthew.poremba@amd.com
Natalie Enright Jerger*† enright@ece.utoronto.ca
Gabriel H. Loh* gabriel.loh@amd.com
*Advanced Micro Devices, Inc. †University of Toronto

ABSTRACT

The interconnect or network on chip (NoC) is an increasingly important component in processors. As systems scale up in size and functionality, the ability to efficiently model larger and more complex NoCs becomes increasingly important to the design and evaluation of such systems. Recent work proposed the "SynFull" methodology that performs statistical analysis of a workload's NoC traffic to create compact traffic generators based on Markov models. While the models generate synthetic traffic, the traffic is statistically similar to the original trace and can be used for fast NoC simulation. However, the original SynFull work only evaluated multi-core CPU scenarios with a very simple cache coherence protocol (MESI). We find the original SynFull methodology to be insufficient when modeling the NoC of a more complex system on a chip (SoC). We identify and analyze the shortcomings of SynFull in the context of a SoC consisting of a heterogeneous architecture (CPU and GPU), a more complex cache hierarchy including support for full coherence between CPU, GPU, and shared caches, and heterogeneous workloads. We introduce new techniques to address these shortcomings. Furthermore, the original SynFull methodology can only model a NoC with N nodes when the original application analysis is performed on an identically-sized N-node system, but one typically wants to model larger future systems. Therefore, we introduce new techniques to enable SynFull-like analysis to be extrapolated to model such larger systems. Finally, we present a novel synthetic memory reference model to replace SynFull's fixed latency model; this allows more realistic evaluation of the memory subsystem and its interaction with the NoC. The result is a robust NoC simulation methodology that works for large, heterogeneous SoC architectures.

1. INTRODUCTION

Modern computer systems have evolved from relatively simple microprocessors to complex systems-on-chips (SoCs). Current systems already integrate CPUs, GPUs, networks on chips (NoCs), memory controllers [1, 2], and more. Looking forward, system sizes in terms of CPU cores and/or GPU compute units are likely to continue to scale to support increasingly complex processing for immersive virtual reality applications [3], "Big Data" and "Big Compute" [4], and more. In particular, recent industry papers point toward future exascale high-performance computing systems making use of heterogeneous compute nodes with extensive GPU capabilities [5, 6]. As future SoCs scale in both functional diversity (inclusion of GPUs or other accelerators) and size (number of CPU/GPU compute resources), the SoC's NoC and memory systems become increasingly critical components in determining the overall performance of the SoC.

To design effective NoC and memory systems for large heterogeneous systems, computer architects need tools to model the behavior and predict the performance of different candidate designs. However, conventional simulation tools driven by system emulation are too slow, and simulations driven by simple synthetic traffic patterns (e.g., uniform random injections), while fast, may not capture important application-dependent behaviors. One recently proposed methodology "SynFull" takes a first step toward providing the "best of both" [7]. SynFull takes traffic traces from detailed cycle-level simulation of application executions, and then analyzes these to create stochastic (i.e., synthetic) Markov model-based traffic generators that are statistically similar to the original applications in terms of representation of different program phases, distributions of message sources and destinations, per-node injection rates, etc. As a result, SynFull enables the efficient NoC simulation using fast synthetic models while still capturing critical application-dependent and time-varying behaviors absent from conventional simplistic synthetic traffic patterns.

For large heterogeneous SoCs, however, the current SynFull methodology is lacking in several dimensions. SynFull only considers multi-core CPU systems. However, with the proliferation accelerated processing units (APUs) with both CPUs and integrated GPUs, along with the growth in general-purpose GPU (GPGPU) computing, evaluations of future NoCs need to account for the differences that APUs and GPGPU applications introduce. Whereas SynFull considered a multi-core CPU with a simple MESI cache coherence protocol, modern APUs that support shared virtual memory between CPU and GPU components [8] use significantly more complex coherence protocols with many more states and message types. GPGPU applications, especially due to the presence of very distinct CPU and GPU phases, can cause the SynFull approach to generate NoC traffic with significant application-scale deviations. We propose a variety of extensions to SynFull to address these and other issues to provide a robust methodology capable of modeling NoC behaviors for GPGPU applications on APUs.

Especially in the context of large future systems, a key limitation of SynFull is that the trace collection and analysis performed on an N-node system can only create synthetic traffic generators for other N-node systems. If one needs to evaluate a system with $M \gg N$ nodes, then one must recollect and reanalyze traces from an M-node system. However, for large M, running a full simulation may take an intractably long time (or impractically large memory capacity); having enough applications that can meaningfully scale up to

978-1-4673-9211-2/16/$31.00

an M-node system poses another challenge. We introduce a methodology by which we can generate an "extrapolated trace" that is similar to a trace collected from an M-node system, which in turn can be fed to our version of SynFull to generate synthetic traffic models for the larger system of interest under both strong and weak scaling scenarios.

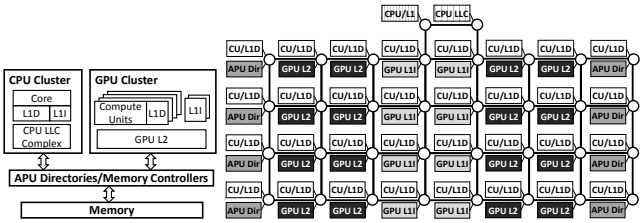
Finally, the original SynFull methodology focused only on the NoC and employed a simple constant-latency memory model. However, memory performance is dependent on a wide variety of factors related to the arrival time of requests, the distribution of requests among channels and banks, and myriad DRAM timing parameters. In the spirit of SynFull's synthetic modeling approach for the NoC, we introduce application-dependent synthetic memory models that capture critical memory-related behaviors such as bank conflicts/row-buffer locality and non-uniform distributions of requests among channels and banks.

Through all of these extensions and enhancements to the original SynFull, we arrive at an overall NoC and memory modeling methodology that can handle the full combination of heterogeneous systems, large scalable systems, and realistic memory systems. While we present results for a specific implementation based on the gem5 simulator [9], the methodology is general and the computer architecture research community can use this for NoC and memory studies of large APUs on other simulator platforms.

2. THE SYNFULL METHODOLOGY

SynFull [7] generates synthetic traffic that resembles the traffic of a homogeneous multi-core CPU with coherent caches. The first step is to find only a few phases of the execution that can accurately represent the network traffic of the whole execution. This is accomplished by using hierarchical clustering (at the macro and micro level) on an application trace that records all network injections. The trace is divided into macrophases each with a fixed duration (*macrophase length*). Using a clustering algorithm, these macrophases are grouped into clusters. Different features of the traffic can be used for clustering, such as node injection rates or source-destination flows. The median macrophase for each cluster is selected as the representative macrophase. Similarly within each macrophase, clustering is performed at the micro-level to bin each microphase (each phase is *microphase length* long) into a cluster. Microphases are designed to better capture the variations in traffic behaviors within individual macrophases. SynFull generates a synthetic traffic model for each cluster, and executes them at the granularity of both micro and macrophases. Transition probabilities calculated by analyzing the application trace govern the transitions between phases.

SynFull groups network traffic into *initiating messages* (e.g., a read or a write request) and *reactive messages* that are triggered by initiating messages. Reactive messages consist of forward requests, invalidate requests, and responses. The synthetic traffic model defines the behavior of initiating messages, forward requests, and invalidations for each cluster. For each initiating message type, three probability distributions govern its behavior. The first distribution generates a source node for a packet. Similarly, the second distribution generates a destination node. The third distribution generates the number of packets that will be injected during a micro phase. Forward messages are defined by a probability that a given message type will be forwarded, and a distribution of its possible destinations. Invalidations are similar to forward messages, but they are sent to multiple destinations that are generated by a different distribution. Responses are deterministic as defined by the cache coherence protocol, and therefore Synfull does not need to provide a synthetic model for them (although the underlying NoC simulator still generates these responses).



(a) Baseline system. (b) Baseline topology.

Figure 1: Baseline APU system.

In the next sections, we keep much of the original SynFull terminology and share its hierarchical approach of macro and micro phases and initiating and reactive messages. However, we make drastic changes to many of the underlying details to yield a new approach that is more accurate for large-scale heterogeneous SoCs and GPGPU workloads. Throughout the rest of the paper we refer to the original SynFull methodology simply as SynFull and our new methodology as APU-SynFull.

