SynFull: Synthetic Traffic Models Capturing Cache Coherent Behaviour

Mario Badr   Natalie Enright Jerger
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
mario.badr@mail.utoronto.ca, enright@ece.utoronto.ca

Abstract

Modern and future many-core systems represent complex architectures. The communication fabrics of these large systems heavily influence their performance and power consumption. Current simulation methodologies for evaluating networks-on-chip (NoCs) are not keeping pace with the increased complexity of our systems; architects often want to explore many different design knobs quickly. Methodologies that capture workload trends with faster simulation times are highly beneficial at early stages of architectural exploration. We propose SynFull, a synthetic traffic generation methodology that captures both application and cache coherence behaviour to rapidly evaluate NoCs. SynFull allows designers to quickly indulge in detailed performance simulations without the cost of long-running full-system simulation. By capturing a full range of application and coherence behaviour, architects can avoid the over or underdesign of the network as may occur when using traditional synthetic traffic patterns such as uniform random. SynFull has errors as low as 0.3% and provides 50× speedup on average over full-system simulation.

1. Introduction

With the shift to multi- and many-core processors, architects now face a larger design space and more complex trade-offs in processor design. The design of the network as a potential power and performance bottleneck is becoming a critical concern. In the power-constrained many-core landscape, NoCs must be carefully designed to meet communication bandwidth requirements, deliver packets with low latency, and fit within tight power envelopes that are shared across cores, caches and interconnects. To do this well, the designer must understand the traffic patterns and temporal behaviour of applications the NoC must support. There are a large number of parameters in the NoC design space that can be tuned to deliver the required performance within a given cost/power envelope, such as topology, routing algorithm, flow control and router microarchitecture. These knobs are most commonly explored through software simulation.

There are a number of simulation methodologies available to NoC designers, however each comes with speed/fidelity tradeoffs [18]. Full-system simulators model each hardware component of the overall system and can run full applications and operating systems. As a result, these simulators provide the highest degree of accuracy, but at the expense of long simulation times. In contrast, a designer can use traditional

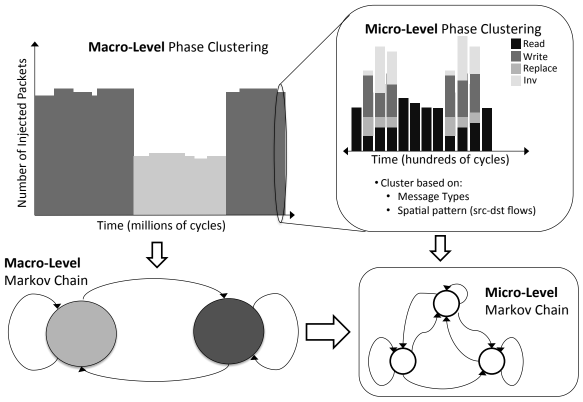

Macro-Level Phase Clustering
Number of Injected Packets
Time (millions of cycles)
Micro-Level Phase Clustering
Read
Write
Replace
Inv
Time (hundreds of cycles)
• Cluster based on:
• Message Types
• Spatial pattern (src-dst flows)
Macro-Level Markov Chain
Micro-Level Markov Chain

Figure 1: High level view of SynFull

synthetic traffic patterns to quickly stress their NoC design and reveal bottlenecks. However, these traffic patterns do not realistically represent the application space so the results are unlikely to be representative of real workloads. Therefore, they are unlikely to produce a properly provisioned network. In this work, we introduce a new approach that strikes a balance between these tradeoffs, providing a fast, realistic simulation methodology for NoC designers.

Realistic traffic patterns will increase the accuracy of NoC simulations. Beyond that, realistic traffic also provides optimization opportunities that do not exist in traditional synthetic traffic patterns. Many recent NoC proposals have exploited particular application [13, 30] or coherence behaviour [25, 26] to provide a more efficient, higher-performing NoC design. As research continues to push the scalability of cache coherence protocols [15, 28, 50], shared memory CMPs continue to be widespread. As a result, we focus on this class of systems.

SynFull Overview and Contributions. SynFull provides a novel technique for modelling real application traffic without the need for expensive, detailed simulation of all levels of the system. We abstract away cores and caches to focus on the network, and provide application-level insight to NoC designers, who in turn can produce more optimized designs. Through our analysis, we determine the key traffic attributes that a cache-coherent application-driven traffic model must capture including coherence-based message dependences (Sec. 4), application phase behaviour (Sec. 5) and injection process (Sec. 6). Fig. 1 shows a high-level overview of our approach. We observe long running (macro-)phases within applications as well as fine-grained variation within macro-phases (micro-phases), and group them through clustering. Within these clusters, we

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

examine the break down of message types dictated by the coherence protocol. These two steps drive a hierarchical Markov Chain that is used to reproduce the traffic behaviour. Our proposed model is independent of the network configuration and can be applied to a wide range of NoC configurations to enable rapid, accurate design space exploration.

To demonstrate the accuracy and utility of our model, we apply our methodology to a variety of PARSEC [5] and SPLASH-2 [48] benchmarks. A single full-system simulation run of each benchmark is required to create the model. We then use our models to synthetically generate traffic and compare NoC performance to full-system simulation. Finally, we demonstrate significant speedup for our methodology over full-system simulation; this allows for rapid NoC design space exploration. In essence, SynFull strives to replace full system simulation for *fast*, yet *accurate* NoC evaluation through richer synthetic traffic patterns.

2. The Case for Coherence Traffic

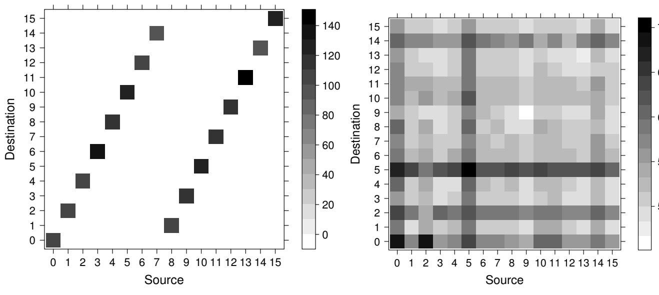
Before describing SynFull in detail, we motivate the need for a new class of synthetic traffic patterns. Traffic patterns such as uniform random, permutation, tornado, etc. are widely used in NoC research. Many of these are based on the communication pattern of specific applications. For example, transpose traffic is based on a matrix transpose application, and the shuffle permutation is derived from Fast-Fourier Transforms (FFTs) [2, 12]. However, these synthetic traffic patterns are not representative of the wide range of applications that run on current and future CMPs. Even if these traffic patterns were representative, the configuration of a cache-coherent system can mask or destroy the inherent communication pattern of the original algorithm due to indirections and control messages.

The arrangement of cores, caches, directories, and memory controllers directly influences the flow of communication for an application. Compare a synthetic shuffle pattern with the FFT benchmark from SPLASH-2 [48]. The shuffle pattern is a bit permutation where the destination bits are calculated via the function $d_i = s_{i-1}\ mod\ b$ where b is the number of bits required to represent the nodes of the network [12]. FFT is run in full-system simulation[1] while shuffle is run in network-only simulation. Fig. 2 shows the number of packets sent from a source to a destination[2]. In Fig. 2b, we see notable destination hot spots at nodes 0, 2, and 5 and source hot spots at nodes 0 and 5. However, Fig. 2a shows hot spots only for specific source-destination pairs.

The best NoC design for the traffic in Fig. 2a is unlikely to be the best NoC for the traffic in Fig. 2b. For example, we can design a ring network for Fig. 2a, and map the nodes to minimize hop count of shuffle on the network. The average injection rate of FFT is used for shuffle. Doing so yields $\sim$10% improvement in average packet latency over a mesh

[1]Configuration details can be found in Sec. 7.