3. CHALLENGES OF A SYNFULL APU

In this section, we first present our baseline accelerated processing unit (APU) architecture, and then we enumerate the challenges associated with applying the SynFull methodology to APUs.

3.1 Baseline System

Fig. 1a shows an overview of our baseline APU system. The system contains both a CPU and a GPU cluster. The CPU cluster consists of CPU cores, private L1 caches, and a last-level-cache (LLC) complex. CPU caches are write-back and are kept coherent through a read-for-ownership MOESI directory protocol; the LLC keeps track of all the cache blocks in the CPU cluster with shadow tags. The GPU cluster consists of compute units (CUs), private L1 data caches, L1 instruction caches shared by every 4 CUs, and a banked unified L2 cache. GPU caches are write-through and write-no-allocate. L1 caches in the GPU are kept coherent by writing through dirty data and invalidating the caches at kernel launch. The CPU and GPU have a unified memory address space. The APU directories are responsible for keeping the CPU LLC and the GPU L2 cache coherent. All memory requests generated by the CUs access the APU directories to stay coherent with the CPU LLC. APU directories are connected to the memory controllers for off-chip memory accesses. The CPU and GPU clusters each have their own coherence protocols, and a system-level protocol (SLP) enables coherent communication between these clusters.

The topology of our baseline system is shown in Fig. 1b. The GPU cluster has 32 CUs that are connected by a 4×8 mesh. APU directories are placed along the left and right edges, and 8 GPU L1 instruction caches are located in the

center of the mesh. The GPU cluster contains 16 address-interleaved L2 cache banks. The mesh is augmented with two additional nodes for the CPU cluster. One node is connected to the CPU L1 cache, and the other is connected to the CPU LLC. The topology shown in Fig. 1b is used as a working example, but the methodologies proposed in this paper are not tied to this specific layout.

3.2 The Need for an Enhanced SynFull

In this section, we describe the limitations of the original SynFull methodology when applied to complex SoCs consisting of coherent CPU and GPUs.

3.2.1 Cache Coherence Protocol(s):

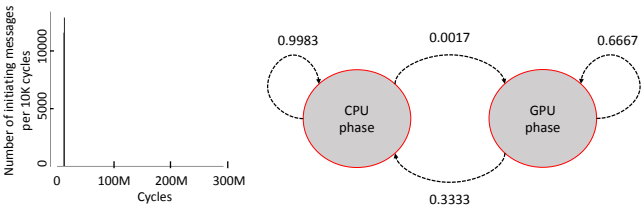
One of the challenges in simulating an APU using SynFull is the incorporation of a complex cache coherence protocol. While the SynFull methodology uses a MESI-like protocol, our baseline APU has CPU and GPU coherence protocols plus the global SLP. Compared to MESI's four stable coherence states, the combination of CPU, GPU, and SLP results in 18 stable states. SynFull uses four initiating message types while the initiating messages in our APU cannot easily be grouped into less than ten categories. For example, write requests from CPU and GPU are classified in different categories because CPU caches are write-back while GPU caches are write-through. In total, SynFull uses ten message types, whereas there are more than 80 message types for our APU. Although not all of the message types are modeled since we group them into fewer types where possible, the grouping must be able to abstract away most of the complexity inside the protocol while still being able to represent critical communication behaviors.

Another important challenge is to model the depth of a complex protocol. Protocol depth is defined as the longest chain of dependent messages generated by a single initiating request. It is dependent on the complexity of the protocol and the total number of levels in the memory hierarchy. In MESI, an initiating request usually results in two or three additional network messages before the coherence transaction is completed. For our APU, an initiating request might result in more than ten network messages during the entire transaction. Additionally, SynFull assumes a simple one-to-one mapping between requests and responses, which does not hold in our APU.

3.2.2 APU Workload Phase Behavior:

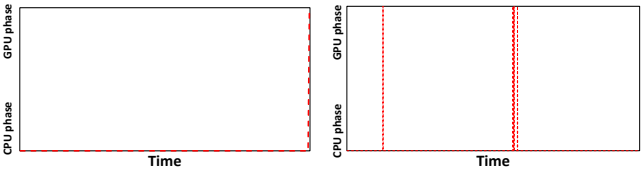
APU workloads have significantly different behaviors than CPU workloads. Current APU workloads typically consist of multiple phases, where in each phase either the CPU or the GPU cores are active.[1] The network traffic during a GPU phase is much higher than during a CPU phase. Fig. 2a shows the number of initiating messages generated by the BFS application over time. The big spike corresponds to the GPU execution where the network traffic is much higher compared to all the other phases where the GPU is idle. If the APU application offloads only a single kernel for GPU computation, as in this simple example, then there is likely to be a single spike in the network traffic similar to Fig. 2a.

[1] Future task-based APU applications may exhibit more concurrent execution of tasks across both CPU and GPU resources that would change the observation that phases predominantly exercise only the CPU or only the GPU.



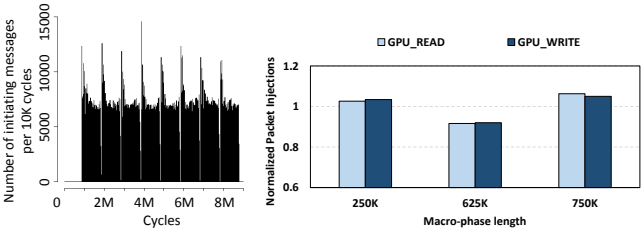
(a) The number of initiating messages over time. (b) Macro-level transition probabilities.

Figure 2: Phase behavior of BFS.



(a) 300M cycles. (b) 3B cycles.

Figure 3: Challenges of probability-based transitions



(a) The number of initiating messages in Hotspot over time. (b) Normalized packet injection rate per message type w.r.t. to the baseline, across different macrophase lengths.

Figure 4: Sensitivity to macrophase length.

Such phase behavior might lead to problems in SynFull. SynFull uses a probability-based transition model between macro phases. In cases where we observe a single spike, a small error in the ratio between the number of executed CPU and GPU phases might result in significant errors in the generated network traffic. This is exemplified by BFS in Fig. 2b which shows the transition probabilities between CPU and GPU phases.

A methodology that uses this probability-based transition model is also heavily dependent on the simulation length. If we simulate 300M cycles, as shown in Fig. 3a, the steady-state condition is reached after the first executed GPU phase causing the simulation to finish. In this scenario, the ratio between the executed CPU and GPU phases is very similar to that of the real execution, but the GPU spike comes at the end, although this is not how the real application behaves. This also requires a priori knowledge of the number of cycles to be simulated. If we run the simulation for 3B cycles (Fig. 3b), the spikes come at random times, and sometimes back to back; back-to-back traffic spikes may overwhelm the NoC, causing high latencies due to congestion that is not present in the real application. If the simulation is capped around 2B cycles, the ratio of GPU phases to CPU phases would be much higher than in real execution. Finally, if the simulation is limited to 150M cycles, the GPU phase is never observed. These results demonstrate that the probability-based transition model does not always emulate real application behavior, or could otherwise require very long simulation times to achieve Markov steady state.

3.2.3 *Sensitivity to Macrophase Length:*

While the original SynFull did not report a strong sensitivity of the methodology to the macrophase length for multi-core CPU applications, we found that this does not appear to hold for APU workloads. To demonstrate this, we execute the Hotspot benchmark using different macrophase lengths (microphase length is kept constant), as shown in Fig. 4a. Unlike BFS (Fig. 2a), Hotspot has a periodic injection pattern of approximately 1M cycles. Fig. 4b shows that choosing a macro-resolution of 250K cycles, which is very close to a quarter of the period between peaks, leads to a close match of the number of initiating messages per unit time compared to the original cycle-level simulation. However in such applications, choosing a poor macrophase length can result in the representative cluster misestimating the number of injected messages. For example, a macrophase length of 625K cycles results in most of the macrophases being misaligned with respect to the underlying traffic periodicity, resulting in approximately 8% fewer message injections. It is interesting to note that the macrophase length need not be perfectly chosen; Fig. 4b also shows an example where the macrophase length is 750K cycles. In this case, the macrophases line up with the underlying traffic patterns once every four macrophases; while the error is slightly greater than the 250K-cycle case, it is still relatively low. These results demonstrate when applications exhibit regular periodic behaviors, simulations will be more accurate if the period is carefully determined.

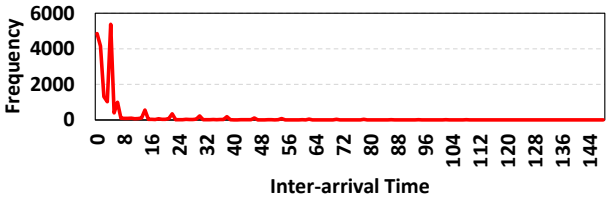
3.2.4 *Capturing Bursts:*

Section 3.2.3 showed that choosing a good macro-resolution is necessary to provide an accurate representation of the underlying traffic patterns; however, this alone is insufficient to achieve accurate latency results because it does not adequately capture the fine-grained bursty behavior of GPU applications.

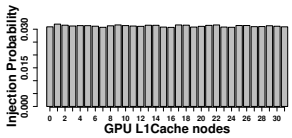
Temporal bursts: A SIMD compute unit executing a single vector-load instruction can generate up to 64 memory requests in a very short amount of time.[2] Fig. 5a shows the inter-arrival time of GPU read requests in the Hotspot benchmark. Most messages are injected very close in time, which results in a very bursty injection pattern. However, SynFull injects messages into the network at uniform intervals within a microphase. At typical microphase lengths of a few hundred cycles, this causes the bursts to be spread out too much and no longer captures the effects of executing the GPU's vector memory operations.

Spatial bursts: The traffic generated by the GPU in an interval of few hundred cycles (microphase granularity) is rarely spread out across all source and destination nodes. While a few source-destination pairs experience high traffic, others do not inject/receive any messages in the same interval (but do so in another interval). Fig. 5b and Fig. 5c show the spatial burstiness of GPU read requests within 500K-cycle and 250-cycle sampling windows, respectively. For the large sampling window of 500K cycles, GPU L1 cache nodes have similar probabilities for injecting requests, resulting in an evenly distributed network traffic pattern. However, during an interval of 250 cycles, less than half of the L1 cache nodes inject traffic; and we further observe that this traffic actually flows to less than half of the L2 cache nodes. In contrast to the case with larger sampling granularity where traffic is evenly distributed, parts of the interconnection network exhibit transient congestion caused by traffic bursts. Because SynFull considers the injection probability and traffic flow at a very large macrophase granularity, it is likely to generate uniform network traffic in a microphase if the source nodes have similar injection probabilities.

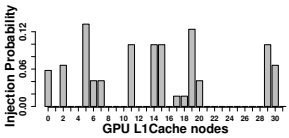
[2] 64 is a typical wavefront length. The actual number of requests per burst is typically less than the maximum possible due to request coalescing, branch divergence, etc.



(a) Frequency of different inter-arrival times of GPU read requests in Hotspot during its GPU phase.

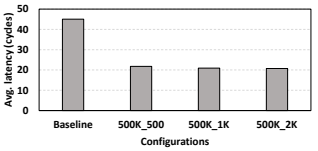


(b) Node injection probability with 500K-cycle sampling window.

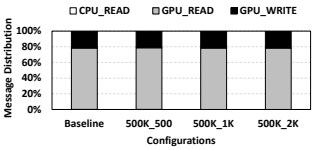


(c) Node injection probability with 250-cycle sampling window.

Figure 5: Temporal and spatial burstiness of Hotspot.



(a) Average latency for the baseline and different microphase lengths.



(b) Distribution of initiating messages for the baseline and different microphase lengths.

Figure 6: Effect of the default SynFull injection model on simulation accuracy in Hotspot.

Fig. 6 demonstrates how the inability to capture bursts affects the estimated latency in Hotspot. We set the macrophase length to 500K cycles, and compare the NoC behavior with microphase lengths of 500, 1000, and 2000 cycles against the baseline of full cycle-level simulation. Fig. 6a shows that for all chosen microphase lengths, the average NoC message latency is about 21-22 cycles while full cycle-level simulation exhibits much higher average latency. This happens despite the fact that for this benchmark SynFull actually does a good job of capturing the distribution of initiating message injections, as shown in Fig. 6b. These results show that the default SynFull injection model is not able to capture the bursty (both temporal and spatial) behavior of GPU workloads.

3.2.5 *Scaling:*

Computer architecture researchers are focused on developing and evaluating innovations for future systems. Given the scaling trends of CPU multi-/many-cores and aggressive GPU/APU architectures, future systems are likely to continue to increase in core counts. Evaluating these new designs requires tools that scale with the proposed systems. SynFull requires a model file that is generated from a trace file collected during a full cycle-level simulation. For Syn-

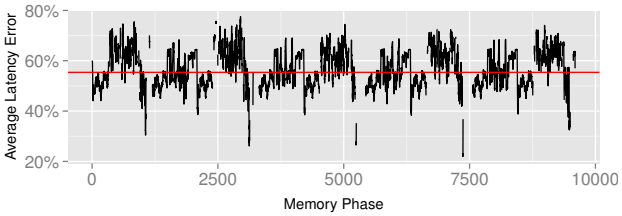


Figure 7: Average latency error of fixed-latency memory in CoMD compared to gem5 built-in model. Each phase on the x-axis represents a fixed number of memory requests.

Full to simulate larger systems, these trace files must first be generated from cycle-level simulations of correspondingly larger system. This is problematic for two key reasons. First, cycle-level simulation of the large system may not be practical due to excessively long simulation times, or if the system is large enough the memory requirements of the simulation may well exceed the host computer's resources. Second, because such large systems do not yet exist, many CPU and GPU applications have not yet been written/re-optimized in such a way so that they can scale to make use of the additional compute resources. So even if the cycle-level simulator could handle a much larger system, there may not be enough interesting workloads to run.

3.2.6 Memory:

Memory latency is a key contributor to overall performance especially during periods of high-burst traffic and directory misses. SynFull uses a fixed-latency memory model, with the latency chosen to generally represent the typical/expected latency of a single memory request (e.g., read). The memory latency is simply added to the latency of a directory response. This type of model does not capture the dynamic behavior of memory performance that results from DRAM timing parameters, bank conflicts, row buffer locality, etc. These factors lead to individual memory request latencies deviating from the mean. We compared the average memory latency between a fixed-latency memory controller and a cycle-accurate model across several applications using average percentage error in memory latency over a memory phase. One memory phase represents a fixed number of memory requests rather than time, which allows for better visualization of GPU kernel execution. Fig. 7 shows an example of the error in memory latency over several memory phases in CoMD, which executes multiple GPU kernels. The average error is 55.4% with no error less than 22.0% and as high as 77.5%. We show the average, minimum, and maximum error over all memory phases for nine benchmarks in Table 1. We see that the fixed-latency memory model performs poorly across a range of APU workloads, with an average error of 54.2% across all nine benchmarks.

SynFull abstracts away memory addresses to keep the model simple to focus on cache coherence protocol behavior. To model memory latency more accurately, we must have a model for generating the address of each memory request. Knowing the address enables the modeling of important DRAM behaviors such as bank conflicts that affect the latency of individual requests.

4. OUR NEW METHODOLOGY

In this section, we describe our new methodology to address the challenges identified in Section 3.2.

	Average	Minimum	Maximum
bitonic	50.4%	30.6%	68.6%
dct	49.2%	18.9%	68.6%
histogram	57.0%	26.6%	68.6%
matrixmul	65.1%	35.7%	69.0%
spmv	63.8%	48.6%	68.6%
comd	55.4%	22.0%	77.5%
bfs	32.8%	19.2%	68.5%
hotspot	59.1%	29.0%	86.4%
nw	55.4%	8.61%	87.1%

Table 1: Summary of average, minimum, and maximum percentage error when using a fixed latency memory controller.