[2]The absolute number of packets in each figure is unimportant in this comparison as we focus on source-destination traffic pairs.



(a) Shuffle Traffic Pattern (b) FFT Application

Figure 2: Spatial behaviour for synthetic vs application traffic

(Network A in Sec. 7) with the naive mapping (baseline) in Fig. 2a. However, using the same ring network in a full-system simulation of the FFT benchmark results in an average packet latency that is over three times worse than the baseline. Clearly, synthetic traffic patterns are not representative of the spatial behaviour exhibited by applications on a shared memory architecture.

The sharp contrast in Fig. 2 is due to coherence transactions needing to visit several nodes in a shared memory architecture before completing. For example, a write request first visits a directory to receive ownership of a cache line. The directory forwards requests to the core caching the data, and also invalidates caches who are sharing the data. Invalidated caches must send acknowledgements – this domino effect can significantly change an application's spatial behaviour and should be correctly modelled for realistic traffic generation.

Differentiating between the types of packets visiting nodes is important when generating realistic traffic. Most synthetic workloads split traffic into two categories: small control packets (requests) and large data packets (responses). However, there are many different packet types in a coherence protocol for both requests and responses. By lumping these packets into two categories, designers cannot explore methods that exploit cache coherence for better performance. For example, techniques exist to reduce traffic caused by acknowledgement packets [27]. Similar research insight is only possible when detailed packet information is available in simulation.

Finally, the traffic imposed by an application is time-varying. Applications exhibit phase behaviour [38]; spatial patterns are likely to change over time. Static traffic patterns and injection rates are not an adequate representation of real application traffic. The behaviour of cache coherence traffic changes with time and can have varying effects on NoC performance. For example, phases that exhibit high data exchange will likely result in several invalidation packets being broadcast into the NoC. It is important to capture these variations in traffic to reveal whether or not an NoC has been correctly provisioned.

3. SynFull Traffic Modelling Overview

Our methodology focuses only on the design of the NoC which has become a first-class component of many-core architectures. Thus, we abstract away the cores, caches, directories and memory controllers. Essentially, the performance characteristics of these elements are fixed for the purposes of our study. However, SynFull can be combined with analytical and abstract models [10, 22] of these components to explore an even richer design space with fast-turnaround time. Developing the network models is a critical first step; combining our model with other models is left as future work. To model application traffic, we focus on answering four key questions:

When to send a packet? In shared memory systems, packets are injected from the application side on a cache miss. This packet *initiates* a coherence transaction to retrieve its data. However, some packets are injected *reactively*. For example, a data packet would only be sent in response to a request.

Who is sending the packet? Not all nodes inject traffic uniformly so we must determine which node should inject that packet. For reactive packets, the answer is clear; the node reacting to the request is the source. However, for initiating packets, a model is required.

Why are they sending the packet? Traditional synthetic workloads do not concern themselves with why. For a cache coherence traffic generator, the question is very important. The *why* helps determine the type of packet being sent, and allows us to classify packets according to the coherence protocol.

Where is the packet going? The packet's destination is a function of both its source and the type of packet being injected (the answers to the previous two questions). Each source node may exhibit different sharing patterns with other nodes, and those sharing patterns may be different depending on the coherence message being sent.

These 4 questions are answered in Sec. 4. However, because applications exhibit phase behaviour [38], we must also capture how the answers *change over time*. We handle this by dividing application traffic into time intervals, and grouping together time intervals that behave similarly. Then, we determine answers for the When, Who, Why and Where questions for each group (phase). We discuss our methodology for grouping intervals in Sec. 5. To complete our SynFull methodology we need a way to transition between phases. For this we use a Markov Chain, where we can determine the probability of transitioning from one phase to another based on the phase we are currently in. The Markov Chain model, along with answers to the above 4 questions, allow us to recreate the injection process associated with an application (Sec. 6).

4. Modelling Cache Coherence Traffic

Focusing on the network only and not modelling application behaviour at the instruction level has the benefit of keeping our methodology generic and simple – we can apply SynFull to any application's traffic data in a straightforward manner. Although we abstract away other system components, not all network messages are equal so it is important to capture different message types injected by the coherence protocol. Message types are a function of the cache coherence protocol, but most protocols are conceptually similar in how they behave. A cache

Table 1: 1-to-1 Request-Response mappings. $ signifies cache.

Message Received	Source	Reaction	Destination
Cache Replacement	Cache	Writeback Ack.	Original Requestor ($)
Forwarded Request	Directory	Data	Original Requestor ($)
Invalidation	Directory	Ack.	Original Requestor ($)
Data	Cache	Unblock	Directory

miss invokes a coherence transaction from the local coherence controller in the form of a read or write which then results in a series of requests and responses [40]. In this section, we explore modelling packets that *initiate* a coherence transaction separately from packets that *react* to received messages.

4.1. Initiating Packets

To model *when* to send initiating messages, we collect the number of packets (P) injected into the network for a given interval spanning C cycles. Then, when generating synthetic traffic, we simply inject P packets uniformly over C cycles[3].

To answer *who* injects a packet, we observe the distribution of packets injected across all network nodes. This distribution gives us the probability a particular node will inject a packet and can capture spatial behaviour of applications [41, 44]. The answer to *where* a packet is going can be modelled using a similar method with relative probabilities. Given the source (S) of the packet, we determine its destination (D) using:

$$P(D \mid S) = \frac{Number\ of\ packets\ sent\ to\ D\ from\ S}{Number\ of\ packets\ sent\ by\ S} \tag{1}$$

Finally, to answer *why* a packet is injected we split P into P_r (total number of reads) and P_w (total number of writes). The distinction between reads and writes is necessary because they result in different reactions – writes lead to invalidations that are broadcast into the NoC; these can significantly impact NoC performance.

4.2. Reactive Packets

Most responses that maintain cache coherence have a simple one-to-one mapping with requests, such as an acknowledgement responding to an invalidation request. Upon receiving a particular message, the protocol *reacts* with a predetermined response. Table 1 shows a simplified view of the reactive aspect of cache coherence. Most reactions are straightforward but some requests lead to multiple different responses, particularly:

Forwarded Requests: If the data is already cached on chip, the coherence protocol forwards the request to the cache containing the data. Otherwise, the request goes off chip to memory.

Invalidates: When a write request arrives for a cache block shared by multiple readers, those readers must be invalidated. Next, we explore these two situations and how to model them so that we may realistically generate cache coherence traffic.

4.2.1. Forwarding vs. Off-Chip When a read or write request arrives at a directory, the requested block may be present in another core's cache. In this case, the request is forwarded to

[3]We also explored injecting packets using bernoulli and exponential distributions. However, the differences in performance are negligible.

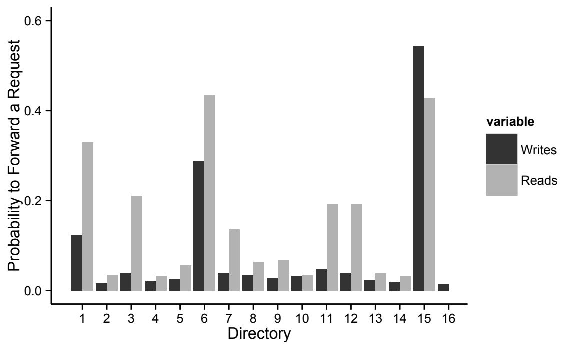


Figure 3: The probability a read or write request is forwarded

the cache holding the data. Otherwise, an off-chip memory request occurs. Fig. 3 shows the fraction of forwarded read and write requests broken down by directory for SPLASH-2's FFT benchmark[4]. The probability of forwarding a read or write changes according to which directory is being requested. Therefore, we model the distribution of forwarding probabilities on a per-directory basis. In Sec. 4.2.2, we show that this has an affect on invalidations, and different directories may act as hot spots in certain applications. We also note that the probabilities of forwarding a read or a write request are not equal. This distinction is critical as write requests will trigger invalidations to sharers which can represent a substantial burst of network traffic for widely-shared data.

4.2.2. Invalidates On a write miss, there is a chance that the cache block being requested has multiple sharers; the number of sharers determines the number of invalidates that will be multicast into the NoC. Fig. 4 shows the per-directory probability of sending 0 to 15 invalidates in a 16-node network for FFT. Some directories (1, 3, 11, and 12) exhibit bimodal behaviour; they invalidate 0 or $n-1$ sharers. Referring back to Fig. 3, we can see that these directories behave similarly in their forwarding probabilities. Other directories resemble an exponential distribution, with 0 invalidates having the highest probability. Invalidates can significantly impact network performance; applications that share and exchange data at a high rate will flood the network with many invalidates and strain its resources. We model the distribution of the number of invalidates on a per-directory basis to ensure our synthetically generated traffic has similar affects on NoC performance.

4.3. Summary

This section showed how we model cache coherence traffic by reacting to messages injected into the NoC. Read and write requests are forwarded with some probability to other nodes in the NoC, and invalidates can be sent out with some probability given the directory a write request has arrived at. To react to messages, read and write requests must first be injected into the NoC. Static injection rates are not sufficient to achieve high accuracy – we must also consider application phase behaviour.

[4]Our system configuration assumes 1 slice of the directory is located at each tile in a 16-core CMP. Addresses are interleaved across directories.

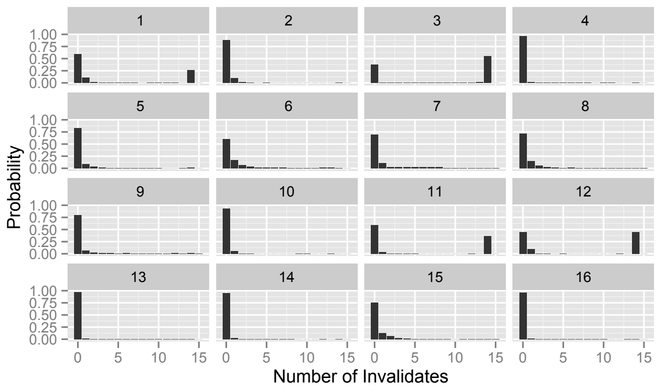


Figure 4: Number of sharers per write at different directories

We explore phase behaviour in Sec. 5 and propose a model that captures and applies phases to generated network traffic.

5. Traffic Phases

Applications are well-known to exhibit phase behaviour [38]. Phases can have a significant impact on the instructions per cycle, miss rates, and prediction rates of various microarchitectures. NoC traffic is also affected by application phases [20, 51]; our methodology needs to capture this phase behaviour if it intends to realistically generate synthetic traffic.

We propose examining traffic at two granularities: macro (millions or billions of cycles) and micro (thousands to hundreds of thousands of cycles). At the macro level, we observe noticeable differences in the behaviour of an application as it moves from one phase to another (perhaps due to a barrier or the end of an outer-loop). At the micro-level we are more likely to capture short bursts of network activity. Each level is divided into fixed-sized, successive time intervals measured in cycles.

Dividing traffic into intervals allows us to analyze network traffic at a fine granularity. Considering the entire application at once captures average behaviour; reproducing the average behaviour will negatively impact the design and evaluation of NoCs. For example, smoothing out periods of high traffic will result in an NoC that becomes saturated during key application phases. Alternatively, bringing low periods of communication up to an average will cause a designer to miss potential opportunities for power gating or DVFS in the NoC. Intervals allow us to capture fine-grain changes in traffic. However, selecting a single (random) interval is not necessarily characteristic of the entire simulation. Yet considering all intervals will be difficult to model with a Markov Chain (Sec. 6) and will yield little simulation speedup. Therefore, we group intervals that behave similarly into different *traffic phases* via clustering.

This section explores various alternative approaches to identifying *similar behaviour* between intervals through feature vectors (Sec. 5.1). Each vector contains elements (features) that measure some aspect of traffic in that interval (e.g., the injection rate). Vectors are then compared by calculating the distance between them; a clustering algorithm creates groups of intervals whose vectors are *close* together (Sec. 5.2).

5.1. Feature Vector Design

Defining similarity between intervals is non-trivial. One has to consider the elements of the feature vector, its dimensionality and scalability. In this section, we present a subset of potential feature vectors that can be used to cluster intervals into traffic phases; this discussion is not meant to be exhaustive but rather captures a range of traffic metrics and feature vector scalability.

It may be tempting to use feature vectors with many elements. There is a trade-off between capturing a range of communication attributes and the effectiveness and ease of clustering. Large feature vectors can suffer from the *curse of dimensionality* where the data available to populate the vector is insufficient for the size of the vector [4]. In addition, having a large number of observations puts additional strain on the clustering algorithm; some clustering algorithms have a complexity of $O(n^3)$ (where n is the number of vectors). We explore two different approaches to construct feature vectors:

1. **Injection Rate**: number of packets injected in an interval
2. **Injection Flows**: number of packets injected between source-destination pairs per interval

We also explored feature vectors that consider cache coherence message types. In this way, intervals with dominant read and/or write phases are clustered together. However, such an approach does not capture the spatial injection distribution of packets. As a result, intervals with similar hot spots are not clustered together. As we show in Sec. 8, this information is crucial if we expect to synthetically generate realistic traffic.

5.1.1. Injection Rate Injection rate can be captured in different ways. Considering the injection rate of all nodes (*Total Injection*) gives simple, one-dimensional feature vectors that allow us to differentiate between intervals that are experiencing high, medium or low levels of communication. The benefit of this vector is that it is easy to create. Calculating the distance between vectors and applying clustering is fast because it is one-dimensional. Yet *Total Injection* may be too simple; the total number of packets does not reveal any spatial characteristics of the traffic. Even when two vectors have similar magnitudes, their respective intervals could exhibit different spatial behaviour, such as hot spots. Using the injection rate of individual nodes alleviates some of these issues. An N-dimensional vector with per-node injection rates (*Node Injection*) captures some spatial characteristics of our applications.

5.1.2. Injection Flows *Node Injection* helps identify *injecting* hotspots – that is, nodes that *send* a lot of packets. But hot spots can also exist at a destination – that is, nodes that *receive* a lot of packets. To capture the relationship between sent and received messages, we can use flows [20]. A *flow* is the injection rate between a source and a destination. For an N-node network, there are N^2 source-destination flow pairs. We construct a feature vector (*Per-Node Flow*) that captures this information. This vector scales quadratically with the number of nodes. Sufficient data has to be present in the traffic or else the feature vector falls prey to the curse of dimensionality.

Table 2: Different traffic feature vectors for an N-node network

Feature Vector	# of Features	Description
Total Injection	1	Total number of packets injected
Node Injection	N	Packets injected for each network node
Row-Column Flow	N	Packets injected between rows and columns of the network
Per-Node Flows [20]	N^2	Packets injected between each source-destination pair

We can simplify *Per-Node Flow* feature vectors by aggregating nodes into rows and columns (*Row-Column Flow*). Each element of the vector corresponds to the number of packets sent by a row of nodes to a column of nodes. We use the words *row* and *column* for simplicity – the actual mapping of nodes in the network does not have to be grid-like.

5.1.3. Summary We introduce four potential feature vectors to classify traffic phases. These are summarized in Table 2. Each vector has its own pros and cons, and some vectors are better suited for either a macro or micro scale. We explore the impact of different feature vectors in Sec. 8.

5.2. Clustering Methods

Feature vectors are used to cluster intervals into traffic phases. We calculate the distance between vectors and then apply a clustering method. Distance calculations are affected by the dimensionality of the vector (i.e. number of features); therefore, feature vectors that scale poorly (Table 2) lead to high overhead and modelling time. In this section, we look at two clustering approaches: partitional and hierarchical and weigh their benefits. Ultimately, we use different approaches at different granularities, as we discuss in Sec. 6.