4.1 Modeling Complex Coherence Protocols

The first challenge of evaluating an APU using SynFull is to model the more complex coherence protocols in APUs. We address this by classifying different types of initiating messages that trigger similar reactive flows into the same category. For example, a cache generates different types of coherence messages based on a cache line's state, the ones that are bound to the same destination(s) with the same message size can be collapsed into a single message type to reduce the complexity of the coherence model. In total, we model 53 message types after grouping the 86 message types, which is still substantially more than in the original SynFull.

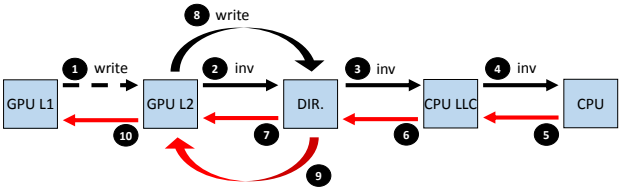


Figure 8: Protocol depth for a GPU write request.

The second challenge is tracking packet dependencies and modeling the protocol depth. SynFull assumes a simple one-to-one mapping between requests and responses, where a read request generates a forward request if data is cached on chip, and a write request generates invalidations if data is shared by multiple readers. In an APU with a combination of coherence protocols, a single initiating request can potentially result in multiple forward/invalidation requests. Furthermore, the generation of forward and invalidation packets in SynFull is based on forward and invalidation probabilities, while in a multi-level memory hierarchy, the probability at each level is independent of the others. Therefore, to model the transaction chain correctly, we introduce new forward/invalidation request types for each level of the memory hierarchy, considering their potentially distinctive behaviors. Consider the following example in which a write request initiates from a GPU core, while the CPU L1 cache has a copy of the same cache line. As shown in Fig. 8, the dashed arrow ❶ is the initiating GPU write request. Because GPU caches are write-through, two requests are generated: an invalidation request ❷, and later on a write request with data ❽. The invalidation request is required because the GPU L2 and CPU caches are kept coherent. After receiving the invalidation, the APU directory notices that the CPU has a copy of the data. Therefore, the directory forwards this invalidation to the CPU LLC ❸, and the CPU LLC forwards the invalidation to CPU L1 cache ❹. Notice that the coherence transaction ❸ is directly caused by ❷ (but not caused

by ❶), while transaction ❹ is directly caused by ❸ (but not caused by ❷). Therefore a forwarding probability for each individual step is required; and messages in ❷ and ❸ are classified into different message types. Once the acknowledgments, denoted by ❺, ❻, and ❼, are received by the corresponding controllers, the GPU L2 is allowed to update the memory with data ❽. It is worth pointing out that the generation of write packet ❽ is dependent on both ❶ and ❼, which breaks the one-to-one request-response mapping assumption in SynFull. Such dependencies must be tracked properly so as to correctly model a complex APU protocol. Finally, the memory update acknowledgements are sent back from directory to GPU L2 ❾, and from GPU L2 to GPU L1 ❿.

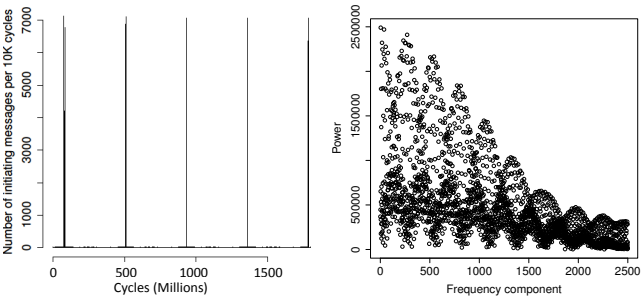
In summary, the differences between our model and the SynFull approach lie in the following aspects. First, we model a chain of transactions by introducing new packet classifications. Second, we introduce the necessary dependency tracking logic to handle the one-to-many and many-to-one mappings between requests and responses.

4.2 Deterministic Macrophase Replay

The probability-based macro-level transition model can lead to inaccuracies as described in Section 3.2.2. To ensure that the high-level behavior of our simulation is consistent with the workload behavior, we introduce a deterministic macrophase replay methodology. This approach records the sequence of cluster IDs of each macrophase. Note that this is a very lightweight approach that stores a single number for every n cycles (n is the length of a macrophase), whereas a true trace-based approach would collect information on every network injection. We still use probability-based transitions between microphases. The only potential downside of this approach is the increase in simulation time in situations where the probability-based approach quickly converges on its Markov steady state behavior. For this reason, we reduce the simulation time by reducing the number of executed microphases. To achieve this, for each macrophase, the minimum number of microphases that needs to be executed to reach steady state is calculated within an error margin (we use 2%). The finalized number of microphases per macrophase is determined by selecting the largest numbers calculated from the previous step. For example, in a scenario where macrophase and microphase resolutions are 5M and 250 (the ratio is 20000), if our scripts determine that macrophases 1 and 2 should execute at least 2000 and 4000 microphases to reach steady state, respectively, we can obtain a simulation speedup close to 5× by executing only 4000 microphases per macrophase, instead of 20000.

4.3 Spectral Analysis for Interval Selection

To automate the process of finding a good macrophase length, we make use of Fourier transform-based frequency-domain analysis. Our approach consists of five main steps. First, the application network trace is processed to generate the injection rate of initiating messages over time, as shown in Fig. 9a. Second, we convert this time-domain information to the frequency domain via a Fast Fourier Transform (FFT). Third, we remove the offset component, and then trim the second half of the FFT result (because the FFT always produces a symmetric/mirrored result). Fourth, as the generated FFT series consist of complex numbers, we calculate the absolute values of these numbers. Fig. 9b shows the absolute value of the first 2500 frequency components for CoMD. The final step is to calculate the period of injection rate using this information. In Fig. 9b, the component with the highest absolute value is the fourth component. This component represents a period of 450M cycles; this is in line with the period that can be observed in Fig. 9a. If this period is too large to be efficient for hierarchical clustering, it can be divided into smaller periods that still align with the periodicity of the overall trace. As shown in Section 3.2.3, Hotspot is another application that shows a periodic injection rate behavior. Our FFT approach is able to automatically determine the period to be 1120K cycles.



(a) The number of initiating messages in CoMD over time.

(b) FFT coefficients vs. frequency components.

Figure 9: Spectral analysis of CoMD.

4.4 Bursty Injection Model

Because SynFull has limitations in capturing bursty network behavior during GPU execution, we propose a new injection model that generates bursts similar to the real workload.

Capturing temporal bursts: SynFull uniformly injects messages into the network during each micro cluster. The amount of traffic is calculated based on a distribution of node injection rate, defined as the number of messages injected in a microphase. In addition to this distribution, we introduce two more distributions. The first one is a distribution of inter-arrival times between two consecutive bursts of initiating message injections. The second distribution collects the number of initiating messages generated in the same cycle. Note that the probability distribution functions of the inter-arrivals and bursts directly observed from the trace (i.e., not a mathematical function like an exponential). However, generating traffic based on these two independent distributions might result in unrealistic traffic patterns. Therefore, we collect the joint distribution of inter-arrival times and bursts. We generate a random inter-arrival time and burst pair from this joint distribution. First, based on the generated inter-arrival time, our injection model determines when a burst occurs. Then, based on the generated burst, the model determines how many initiating messages are injected at once. We continue generating initiating messages until the injection amount is reached, which is calculated by SynFull based on the injection rate distribution. This approach allows the number of generated messages per microphase to be similar to the real workload behavior, while capturing the bursty behaviors in the temporal dimension.

Capturing spatial bursts: To limit the number of injecting nodes and the source-destination pairs, for each micro

cluster and initiating message type, we collect the distribution of the number of unique nodes that inject traffic into the network in a microphase, and the number of unique source-destination pairs that observe traffic. Based on these distributions, the traffic generator generates a limit on the number of source nodes as well as the source-destination pairs for each microphase. Using these limits and the distributions that SynFull uses to generate source and destination nodes, APU-SynFull generates a set of source nodes and a set of source-destination pairs that can generate and receive traffic in that microphase. This mechanism allows the generated messages per microphase to have a spatially bursty behavior, similar to what is observed in real executions.