5.2.1. Partitional Clustering Partitional clustering designates a feature vector that is *central* to each group; we use Euclidean distance as a measure of closeness between vectors. Although k-means is the most popular, we use k-medoids (specifically, Partitioning-Around-Medoids or PAM). PAM performs a pairwise comparison of the distances between a vector (V) and every other vector in the group. Although slower than k-means, PAM is able to provide the central vector (medoid) for each group. This allows us to select the interval that is most representative of its traffic phase. Partitional clustering is an NP-hard problem, however heuristics are available that keep its complexity and speed low [46].

Partitional clustering requires the number of traffic phases (or clusters k) to be an input to the algorithm. Formal methods exist [34] to determine an optimum k value, though not all methodologies agree on the same k. Two common methods that estimate an optimal k are Average Silhouette Width (ASW) [35] and the Calinksi-Harabasz (CH) index [6]. We explore the effects of k using these two methods in Sec. 8.1.

5.2.2. Hierarchical Clustering Hierarchical clustering is an efficient, deterministic approach to grouping traffic phases. However, it has a complexity of $\mathscr{O}(n^3)$ (where n is the number of vectors), making it better suited to clustering smaller data sets. Hierarchical clustering creates a tree (a dendogram) of all feature vectors, linking vectors together based on distance and a linkage criterion[5]. The algorithm iteratively combines the two clusters that have the least impact on the sum of squares error. Different levels of the tree indicate which vectors belong to which clusters; the tree can be cut at a user-defined level to provide the desired number of traffic phases. We use the L-method [36] to determine the appropriate number of clusters in hierarchical clustering.

6. Injection Process

In Sec. 5, we introduce macro- and micro-level granularities for intervals. Each *macro-interval* is further broken down into *micro-intervals*. Then, we group intervals into traffic phases using clustering. Next, we demonstrate how to construct a hierarchical Markov Chain for the macro- and micro-levels. Fig. 1 shows an overview of our approach, where macro-scale traffic has been decomposed into micro-scale intervals, and two Markov Chains govern the transitions between phases.

Markov Chains are typically used to model stochastic processes. A Markov Chain is made up of a number of states, with transition probabilities defined for moving from one state to another. In our case, states correspond to macro- or micro-phases, and transitioning from one phase to another allows us to accurately replicate the time-varying behaviour of an application's injection process.

Macro Scale Given long application runtimes, the number of intervals at the macro level ranges from hundreds to thousands. This variability and the resulting large number of vectors means hierarchical clustering is not a good fit because of its $\mathcal{O}(n^3)$ complexity; therefore we use PAM at the macro scale. PAM gives us the medoid of each traffic phase – that is, a single macro interval that best represents the macro phase. Having a single macro-interval for each phase significantly reduces the amount of data modelled. Once we have the medoid for each traffic phase, we pass them to our micro model and analyze the traffic at a finer granularity. We create a micro model for each macro-interval selected.

Micro Scale The micro scale looks at only a small subset of the overall traffic. Dividing a macro-interval into micro-intervals allows us to capture the injection process at a finer granularity; this is necessary to capture *bursty* fluctuations in traffic that can greatly influence network performance. Unlike at the macro-level, we are not looking for a single representative interval per traffic phase. A single representative interval does not contain enough data to form an accurate micro-level model. Since we do not need a medoid, we use hierarchical clustering at the micro scale.

Hierarchy We model multiple Markov Chains for our hierarchy of macro- and micro-levels. One Markov Chain governs transitioning between macro-phases. For each macro-phase we define another Markov Chain for its micro-phases. Fig. 1 shows the two level hierarchy with two macro-phases and three micro-phases. An important property of Markov Chains is that they can reach *equilibrium* (π). That is, after infinite time, the Markov Chain converges to a steady state where the probability of being in a given state is constant. We exploit this property to achieve significant speedups over full-system simulation in Sec. 10.

[5]We use minimum-variance based on Ward's method [47].

Processor	16 Out-of-Order cores, 4-wide, 80-instruction ROB
L1 Caches	16 Private, 4-way, 32 KB
L2 Caches	16 Private, 8-way, 512 KB
Coherence Protocol	Directory-Based MOESI (blocking)

Network	A	B	C
Topology	Mesh	Mesh	Flattened Butterfly [23]
Channel Width	8 bytes	4 bytes	4 bytes
Virtual Channels	2 per port	2 per port	4 per port
Routing Alg.	XY	Adaptive XY-YX	UGAL
Buffer Depth	8 flits		
Router Pipeline	4 stages		

Table 3: Simulation configurations

7. Methodology

We evaluate SynFull using a 16-core CMP with the configuration given in Table 3. Each node contains a core, private L1 cache, private L2 cache and a directory. Data is collected using FeS2, a full-system simulator [31] integrated with Booksim, a cycle-accurate network simulator [19]. We run PARSEC [5] and SPLASH-2 [48] benchmarks with the `sim-small` input set. All benchmarks are run to completion with the exception of facesim, which was capped at three hundred million cycles.

To generate the SynFull models, we collect traces from full-system simulation assuming an ideal fully-connected NoC with a fixed one cycle latency. Using an ideal network ensures that our model does not contain artifacts of the network, and therefore cannot be influenced by a certain topology, routing algorithm, etc. Thus a single model can be used to simulate a wide range of NoC configurations. We compare NoC performance of our synthetically generated network traffic with full-system simulation and trace-based simulation using state-of-the-art packet dependency tracking based on Netrace [18].

To demonstrate that our methodology is network agnostic, we compare against three different NoC configurations (Table 3). That is, we can apply SynFull to different NoC configurations and capture similar behaviour to what would have been exhibited by full-system simulation, regardless of the network's configuration.

8. SynFull Exploration

Our proposed SynFull traffic model has a number of parameters that can be changed. Initially, it is not obvious or intuitive what the values of these parameters should be to accurately model traffic. In this section, we explore these model parameters and discuss their affects on the generated network traffic, NoC performance and model accuracy. Specifically, we: (i) Evaluate how the number of macro phases affect NoC performance; (ii) Demonstrate how to adjust the amount of congestion at the micro level with different feature vectors;

Benchmark	ASW NI	ASW TI	CH NI	CH TI
Lu	2	2	2	10
Raytrace	2	2	8	7
Swaptions	2	2	2	6

Table 4: Number of macro phases for different formal methods and feature vectors

and (iii) Explore how the size of time intervals can change traffic generated by SynFull.

We look at the effects of different parameters quantitatively on three benchmarks: Lu (contiguous), Raytrace, and Swaptions. The domains of these benchmarks are different; Lu is a high-performance computing application that relies heavily on barriers as its synchronization primitive, Raytrace is a graphics-based benchmark that relies heavily on locks, and Swaptions deals with financial analysis and is not very communication intensive. Once we have explored the parameters across these three benchmarks, we make recommendations to achieve NoC performance estimates that are accurate with respect to full-system simulation results.

8.1. Macro Phases

Macro phases constitute the largest granularity for our model – a macro interval can be several hundred thousand cycles long. The number of macro phases we need to model is a function of application behaviour. In order to determine this number, we apply formal methods (CH and ASW) to a particular clustering of macro-intervals. Clustering is also affected by the feature vectors used. The number of macro phases used by SynFull affects the variety of traffic exhibited at the macro granularity.

We explore two feature vectors at the macro-level: Total Injection (TI) and Node Injection (NI). Our goal is to reduce the clustering overhead at the macro level because the number of observations can be quite large and varies by benchmark – TI and NI require the least processing time of all the proposed feature vectors. Using these two feature vectors, we apply CH and ASW to the clustering to determine the optimal number of macro-phases. We assume macro-intervals of 500,000 cycles and micro-intervals of 200 cycles, and the NI feature vector at the micro level. We create our model from full-system simulation with an ideal network, and then apply the traffic to Network A. We compare the resulting average packet latency to full-system simulation (FSYS); this metric includes the time a node is queued waiting to be injected into the network.

Table 4 shows the number of phases suggested by the ASW and CH formal methods for the NI and TI feature vectors, and Fig. 5 shows the results of using those parameters. There is little variation in average packet latency when tweaking macro parameters for Lu and Swaptions. Raytrace, however, shows more accuracy using the CH index, which recommends 7 or 8 macro phases with TI and NI, respectively. Raytrace traffic has several macro intervals that deviate from the norm, likely due to the several thousand locks it uses [48], and therefore should be modelled with more macro phases. The locking in Raytrace results in an unstructured communication pattern with high

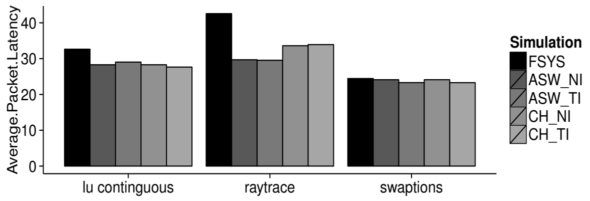


Figure 5: Macro-level sweeping of feature vectors & number of phases (Table 4).

variation. Too few macro phases would force interval outliers into phases where they do not belong.

The use of barriers in Lu results in distinct periods of low and high communication; when all threads reach a barrier there is a sudden burst of packets into the NoC. This communication pattern maps well to 2 distinct macro phases. CH+TI has 10 macro phases which results in the highest error for SynFull. Too many phases can lead to poor clustering quality because some phases will have very few, or even a single interval, associated with them. These phases are superfluous and negatively impact the Markov Chain because they will be rarely visited.

The single dimension of TI makes the clustering sensitive to fluctuations between macro intervals; that is, two high-communication macro-intervals may not be clustered together due to a small difference in total packets. This sensitivity is alleviated by using more dimensions, so that deviations in one element are neutralized by similarity in others. This helps prevent the case where we have too many phases for macro intervals; thus, we recommend NI for macro clustering and CH for the number of macro phases.

8.2. Congestion at the Micro Level

Sec. 8.1 uses Node Injection (NI) as the feature vector at the micro level. NI clusters micro intervals according to the distribution of injected packets across nodes. While this will cluster hot spots at source nodes, there are situations where hot spots exist between source-destination pairs. For example, a many-to-one communication pattern is not accurately captured by the NI vector. The Row-Column Flow (RCFlow) and Per-Node Flow (Flow) feature vectors are better suited to capturing these hot spots, allowing for the synthetically generated traffic to cause congestion as full-system simulation might.

In this section, we use CH+NI at the macro level with interval sizes of 500,000 cycles. We compare the NI feature vector to RCFlow and Flow with 200-cycle micro intervals. We run our models on Network A and show average packet latency in Fig. 6. The RCFlow and Flow vectors are more accurate with respect to full-system simulation for Raytrace; the locks used by Raytrace result in specific source-destination sharing that NI does not capture. Also important is that the two vectors did not negatively affect the accuracy for the Lu and Swaptions; that is, RCFlow and Flow did not artificially create congestion for benchmarks that do not exhibit that behaviour.

We are not only interested in average behaviour but in capturing the highs and lows of network traffic. Looking at packet

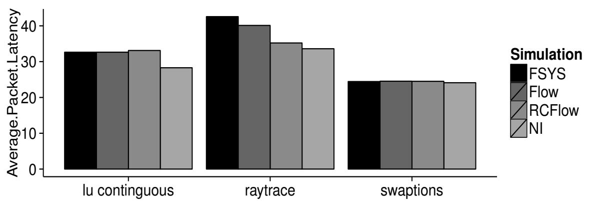


Figure 6: Micro-level sweep of feature vectors

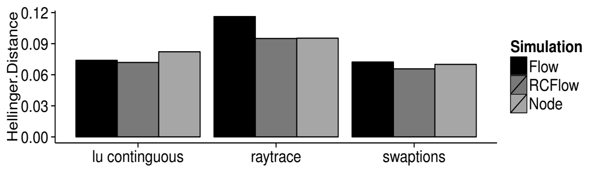


Figure 7: Hellinger distance comparing packet latency distributions of synthetic simulations to full system. Lower is better.

latency distributions, we can see the number of packets that achieve a wide range of latencies while in the network; this distribution gives insight into the congestion the network has experienced. The Hellinger Distance defines the similarity between two distributions. The Hellinger Distance is defined in Equation 2, where P and Q are two discrete distributions (in our case, packet latency distributions), and p_i and q_i are the i^{th} element of P and Q, respectively.

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

Fig. 7 shows the Hellinger Distance for our synthetic traffic latency distributions compared to full-system simulation. The lower the distance, the more similar the latency distributions are. We can see that, although the error in average packet latency is less for Raytrace with the Flow vector (Fig. 7), the distribution of packet latencies is not as close to full system as RCFlow. This is because the Flow vector causes more high latency packets than full-system simulation, driving up the average packet latency with more congestion than necessary. In all cases, RCFlow is more similar to the desired packet latency distribution exhibited by full-system simulation, and its error in average packet latency is comparable to Flow. Therefore, we recommend RCFlow for micro clustering.

8.3. Time Interval Size

So far we have used 500,000 cycles per macro interval and 200 cycles per micro interval. This results in $500,000/200 = 2,500$ micro intervals (observations) per macro interval, which is low enough to keep hierarchical clustering time reasonable. Now, we sweep the macro and micro interval sizes together so that they always result in 2,500 observations. We use CH+NI at the macro level, and compare the RCFlow and Flow feature vectors at the micro level with various interval sizes.

Fig. 8 shows the average packet latency for SynFull traffic with different interval sizes. There is not a clear cut interval size that is best for every application. RCFlow works best with a micro-interval size of 100 cycles for Raytrace, but performs worse for Lu. Applications may exhibit different periodic behaviour at the micro level depending on their algorithm or an application may not have periodic behaviour at all. When using large interval sizes of 500 cycles or more, we risk not capturing bursty application traffic because deviations in injection rate get averaged out across the interval. For applications without bursty traffic, large interval sizes work well because the standard deviation of packets injected over time is low. Choosing a universal interval size for all applications may lead to slightly less accurate SynFull results for a subset of benchmarks. In future work, we will investigate automatically determining the interval size based on application traffic.

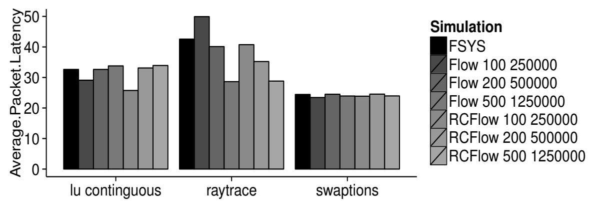


Figure 8: NoC performance for different interval sizes.

	Macro-Level Model	Micro-Level Model
Feature Vector	Node Injection	RCFlow
Cluster Algorithm	PAM	Hierarchical
Formal Method	CH Index	L-Method
Interval Size	500,000	200

Table 5: Final SynFull Configuration

8.4. Parameter Recommendations

Based on the results presented in this section, we make some recommendations regarding model parameters used in SynFull. Changing the feature vector at the macro level does not have a significant effect on network performance. However, in terms of the clustering quality (recall TI vs. NI for Lu's barriers), using the NI feature vector with the CH index yields the best results. For feature vectors at the micro level, it is important to select a vector that adequately captures hotspots. Both RCFlow and Flow feature vectors show good results, however RCFlow scales better with the number of nodes being simulated and takes significantly less time to model (typically, an RCFlow model takes a few minutes to generate whereas a Flow model can take over 20). Finally, the interval sizes of the macro and micro levels can greatly influence traffic generated by SynFull. For the rest of this paper, we will use 200 cycles at the micro-level and $500,000$ cycles at the macro-level.