4.5 Trace Extrapolation

A key challenge to simulating large, heterogeneous SoCs is to take statistical traffic data from a smaller system to accurately evaluate a larger system. We first focus on strong scaling scenarios where an application's problem size is fixed and a larger system is used to find a solution faster. We will later revisit weak scaling where the problem size increases with the compute resources. To model larger systems, we introduce a novel trace extrapolation methodology. Trace extrapolation is the process of taking traces collected on smaller systems and projecting the traces to a parametrically different system. The goal is to ensure that the extrapolated trace is representative of the NoC communication behavior of the target system. Trace extrapolation faces the following challenges. First, parallel programs exhibit nondeterminism. A parallel program in general has a large number of possible execution paths, resulting in different communication patterns. Even if the execution path is deterministic when the same input set is given, depending on how computation is allocated and scheduled, on-chip traffic patterns vary for different systems. Second, runtime information is not available during trace extrapolation. For example, a cache miss generates a request or a directory miss causes a broadcast. However, neither cache miss nor directory miss information is available without a real simulation. Therefore, it is challenging to predict the timing and flow of on-chip communication in trace extrapolation.

In this subsection, we present a methodology to generate synthetic traces for a larger system by extrapolating communication behavior from application traces collected on a series of smaller systems. The proposed mechanism considers three behavior characterizations: 1) injection flow, which indicates where a message originates from and is destined to; 2) execution time, which indicates how long a kernel executes until completion; and 3) injection rate, which refers to the number of messages generated in an interval.

Injection flow extrapolation: Extrapolation of injection flow involves projecting the source and destination nodes for each injection individually. To achieve this goal, we must know the injection distribution of all source nodes, as well as the receiving distribution of all destination nodes. Such distributions inform the probability of which individual nodes inject/receive messages. Assuming injection and receiving processes are independent of each other, a series of injection flows can be constructed based on the probability. Both injection and receiving distributions can be obtained in the three steps described below.

Step 1: Group the network nodes into a source cluster and a destination cluster. For example, consider GPU L1 cache miss events: all GPU L1 caches are grouped into one source cluster, while all GPU L2 caches fall into a destination cluster. Other nodes such as CPU caches and directory nodes, are not considered for this particular event. Similarly, for GPU L2 cache miss events, only GPU L2 cache nodes and directory nodes are considered for grouping. In general, communication taking place between different levels of the memory hierarchy can be extrapolated separately, which simplifies the projection process.

Step 2: Calculate the distribution across all nodes within each cluster. Specifically, the injection probability of each node is calculated in the source cluster, and the receiving probability is calculated for nodes in the destination cluster. Fig 10a shows the injection distribution of GPU L1 caches in a 32-CU system (sorted by injection rate), with the x-axis being a collection of GPU L1 cache nodes and the y-axis being the injection probability. A linear regression is applied to the distribution curve (regression trend line shown). The statistics collected from this step are the slope and intercept of the fitted line.

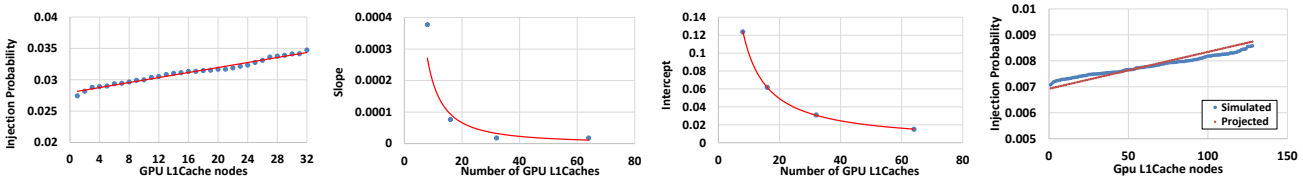
Step 3: Project distributions for the target system. Given a series of slope and intercept values for smaller systems, we apply another curve fitting to project the slope and intercept for the target system. Then a distribution can be constructed accordingly. Fig 10b and Fig 10c demonstrate the curve fitting process for slope and intercept. The projected distribution result is shown in Fig 10d. Four data points collected from 8-CU, 16-CU, 32-CU, and 64-CU systems are used to project the distribution of the target 128-CU system.

To form an injection flow, we probabilistically select a node from the source cluster and a node from the destination cluster following the projected distribution.

Execution time extrapolation: Given the amount of traffic injection, execution time impacts the overall network utilization, which in turn affects the network performance. The execution time of an application may or may not scale as the system size grows. The scaling factor is dependent on the parallelism of the application, as well as the hardware architecture. When extrapolating execution time, we assume all hardware resources scale proportionally (i.e., the compute unit to storage bandwidth ratio remains the same as the system scales). Our methodology is applicable to non-proportional scaling cases, but we have not yet evaluated these scenarios.

Similar to the approach described in Step 3 of the injection flow extrapolation, we project execution time for the target system by curve fitting. A single GPGPU application exhibits multiple phases; some of the phases correspond to CPU execution while some of them correspond to different kernels. We differentiate these phases and extrapolate the execution time for each phase individually. The purpose is to separate scalable phases from non-scalable ones. During extrapolation, different scaling factors are applied to different phases, resulting in reasonable accuracy for overall execution time extrapolation.

Injection rate extrapolation: The spatial distribution of on-chip communication is determined by injection flow extrapolation, while the injection rate extrapolation is respon-



(a) Injection distribution. (b) Slope curve fitting. (c) Intercept curve fitting. (d) Projection result.

Figure 10: Injection flow extrapolation process.

sible for deciding the temporal distribution of the traffic injection. To extrapolate the injection rate, we use a probabilistic approach in our model. Each CPU/GPU phase is further divided into 1000-cycle epochs. Injection rate for the entire phase is assumed to follow a Weibull distribution. However, within each epoch, traffic has a constant injection rate λ. Given the probabilistic (Weibull) distribution and the amount of packets injected for each phase, the traffic injection amount for each epoch is calculated individually. Here we apply the same curve fitting approach described above to extrapolate the parameters for the Weibull distribution. Compared to a deterministic approach where the injection rate is constant throughout the entire phase, this probabilistic approach is capable of capturing traffic burstiness.

Weak scaling: The strong scaling scenarios are somewhat more difficult because the amount of work per compute resource tends to decrease, which affects cache, network, and memory behaviors. With weak scaling, the work-per-compute resource remains constant, and so individual compute units in a larger system tend to continue to behave like the compute units from smaller systems. When extrapolating the injection rate in weak scaling, we make use of the same injection flow methodology described above to generate the proper distribution of sources and destinations for the larger system. However, for injection rates, we simply select a smaller-sized system as a baseline and preserve the same injection rate (per source) in the extrapolated system.

4.6 Memory

Using synthetic addresses, we want to capture memory characteristics during a SynFull phase without the need for a full address trace. Our address generation approach decomposes address traces into memory phases with a fixed number of requests and analyzes those phases to produce synthetic addresses with similar memory request behavior. We observe that utilizing a bank access distribution (i.e., number of accesses to each unique bank), request ordering, and row-buffer hit rate information on a per-phase basis can provide accurate modeling of synthetic access counts, row-buffer hit rates, and memory latencies. Exactly generating the same absolute addresses is not necessary so long as the result is that the final DRAM bank access distributions, row-buffer hit rates, and overall memory latencies are similar to what would otherwise be observed with a full cycle-level simulation. The general process of creating this synthetic model is shown in Fig. 11.

An input trace is first binned into bank and channel pairs based on, for example, a typical memory address decoder or bank/channel interleaving function. The number of accesses to each destination are counted to generate a bank access distribution. We also consider the ordering of addresses by generating a stochastic matrix representing the probability

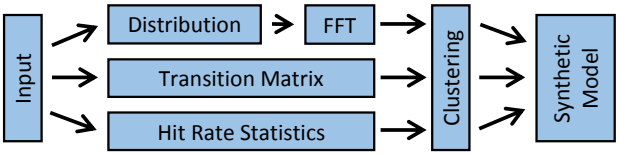


Figure 11: Overview of generation of synthetic model from address trace.

of transitioning from one bank to another. This aims to capture bank conflicts more accurately by capturing bursts of requests to the same bank or lack thereof. Using this transition probability, we generate "resequenced" bank accesses containing a number of requests to each bank corresponding to the bank access distribution. At this point we can generate an intermediate trace by combining the new sequence and hit rates together. The previous row address is used when a row-buffer hit should occur while a random row address is selected otherwise. Addresses are encoded by reversing the binning function to create a synthetic address from the row addresses and bank/channel pairs.

In order to reduce the phase count and provide the ability to extrapolate traces, clustering is used similar to original SynFull. To cluster phases together, we compare the similarity of bank access distributions, transition probabilities, and row-buffer hit rates after completing the steps above. For this, we return to a spectral analysis approach and use a clustering algorithm to group similar phases. Applying an FFT to the bank access distribution allows us to decompose aggregated memory requests into more simplistic periodic functions. From this point, we can choose a fixed number of frequencies for the periodic functions that reduce the error on the bank distribution to represent the phase. This allows for significant reduction in the number of dimensions when clustering phases and greatly simplifies the process.

5. EVALUATION

5.1 Simulation Methodology

We use an APU simulation platform consisting of gem5 [9] and a modified version of the GPU model [10] to collect traces for APU-SynFull model generation and to compare the accuracy of our models against the baseline full-cycle simulation. We use Garnet [11] to simulate the network. We use 2-stage routers; and each router input port has 4 virtual channels, with 8-flit deep buffers. Our baseline system is shown in Fig. 1. We replace the CUs and the CPU core with SynFull models to evaluate APU-SynFull and SynFull. Our memory model utilizes the built-in gem5 model [12] modified with HBM timings [13]. There are 8 memory channels with 8 banks in each channel. We ignore the impact of refresh to focus on reducing modeling error considering all other timing parameters. We execute the ap-

Application	Input size	Application	Input size
bitonic [14]	262144	comd [18]	16
dct [14]	2048	backprop [16, 17]	131072
histogram [14]	1024	bfs [16, 17]	65536
matrixmul [14]	512	hotspot [16, 17]	1024
spmv [15]	256	nw [16, 17]	2048 2048 10

Table 2: List of workloads.

plications listed in Table 2 from the AMD SDK [14], Open Dwarfs [15], Rodinia [16, 17] and Proxyapps [18] suites.

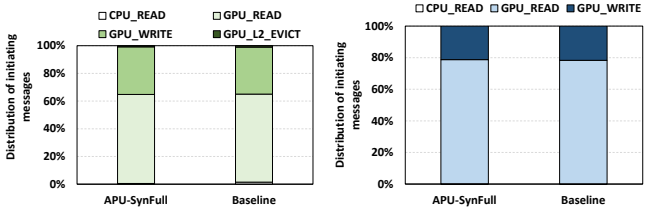
We generate the traffic models using a modified version of SynFull scripts. We use a microphase length of 250 for all applications. For non-periodic applications, we use a macrophase length of 5M in order to keep the ratio between these resolutions high. A higher ratio increases accuracy, and also provides more scope for increasing simulation speed (see Section 4.2). However, it also causes clustering to take very long, and becomes impractical. CoMD and Hotspot use the automated macrophase length generation methodology (Section 4.3) and use macrophase lengths of 7.03M and 560K, respectively.

We report three metrics to evaluate accuracy. We use *average network latency* for an overview of the traffic behavior. We also compare the latency and initiating message type distributions that APU-SynFull yields with those of the baseline. Comparing the latency distributions is useful to determine if congestion is accurately modelled by APU-SynFull. We use the Hellinger Distance defined in Equation 1 to calculate the similarity between two distributions, where P and Q are two discrete distributions (in our case, packet latency distributions or the initiating message type distributions), and p_i and q_i are the i^{th} element of P and Q, respectively.

$$H(P,Q) = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^{k} (\sqrt{p_i} - \sqrt{q_i})^2} \quad (1)$$

5.2 APU-SynFull Results

In this section, we evaluate our APU-SynFull methodology. Fig. 12a shows the ratio between the number of injected messages for four different initiating message types for matrixmul, with APU-SynFull and the baseline. Fig. 12b shows the same for hotspot. APU-SynFull is successful in generating the initiating messages with a distribution close to that of the baseline. The overall accuracy of initiating message distributions for all applications are given in Fig. 14b.



(a) The ratio of different initiating messages in matrixmul.

(b) The ratio of different initiating messages in hotspot.

Figure 12: Accuracy of generated initiating messages.

Next, we describe the simulation accuracy of each benchmark in detail. Fig. 13 shows the error in average network latency with SynFull and APU-SynFull. Fig. 14a shows the Hellinger distance for SynFull and APU-SynFull latency distributions compared to the baseline. Fig. 14b shows the

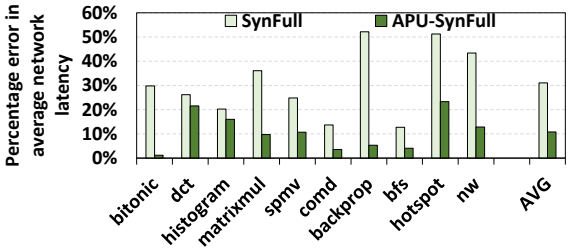
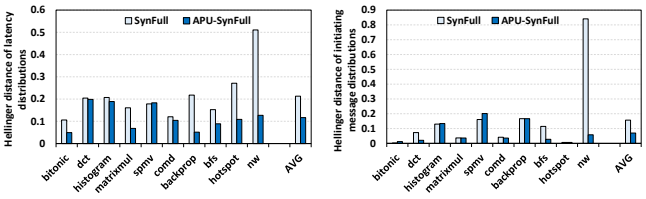


Figure 13: Percentage error in average network latency with respect to the baseline.



(a) Latency distributions.

(b) Initiating message distributions.

Figure 14: Hellinger distance for SynFull and APU-SynFull compared to the baseline.

Hellinger distance for SynFull and APU-SynFull initiating message distributions compared to the baseline. The lower the distance, the more similar the distributions are.

• **bitonic:** bitonic is a very bursty application (it is the application with the highest network latency in our suite), and SynFull fails to generate traffic with high enough latency, mainly due to its inability to capture the spatial bursts. The average latency with APU-SynFull is very close to the baseline. Moreover, the latency and the initiating message distributions are close to the baseline as well.

• **dct:** Although the Hellinger distance of initiating message distribution is low, the total number of generated GPU read and write requests is lower compared to the baseline. This is mainly due to clustering, and thus results in low latency distribution accuracy for both SynFull and APU-SynFull. However, APU-SynFull performs slightly better than SynFull due to its better injection model.

• **histogram:** This is a short running application with a very short GPU phase. Due to this, clustering is not very effective in representing all phases of the application, leading to inaccurate initiating message distribution. This causes both SynFull and APU-SynFull to have inaccurate latency distributions. SynFull also yields lower latency than the baseline, mainly due to the inaccurate number of simulated CPU and GPU phases.

• **matrixmul:** SynFull yields low average latency due to its uniform injection model. APU-SynFull captures the bursty behavior for the GPU requests, both temporally and spatially. However, in this application, many cache lines are requested by multiple GPU cores simultaneously. Once such a requested cache line reaches the GPU L2 cache, it is replicated and sent to its requesters, causing a burst in the GPU reply network. APU-SynFull is currently unable to emulate this behavior, resulting in slightly lower latency for the GPU reply network than that of the baseline. Latency and message distributions are reasonably accurate.

• **spmv:** This application, similar to histogram, is short running with a very short GPU phase, causing clustering to be

ineffective in representing the whole execution. The initiating message and latency distributions are not accurate. The significant improvement in average latency with APU-SynFull over SynFull is attributed to its injection model.

• **CoMD:** This application uses our automated macrophase generation methodology. SynFull suffers from very inaccurate initiating message distribution because of its probability-based transition model. Using the macrophase replay method provides much better message distribution and better latency, and APU-SynFull provides significantly more accurate latency results.

• **backprop:** This is a bursty application, and SynFull fails to generate representative network traffic. APU-SynFull provides very accurate latency distribution and average latency. Its initiating message distribution is less accurate due to CPU requests. Because CPU requests have much lower latency impact compared to GPU requests, this inaccuracy in initiating message distribution does not impact overall latency significantly.

• **BFS:** APU-SynFull is accurate in average network latency, latency distribution, and the initiating message distribution, and is more accurate than SynFull due to both its ability to capture temporal bursts and the macrophase replay model.