9. Results

We evaluate SynFull with PARSEC and SPLASH-2 benchmarks on the three network configurations introduced in Table 3. We compare SynFull against full-system simulation and trace simulation with packet dependences. For SynFull, we use the recommendations in Sec. 8.4 summarized in Table 5. Initially for both SynFull and trace simulations, the number of cycles simulated is equal to the number of cycles required to

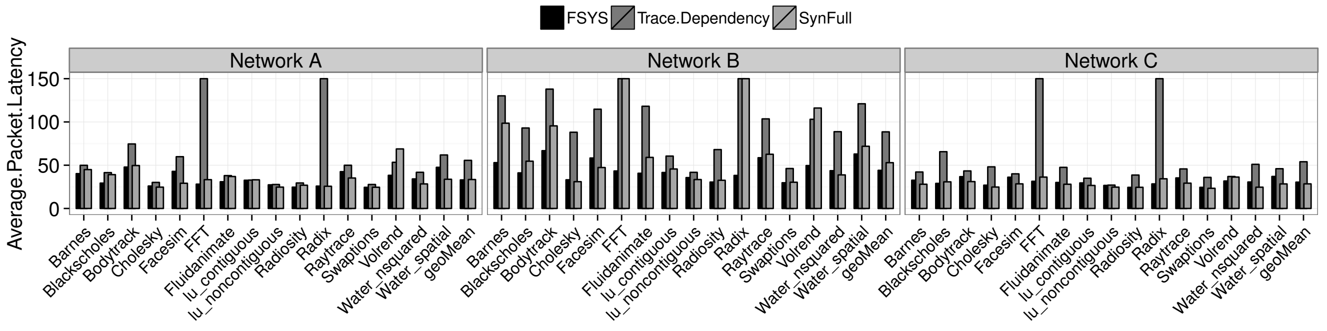


Figure 9: NoC performance. Bars that reach the top of the y-axis (e.g. FFT) are truncated so that other results may be seen more clearly.

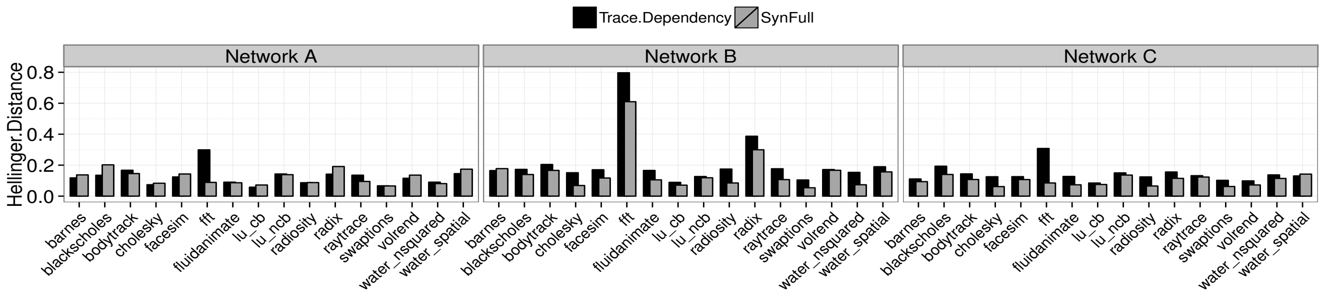


Figure 10: Comparing similarity of packet latency distributions with full-system simulation

complete a full-system simulation of the benchmark with an ideal network. Later, we explore early simulation termination due to the Markov Chain reaching steady-state.

Incorporating packet dependences into trace simulation improves the fidelity of traditional trace-based simulation on NoCs [18]. Traditionally, packets from a trace are injected into the network with no regard for when they arrive at their destinations. This is unrealistic due to the reactive nature of some packets, as explained in Sec. 4. Dependence tracking aims to capture the reactive nature of packets, and only inject them when their requesting packet has arrived; the injection of dependent packets is triggered by another packet's arrival, rather than the timestamp of the original trace.

We compare average packet latency across simulation methodologies (Fig. 9). SynFull does very well on NoCs A and C, with a geometric mean error of 8.9% and 9.5% across all benchmarks. NoCs A and C are reasonably well-provisioned; most applications do not experience significant contention on these networks. SynFull achieves accurate average packet latency both for applications that do not stress the network (e.g. Cholesky Radix, Radiosity, Swaptions), and applications that do stress the network (e.g. Barnes, Bodytrack, Fluidanimate). Network throughput has similar accuracy, with geometric mean errors of 11.78% and 12.42% for NoCs A and C. Running an ideal network trace with dependences does not fair as well (geometric mean packet latency error of 18% and 12.8% for NoCs A and C) because dependences are not tracked at the application level. While reactive packets are throttled correctly waiting on the arrival of predecessor packets, independent packets continue to be injected according to their timestamp. For most applications, especially FFT and Radix, this has a significant impact on NoC performance.

NoC B is the least provisioned of the 3 networks. As a result, discrepancies in initiating packet injections are more pronounced for both SynFull (16.1% packet latency error and a 16.11% throughput error) and Traces (30.2% packet latency error). Traces with dependences have significant error even for applications with low communication requirements (e.g. Radiosity), while SynFull is capable of reproducing similar NoC performance for benchmarks of this type. Some applications running on NoC B have significant error for both SynFull and Traces. In particular, Radix and FFT (excluded from geoMean calculations) run off the chart. These are special cases where the application has macro-level intervals with very high injection rates that dwarf the injection rate across the rest of the application. For example, running FFT on an ideal network, there is a spike of several macro-intervals during the middle of simulation with an order of magnitude larger injection rate than other intervals. When running FFT in full-system simulation on the considerably less provisioned NoC B, the spike is longer but with a much lower (less than 50%) injection rate. This is due to application-level dependences and the core's reorder buffer throttling instruction issue which in turn throttles network injection. However, this is an extreme case and one not typically found in many of the applications we consider; we are investigating techniques to adapt our model to these scenarios.

We discussed the importance of packet latency distributions

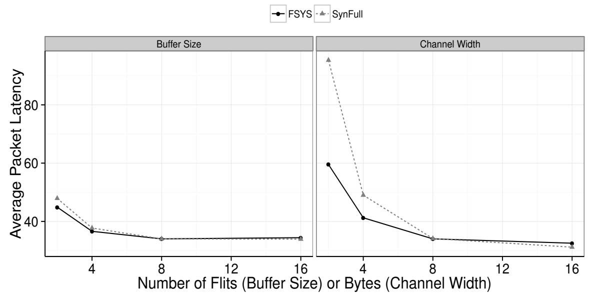


Figure 11: Two case studies of packet latency trends across all workloads

in Sec. 8.2, and use the Hellinger Distance to compare distributions to full-system simulation. Fig. 10 shows packet latency distribution Hellinger Distance for SynFull and Traces compared to full-system simulation. Consistent with the average packet latency error, SynFull modelling FFT (NoC B) has a large Hellinger Distance which indicates that the resulting distribution does not resemble the latency distribution seen in full-system simulation. Outside of FFT, our technique fares well for PARSEC and SPLASH-2 applications. Applications with low communication requirements typically have the lowest Hellinger Distance because both SynFull and full-system simulation do not have a large tail in the distribution. For applications with more bursty behaviour, Hellinger Distances are greater but still comparable.