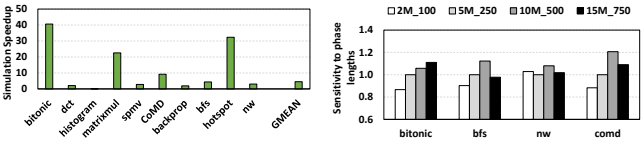
• **hotspot:** This application uses our automated macrophase generation methodology, and provides good initiating message distribution. It demonstrates high spatial bursts, and SynFull is unable to capture it, causing it to be very inaccurate. APU-SynFull, due to its improved injection model, provides better latency.

• **nw:** Due to the problem described in Section 3.2.2, the probability-based transition model in SynFull fails to generate a GPU phase, and thus generates very low network traffic. Although this is a low-traffic application, SynFull latency is still far from that of the baseline. Overall, APU-SynFull provides reasonable accuracy in terms of initiating message and latency distributions.

These results demonstrate that the traffic generated by APU-SynFull resembles the traffic of real APU workloads. Moreover, average network latency is within 11% of the system emulation, outperforming SynFull in terms of accuracy.

Fig. 15a shows the simulation speedup obtained by APU-SynFull. Applications such as bitonic, matrixmul, CoMD and hotspot obtain significant speedups. Short running applications such as histogram, spmv, and backprop do not benefit significantly in terms of simulation speedup. In dct, we determine that the ratio of macrophase length to microphase length should be high. This does not allow discarding the execution of some microphases, limiting simulation speedup. Overall, APU-SynFull reduces the simulation time of ten applications by 4.5×, on average (geometric mean).

Fig. 15b shows the sensitivity of APU-SynFull to the choice of macrophase and microphase lengths, using four representative applications. We change the macrophase and microphase lengths, while keeping the ratio between them constant. We do so to ensure that all applications execute enough microphases in a macrophase to reach the steady-state condition. The legend shows the choice of macrophase and microphase lengths, respectively. Although using the joint distribution of inter-arrival times and injection bursts reduces the sensitivity to phase lengths, augmenting the injection model with the ability to capture spatial bursts introduces a slight dependency to the choice of phase lengths. The maximum observed discrepancy is 20%.



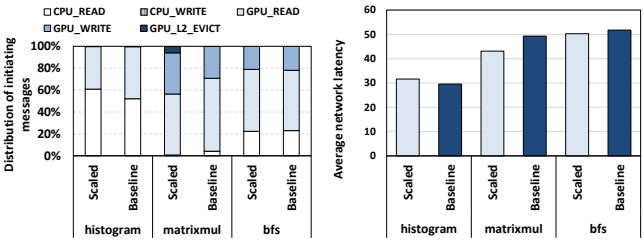
(a) Simulation speedup with APU-SynFull.

(b) Sensitivity of average network latency to macrophase and microphase lengths.

Figure 15: Simulation speedup and phase length sensitivity.

5.3 Scaling Results

In this section, we validate our scaling methodology with a 128-core system in the context of strong scaling. We also present a use case of NoC design space exploration with weak scaling. Three representative applications are considered. Histogram is a short running application with a light-weight kernel, where CPU and GPU traffic injection volumes are comparable. Matrixmul, a highly parallel application, whose performance scales well as core count grows. BFS which is an application with bursty traffic injection during kernel execution.



(a) Initiating message distribution.

(b) Average network latency.

Figure 16: Accuracy of strong scaling in 128-CU system.

For strong scaling validation, an extrapolated trace for a 128-CU system is generated based on our trace extrapolation methodology. Then we generate a model file and use it for APU-SynFull simulation (referred to as Scaled in this evaluation). In comparison, we launch a full-cycle simulation with the same configuration (referred to as Baseline). Fig. 16a shows the initiating message distribution comparison. In histogram, the scaled system generates more CPU traffic than GPU traffic, which is also observed in the baseline. In strong scaling, the amount of on-chip storage (i.e., L1 caches and L2 caches) increases as system size grows. Therefore, a lower cache miss rate is expected in a larger system for matrixmul. However, our trace extrapolation methodology currently does not account for runtime information such as cache miss rates, thus it ends up generating more requests than necessary. The cause of inaccuracy in ratio is multifaceted: the algorithm, cache thrashing, address mapping, and use of local memory can all simultaneously impact the generation of read/write requests. BFS has an injection spike after kernel launch, but the scaling factor of initiating messages is relatively constant compared to the other two applications. The proposed scaling methodology is able to project the traffic injections accurately. Average network latency is compared in Fig. 16b. Packet latency is

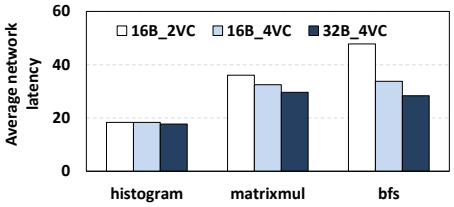


Figure 17: NoC exploration with weak scaling in 128-CU system.

related to network congestion. While network congestion varies during runtime depending on both application phases and on-chip hardware resources, our methodology is able to reasonably mimic the congestion behavior. We believe better accuracy can be achieved if detailed runtime information is provided for trace extrapolation.

It is hard to validate weak scaling accuracy because finding an application whose computation and storage complexity scales by the same factor is difficult. In addition, current applications may not scale well to large system sizes. Our weak scaling approach provides a synthetic evaluation environment that allows one to project how applications might behave on future systems in the absence of applications that are well-tuned for such large SoCs. Instead, we present how weak scaling can be used for NoC design space exploration in Fig. 17. In this experiment, we vary the channel width (16-Byte/32-Byte) as well as the number of virtual channels (2-VC/4-VC) using model files generated from extrapolated weak scaling traces. Virtual channels impact the network congestion while channel width affects the number of flits in a data packet. From the results, we can tell that both matrixmul and BFS are sensitive to NoC bandwidth variation. By further sweeping through the network parameters, we will be able to find out a combination of parameters that satisfies the design requirement. In summary, weak scaling enables NoC design exploration for future applications.

5.4 Memory Results

Our memory results focus on reduction of average read, write, and hit rate error over a fixed latency model. The baseline for our results is the memory latency and row-buffer hit rate from gem5's built-in memory model when running in cycle-level system emulation mode. Latency error and standard deviation across all applications is shown in Fig. 18. Overall, the error in memory latency is reduced by about 37% compared to a fixed-latency model, which provides a substantial improvement, although we readily admit that there is more that can be done. The observed row-buffer hit rates of our synthetic address models were also within 12% of the cycle-level model. Sources of remaining error include read/write interleaving in each phase, FFT component selection, and transition matrix clustering.

Interleaving of writes with reads typically resulted in the largest latency error. Write times and addresses are highly dependent on higher-level memory replacement policy, special requests such as GPU read-modify-writes, and memory controller optimizations such as write buffering. For example, our memory system models separate read/write queues, the rate at which the write queue fills will impact when the queue is drained. In BFS this resulted in up to 10× increased latency error during heavy write phases. By modeling a simple, single queue controller, latency error in BFS is reduce to at most 50%.

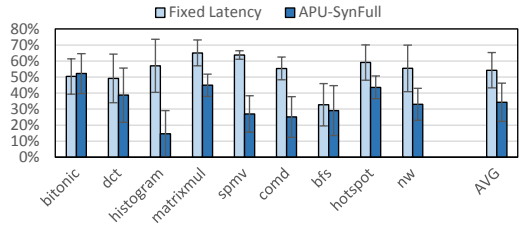


Figure 18: Memory latency error and deviation comparing fixed-latency SynFull and APU-SynFull model to the baseline.

In address generation, using a subset of FFT components described in Section 4.6 is inherently lossy and results in differences between input bank distribution and synthetic model bank distribution. Clustering phases together based on similar FFT components produces additional errors, as the cluster center may be averaged to a different subset of frequency components. However, across the entire application run, we observed negligible difference in latency error using all 64 FFT components compared to as little as 5-10 components when selected by largest magnitude.

Similar results were observed for the transition matrix, which was introduced to model bank conflicts by ordering synthetic addresses more similar to the input ordering compared to random sampling. A small amount of information can be used to replay addresses without much variation in accuracy. For example, average transition probability can be used to determine if a distribution of addresses are sequential or random. We found in most cases the transition matrix does not provide enough benefit to justify increasing the complexity of clustering.

The introduction of address generation provides the ability to utilize existing memory models within the APU-SynFull framework. These results show that synthetic approaches are promising and fixed-latency memory models should not be used.

6. RELATED WORK

Modeling and simulation. As discussed in Section 2, the most closely related work to ours is SynFull [7]. gem5-gpu [19] combines gem5 [9] and GPGPU-Sim [20] to model a flexible and cache-coherent [21] APU-like system. Macsim [22] simulates the network traffic of multi-programmed CPU and GPU workloads, but does not model a unified memory address space and cache coherence. Multi2Sim [23] models both CPU and GPU cores but with distinct memory hierarchies. Several network-on-chip simulators [24, 25, 26] provide timing and power analysis of NoCs. Recent works that use DSLs provide both hardware and software model generator tools for NoCs [27], and CMPs [28]. Our work is the only work that provides a fast simulation methodology for cache-coherent APUs, and enables scalability studies for larger future systems. A broad range of memory models have developed in the past for use in simulation. Most work focuses on detailed cycle-level models rather than fixed-latency of queuing model studies. DRAMSim2 [29] and USIMM [30] are two popular simulators. DrSim [31] extends DRAMSim2 to provide flexibility. Later simulators such as gem5's model [12], NVMain [32],

and Ramulator [33] aim to provide flexibility and extendibility for newer memories such as NVM and die-stacked memory, and standards such as HBM [13] and HMC.

Synthetic and statistical models. Prior works [34, 35] investigate generating synthetic workloads that represent application behavior. Wunderlich et al. [36] propose methodologies that use statistical sampling to increase simulation speed. Several works [37, 38] use synthetic traffic models in the context of NoC simulation methodologies. Eeckhout et al. [39] use statistical methods to generate statistically correct synthetic benchmark traces. Bighouse simulator [40] uses stochastic modeling to simulate power, performance and reliability of data centers. To our knowledge, our work is the only statistical method that generates synthetic application traffic for APUs. Workload cloning allows synthetic workloads to be generated and released from proprietary ones [41, 42, 43]. Specifically, work focusing on replicating cache behavior in workload clones has been explored [42]; they analyze the cache statistics needed to accurately capture memory access behavior. In our work, we take a similar approach in identifying memory access characteristics that are needed to synthetically generate a memory address stream. Spatio-temporal memory cloning (STM) [43] is a methodology to accurately generate clones of memory access patterns. STM focuses on caches and TLB behavior using a variety of stride patterns while we capture memory behavior such as bank conflicts and row buffer hit rates.

7. CONCLUSIONS

In this work, we propose novel extensions to the SynFull methodology to tackle challenges of large scale heterogeneous computing systems. With this new methodology, computer architecture researchers can now explore NoC and memory designs at scale without being bogged down by otherwise slow or unscalable simulations. Overall, APU-SynFull is a robust evaluation methodology targeting NoCs and memory systems for future large scale heterogeneous SoCs that will be of great value to the architecture community looking forward.

Acknowledgment

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

8. REFERENCES

[1] G. Krishnan *et al.*, "Energy Efficient Graphics and Multi-media in 28nm Carrizo APU," in *HotChips*, 2015.

[2] P. Hammarlund *et al.*, "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, pp. 6–20, March–April 2014.

[3] D. Kanter, "Graphics Processing Requirements for Enabling Immersive VR," *AMD White Paper*, July 2015.

[4] C. Byun *et al.*, "Driving Big Data With Big Compute," in *HPEC*, 2012.

[5] M. J. Schulte *et al.*, "Achieving Exascale Capabilities through Heterogeneous Computing," *IEEE Micro*, pp. 26–36, July–August 2015.

[6] O. Villa *et al.*, "Scaling the Power Wall: A Path to Exascale," in *SC*, 2014.

[7] M. Badr and N. Enright Jerger, "SynFull: Synthetic Traffic Models Capturing a Full Range of Cache Coherence Behaviour," in *ISCA*, 2014.

[8] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," *AMD*, August 2012.

[9] N. Binkert *et al.*, "gem5: A Multiple-ISA Full System Simulator with Detailed Memory Model," vol. 39, June 2011.

[10] AMD Research, "The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5," in *gem5 User Workshop*, 2015.

[11] N. Agarwal *et al.*, "GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator," in *ISPASS*, 2009.

[12] A. Hansson *et al.*, "Simulating DRAM Controllers for Future System Architecture Exploration," in *ISPASS*, 2014.

[13] JEDEC, "High Bandwidth Memory (HBM) DRAM," http://www.jedec.org/standards-documents/docs/jesd235.

[14] AMD Inc., "AMD SDK," http://developer.amd.com/tools-and-sdks.

[15] K. Krommydas *et al.*, "On the Characterization of OpenCL Dwarfs on Fixed and Reconfigurable Platforms," in *ASAP*, 2014.

[16] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[17] ——, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *IISWC*, 2010.

[18] J. Mohd-Yusof, "CoMD Proxy Application," http://www.exmatex.org/comd.html.

[19] J. Power *et al.*, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," vol. 13, no. 1, Jan. 2014.

[20] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[21] J. Power *et al.*, "Heterogeneous System Coherence for Integrated CPU-GPU systems," in *MICRO*, 2013.

[22] HPArch, "MacSim Simulator," http://code.google.com/p/macsim/.

[23] R. Ubal *et al.*, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *PACT*, 2012.

[24] A. B. Kahng *et al.*, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration," in *DATE*, 2009.

[25] C. Sun *et al.*, "DSENT - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *NoCS*, 2012.

[26] N. Jiang *et al.*, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *ISPASS*, 2013.

[27] F. Fatollahi-Fard *et al.*, "OpenSoC Fabric: On-Chip Network Generator: Using Chisel to Generate a Parameterizable On-Chip Interconnect Fabric," in *NoCArc*, 2014.

[28] D. Lockhart *et al.*, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," in *MICRO*, 2014.

[29] P. Rosenfeld *et al.*, "DRAMSim2: A Cycle Accurate Memory System Simulator," vol. 10, no. 1, pp. 16–19, 2011.

[30] N. Chatterjee *et al.*, "USIMM: the Utah SImulated Memory Module," University of Utah, TR UUCS-12-002, 2012.

[31] M. K. Jeong *et al.*, "DrSim: A Platform for Flexible DRAM System Research," "http://lph.ece.utexas.edu/public/DrSim", 2012.

[32] M. Poremba *et al.*, "NVMain 2.0: Architectural Simulator to Model (Non-)Volatile Memory Systems," *CAL*, vol. PP, no. 99, pp. 1–1, 2015.

[33] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," vol. PP, no. 99, pp. 1–1, 2015.

[34] K. Sreenivasan and A. Kleinman, "On the Construction of a Representative Synthetic Workload," *Communications of the ACM*, vol. 17, no. 3, pp. 127–133, 1974.

[35] D. Ferrari, "On the Foundations of Artificial Workload Design," in *SIGMETRICS*, 1984.

[36] R. E. Wunderlich *et al.*, "SMARTS: Accelerating Microarchitecture Simulation Via Rigorous Statistical Sampling," in *ISCA*, 2003.

[37] V. Soteriou *et al.*, "A Statistical Traffic Model for On-chip Interconnection Networks," in *MASCOTS*, 2006.

[38] L. Tedesco *et al.*, "Application Driven Traffic Modeling for NoCs," in *SBCCI*, 2006.

[39] L. Eeckhout *et al.*, "Performance Analysis Through Synthetic Trace Generation," in *ISPASS*, 2000.

[40] D. Meisner *et al.*, "BigHouse: A Simulation Infrastructure for Data Center Systems," in *ISPASS*, 2012.

[41] A. Joshi *et al.*, "Cloning: A Technique for Disseminating Proprietary Applications at Benchmarks," in *IISWC*, 2006.

[42] G. Balakrishnan and Y. Solihin, "WEST: Cloning Data Cache Behavior Using Stochastic Traces," in *HPCA*, 2012.

[43] A. Awad and Y. Solihin, "STM: Cloning the Spatial and Temporal Memory Access Behavior," in *HPCA*, 2014.