Traces that perform well in average packet latency on NoCs A and C perform better than SynFull in Hellinger Distance (e.g. Cholesky, Lu, Radiosity). These applications have low communication requirements. As a result, the issue of independent messages flooding the network is minimized on a well-provisioned network, and the trace faithfully reproduces application traffic. Due to the randomness associated with Markov Chains, SynFull phases do not exactly coincide the way a trace would. As a result, we have slightly higher Hellinger Distances, but the results are still comparable. However, when comparing applications across all domains, SynFull is the clear winner.

9.1. Capturing Trends

While absolute error values are useful, designers expect a methodology to accurately capture the relationship between networks designs. That is, if one network performs better than another in full-system simulation, then the trend should be the same when using SynFull. Here we demonstrate that the relationship is captured with more intuitive trends. Specifically, we perform two separate sweeps on channel width and virtual channel buffer size. In the first sweep, we look at networks with 16, 8, 4, and 2 byte channel widths. In the second sweep, we look at networks with 16, 8, 4, and 2 flits per buffer. Intuitively, larger channel widths and buffer sizes would lead to better performance than smaller ones. Indeed, this is the case as shown in Fig. 11; results are averaged across all workloads.

Packets are subdivided into flits based on the channel width. Our simulations use 8-byte control packets and 72-byte data packets. From Fig. 11 (right) we see that there is not much difference in performance between an 8 and 16 byte channel width. This is because a 16 byte channel width only improves transmission of data packets, since 8 bytes is all that is needed for a control packet. As the channel width decreases, so too does performance due to the increased serialization latency of all packets. Buffer depth also affects performance. Smaller buffers increases the latency of packets because flits have to wait until space becomes available before proceeding towards their destination. In this case study, Fig. 11 (left) shows that SynFull captures the relationship almost perfectly.

Overall, SynFull is a superior approach to trace dependences in terms of fidelity. SynFull is less prone to error across a variety of applications and stresses an NoC in the same way an application would in full-system simulation. SynFull also captures the same trends found through full-system simulation. High accuracy is an important attribute of SynFull; independent of its accuracy relative to full-system simulation, SynFull provides a meaningful collection of synthetic traffic models that capture a diverse range of application and cache coherence behaviour making SynFull an invaluable tool in a NoC designer's arsenal. In Sec. 10, we explore the speed of SynFull relative to full-system simulation, and how it can be further accelerated using a special property of Markov Chains.

10. Exploiting Markov Chains for Speedup

Simply running SynFull for the same number of cycles as full-system simulation results in significant speed up – this is because SynFull itself does not require much processing time. The NoC simulator is the limiting factor, but is still substantially faster than a full-system simulator. We can further improve the simulation time of SynFull by exploiting the stationary distribution of Markov Chains. An important property of Markov Chains is that they can reach *equilibrium*. That is, after infinite time, the Markov Chain converges to a steady state where the probability of being in a given state is constant.

In SynFull, when the macro-level Markov Chain has converged to its equilibrium, we exit the simulation prematurely. This implies that all traffic phases have been simulated for an adequate time, and our simulation has reached its steady state. We cannot apply the same methodology to trace-based simulation because it follows the same progression as full-system simulation. If we exit a trace prematurely, we may miss out on a large period of bursty communication or low communication, both of which would give very different overall NoC performance results. For example, if trace simulation of FFT were to exit early, it would not reach the large spike of macro intervals, leading NoC researchers to draw incorrect conclusions.

Fig. 12 shows the average speedup of traces, SynFull, and with SynFull exiting simulation at steady-state (SynFull_SS). The numbers are calculated by averaging the total runtime of simulations across each of the three network configurations (A, B, and C) for each application. Without steady-state, SynFull

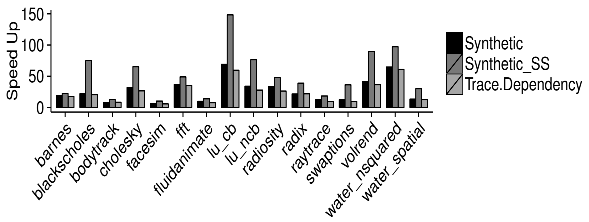


Figure 12: The average speedup over full system simulation

and Trace Dependency speed-ups are very similar since they simulate the same number of cycles. The simulation bottleneck here is the NoC itself and not the methodology for driving traffic. With steady state, we achieve substantial speedup; speedup is as high as $\sim150\times$ and over $50\times$ on average.

SynFull models two Markov Chains; however, we only exit when steady state is reached at the macro level. We could potentially end a macro-interval early by observing steady state at the micro level. However, this would result in different length macro intervals, which could negatively affect performance accuracy. For example, imagine a low injection macro interval reaches steady state very early while a high injection macro interval does not. There would be a disproportionate amount of high injection to low injection, negatively impacting the accuracy of our model. By only observing steady state at the macro-level Markov Chain, we achieve similar error compared to running SynFull to completion; a full run of SynFull has a geometric mean error of 8.9%, 16.1%, and 9.5% across networks A, B, and C, while SynFull with steady state yields errors of 10.5%, 17.1%, and 9.1%.

11. Related Work

Simulation acceleration. There has been considerable work done to improve simulation time. FPGA-based acceleration has been proposed [11, 43]. FIST implements an FPGA-based network simulator that can simulate mesh networks with significant speed up over software simulation [32]. User-level simulators exist as an alternative to full-system simulation for exploring thousands of cores [7, 29]. ZSim exploits parallel simulation with out-of-order core models [37]. Sampling for microarchitectural simulation has been widely explored [38, 39, 49] and has received renewed attention for multi-threaded and multi-core processors [1, 8]. Zhang et al. leverage SimPoints for network traffic so that they may speed up simulations for parallel applications [51]. Hornet [33] focuses on parallelizing a NoC simulation. Simulators such as Hornet [33], ZSim [37] and Slacksim [9] are great tools but designers should still prune the design space to a few candidates prior to using such detailed simulators; SynFull bridges the gap between existing synthetic models and detailed full-system simulation.

Workload modelling. Cloning can mimic workload behaviour by creating a reduced representation of the code [3, 21]. Much of this work focuses on cloning cache behaviour; SynFull can be viewed as creating clones of cache coherence behaviour to stimulate the network. Creation of synthetic benchmarks for multi-threaded applications has been explored [17]; this work generates instruction streams that execute in simulation or on real hardware. Our work differs as we reproduce communication patterns and coherence behaviour while abstracting away the processor and instruction execution. MinneSPEC [24] provides reduced input sets that effectively match the reference input for SPEC2000; rather than focus on input set or instruction generation, we provide a reduced set of traffic based on the steady state of a Markov Chain.

Workload Design and Synthetic Traffic. Synthetic workloads have been a focus of research long before NoCs emerged [16, 42]. Statistical profiles can be used to generate synthetic traces for microarchitectural performance analysis [14]. Methods for synthetic trace generation at the chip level have also been proposed [44, 45]; Soteriou et al. propose a 3-tuple statistical model that leverages self-similarity to create bursty synthetic traffic [41]. To our knowledge, there has been no work done to synthetically generate network traffic that includes cache coherence. The benefits of such an approach allows us to remove the necessity for full-system simulation while still allowing works that exploit coherence traffic. In addition, most statistical models do not compare generated traffic with full-system simulations, ignoring performance metrics such as packet latency.

12. Conclusion

Full-system simulation is a long and tedious process; as a result, it limits the range of designs that can be explored in a tractable amount of time. We propose a novel methodology to accelerate NoC simulation. SynFull enables the creation of synthetic traffic models that mimic the full range of cache coherence behaviour and the resulting traffic that is injected into the network. We accurately capture spatial variation in traffic patterns within and across applications. Furthermore, burstiness is captured in our model. These two attributes lead to a model that produces accurate network traffic. We attain an overall accuracy of 10.5% across 3 configurations for all benchmarks relative to full-system simulation. Furthermore, our technique uses the steady-state behaviour of Markov chains to speedup simulation by up to $150\times$. SynFull is a powerful and robust tool that will enable faster exploration of a rich design space in NoCs. SynFull can be downloaded at `www.eecg.toronto.edu/~enright/downloads.html`

Acknowledgements

This research was funded by a gift from Intel. Additional support was provided by the Canadian Foundation for Innovation and the Ontario Research Fund. We thank Mike Kishinevsky and Umit Ogras for their invaluable feedback and insight while developing SynFull. We further thank Emily Blem, Andreas Moshovos, Jason Anderson, the members of the Enright Jerger research group and the anonymous reviewers for their thoughtful and detailed feedback on this work.

References

[1] E. Ardestani and J. Renau, "ESESC: A fast multicore simulator using time-based sampling," in *Proc. of Intl. Symposium on High Performance Computer Architecture*, 2013.

[2] J. H. Bahn and N. Bagherzadeh, "A generic traffic model for on-chip interconnection networks," *Network on Chip Architectures*, p. 22, 2008.

[3] G. Balakrishnan and Y. Solihin, "WEST: Cloning data cache behavior using stochastic traces," in *Proc. of Intl. Symposium High Performance Computer Architecture*, 2012.

[4] R. Bellman, *Adaptive Control Processes: A Guided Tour*, ser. A Rand Corporation Research Study Series. Princeton University Press, 1961.

[5] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[6] T. Caliński and J. Harabasz, "A dendrite method for cluster analysis," *Comm in Statistics-theory and Methods*, vol. 3, no. 1, pp. 1–27, 1974.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc of Supercomputing (SC)*, 2011, p. 52.

[8] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Intl. Symp. Performance Analysis of Systems and Software*, Apr. 2013.

[9] J. Chen, L. K. Dabbiru, D. Wong, M. Annavaram, and M. Dubois, "Adaptive and speculative slack simulations of CMPs on CMPs," in *Proc. of Intl. Symposium on Microarchitecture*, 2010.

[10] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching and MSHRs," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 3, October 2011.

[11] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proc of the International Symposium on Microarchitecture*, 2007, pp. 249–261.

[12] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann, 2003.

[13] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Aergia: exploting packet latency slack in on-chip networks," in *Proc. of Intl. Symposium on Computer Architecture*, 2010.

[14] L. Eeckhout, K. De Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," in *Intl. Symp. Performance Analysis of Systems and Software*, 2000, pp. 1–6.

[15] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *Intl Symp on High Performance Computer Architecture*, 2011, pp. 169–180.

[16] D. Ferrari, *On the foundations of artificial workload design*. ACM, 1984, vol. 12, no. 3.

[17] K. Ganesan and L. John, "Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors," *IEEE Trans. on Computers*, vol. 99, 2013.

[18] J. Hestness, B. Grot, and S. W. Keckler, "Netrace: dependency-driven trace-based network-on-chip simulation," in *Proc. of the 3rd International Workshop on Network on Chip Architectures*, 2010, pp. 31–36.

[19] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Intl. Symp. Performance Analysis of Systems and Software*, 2013.

[20] Y. Jin, E. J. Kim, and T. Pinkston, "Communication-aware globally-coordinated on-chip networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 242 –254, Feb. 2012.

[21] A. Joshi, L. Eeckhout, R. Bell, and L. John, "Cloning: A technique for disseminating proprietary applications at benchmarks," in *Proc. of IEEE Intl Symposium Workload Characterization*, 2006.

[22] T. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc of the Intl Symp on Computer Architecture*, 2004.

[23] J. Kim, J. Balfour, and W. Dally, "Flattened Butterfly Topology for On-Chip Networks," in *Proc of the International Symposium on Microarchitecture*, 2007, pp. 172–182.

[24] A. KleinOsowski and D. J. Lilja, "MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research," *Computer Architecture Letters*, vol. 1, June 2002.

[25] T. Krishna, L.-S. Peh, B. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Proc. of the International Symposium on Microarchitecture*, 2011.

[26] M. Lodde, J. Flich, and M. E. Acacio, "Heterogeneous NoC design for efficient broadcast-based coherence protocol support," in *International Symposium on Networks on Chip*, 2012.

[27] S. Ma, N. Enright Jerger, and Z. Wang, "Supporting efficient collective communication in NoCs," in *Proc of Intl. Symposium on High Performance Computer Architecture*, 2012, pp. 165–177.

[28] M. Martin, M. Hill, and D. Sorin, "Why on-chip cache coherence is here to stay," *Comm of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[29] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proc. of Intl. Symposium on High Performance Computer Architecture*, Jan. 2010, pp. 1 –12.

[30] A. Mishra, O. Mutlu, and C. Das, "A heterogeneous multiple network-on-chip design: An application-aware approach," in *Proc. of the Design Automation Conference*, 2013.

[31] N. Neelakantam, C. Blundell, J. Devietti, M. M. Martin, and C. Zilles, "FeS2: A Full-system Execution-driven Simulator for x86," Poster presented at ASPLOS, 2008.

[32] M. Papamichael, J. Hoe, and O. Mutlu, "FIST: A fast, lightweight, FPGA-friendly packet latency estimator for NoC modeling in full-system simulations," in *Intl Symp on Networks on Chip*, 2011.

[33] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas, "HORNET: A cycle-level multicore simulator," *IEEE Trans. Comput-Aided Design Integr. Circuits Syst.*, vol. 31, no. 6, 2012.

[34] A. Reynolds, G. Richards, B. De La Iglesia, and V. Rayward-Smith, "Clustering rules: a comparison of partitioning and hierarchical clustering algorithms," *Journal of Mathematical Modelling and Algorithms*, vol. 5, no. 4, pp. 475–504, 2006.

[35] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[36] S. Salvador and P. Chan, "Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms," in *Int. Conf. on Tools with Artificial Intelligence*, 2004, pp. 576–584.

[37] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. of the International Symposium on Computer Architecture*, 2013.

[38] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Parallel Architecture and Compilation Techniques*, 2001, pp. 3–14.

[39] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. of Architecture Support for Programming Languages and Operating Systems*, 2002, pp. 45–57.

[40] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[41] V. Soteriou, H. Wang, and L.-S. Peh, "A statistical traffic model for on-chip interconnection networks," in *MASCOTS*, 2006, pp. 104–116.

[42] K. Sreenivasan and A. Kleinman, "On the construction of a representative synthetic workload," *Comm of the ACM*, vol. 17, no. 3, pp. 127–133, 1974.

[43] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovic, and D. Patterson, "A case for FAME: FPGA architecture model execution," in *Proc. of Intl Symposium on Computer Architecture*, 2010.

[44] L. Tedesco, A. Mello, L. Giacomet, N. Calazans, and F. Moraes, "Application driven traffic modeling for NoCs," in *Proc of the 19th Symp on Integrated Circuits and Systems Design*. ACM, 2006, pp. 62–67.

[45] G. V. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *IEEE Trans on Very Large Scale Integration Systems*, vol. 12, no. 1, pp. 108–119, 2004.

[46] T. Velmurugan and T. Santhanam, "Computational complexity between k-means and k-medoids clustering algorithms for normal and uniform distributions of data points," *Journal of Computer Science*, vol. 6, no. 3, p. 363, 2010.

[47] J. H. Ward Jr, "Hierarchical grouping to optimize an objective function," *J. Amer. Statist. Assoc.*, vol. 58, no. 301, pp. 236–244, 1963.

[48] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Intl Symp on Computer Architecture*, 1995, pp. 24–36.

[49] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. of Intl Symposium on Computer Architecture*, 2003.

[50] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Intl Symp on Microarchitecture*, 2009.

[51] Y. Zhang, B. Ozisikyilmaz, G. Memik, J. Kim, and A. Choudhary, "Analyzing the impact of on-chip network traffic on program phases for CMPs," in *Intl Symp on Performance Analysis of Systems and Software*, 2009, pp. 218–226.