

# NOCS Special Section: PDG\_GEN: A Methodology for Fast and Accurate Simulation of On-chip Networks

Kevin Macdonald, Christopher Nitta, Matthew Farrens, and Venkatesh Akella

**Abstract**—With the advent of large scale chip multiprocessors, there is growing interest in the design and analysis of on-chip networks. Full-system simulation is the most accurate way to perform such an analysis, but unfortunately it is very slow and thus limits design space exploration. To overcome this problem researchers frequently use trace-based simulation to study different network topologies and properties, which can be done much faster. Unfortunately, unless the traces that are used include information about dependencies between packets, trace-based simulations can lead one to draw incorrect conclusions about network performance metrics such as average packet latency and overall execution time.

The primary contributions of this work are to demonstrate the importance of including dependency information in traces, and to present PDG\_GEN, an inference-based technique for identifying and including dependencies in traces. This technique uses traces obtained from multiple full-system simulations of an application of interest to infer dependency information between packets and augment traces with this information. On the SPLASH-2 benchmark suite, PDG\_GEN is 2.3 times more accurate at predicting overall execution time and almost 4000 times more accurate at predicting average packet latency than traditional trace-based methods.

**Index Terms**—Modeling methodologies, Simulation.



## 1 INTRODUCTION

FROM servers to mobile phones [1], future chips will contain dozens, if not hundreds or even thousands, of processors, memories, and/or hardware accelerators connected by on-chip networks. The on-chip network is a critical component of the chip, as it constitutes a significant fraction of the area and power consumed. As a result, a “one-size-fits-all” approach to designing one is inadequate, and a thorough exploration of the design space is required. For example, the buffer sizes, number of virtual channels, topology of the network, arbitration and flow control schemes, and amount of heterogeneity can all be optimized for a given application or market segment. The most accurate way to evaluate potential on-chip network designs is through the use of full system simulation, using a real operating system running real applications. To compare two different network designs, for example, a set of full system simulations should be run for each configuration. Doing so will give the best measure of how the designs compare. Unfortunately, full system simulation is very slow. The execution time can grow quadratically with increased node counts, preventing designers from doing full system simulations with a large number of nodes.

For example, FFT benchmark from SPLASH-2 on 64 cores takes months to complete!

One commonly used method for circumventing this problem is to use a full system simulator to extract a record of network activity (a *trace*) and feed it into a trace based network simulator to evaluate various network configurations. Trace based simulations run much faster than full system simulations, and are widely used [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. Unfortunately, these traces only include information about the order of and time between packet transmissions. In real systems there are *dependencies* between packets as well - some outgoing packets cannot be generated until incoming data has been received, for example. These dependencies are analogous to data dependencies in pipelined processors, and will be referred to as *reception dependencies*.

While a trace from a given full system simulation will implicitly include the reception dependencies for that particular network configuration, the whole purpose of network simulation is to be able to vary network parameters and evaluate the results. The absence of *explicit* information about reception dependencies means that packets are often injected into the network by the simulator at a higher rate than would occur in a real system, because the simulator does not know it needs to wait for certain events to occur. The ramification of this unrealistically high packet injection rate is that measured latencies can climb dramatically for the network being analyzed, since many messages are spending an artificially large amount of time sitting in network buffers. Simulating a trace taken from a slower network on a higher performing network is also a problem,

- K. Macdonald, C. Nitta and M. Farrens are with the Department of Computer Science, University of California at Davis, One Shields Avenue, Davis, CA 95616
- V. Akella is with the Department of Electrical & Computer Engineering, University of California at Davis, One Shields Avenue, Davis, CA 95616  
Email: klmacdonald@ucdavis.edu, cjnitta@ucdavis.edu, farrens@cs.ucdavis.edu, akella@ucdavis.edu

TABLE 1  
Example Trace

Packet #	Time Sent	Source	Destination
1	20	A	C
2	22	B	C
3	24	C	D
4	26	D	A

because it may not show the simulated network's true potential since packets are injected at a lower rate than they would be in a real system. Thus, drawing any meaningful conclusion about system parameters such as overall speedup or ideal buffer size based on trace simulation results is exceedingly difficult, if not impossible. Unfortunately, many studies that use dependency-free traces do include results relating to network speedup [3], [4], [6], normalized execution time [2], [5], or network latency [2], [7], [8], [9], [10].

The rest of this paper is organized as follows: Section 2 provides a motivating example that shows why including dependency information in traces is so important, and Section 3 introduces our proposed solution. We discuss related works from the literature in Section 4. Section 5 provides a detailed description of our PDG\_GEN algorithm, and the accuracy of PDG\_GEN, as well as the feasibility of using it in conjunction with common simulation tools, is evaluated in Section 6.

## 2 MOTIVATION

### 2.1 Simple Example

To better understand the potential pitfalls of simulating networks using traces that do not include dependencies, consider the example trace shown in Table 1, obtained from a full system simulation which used a network with a single cycle latency. Executing this trace on a network simulator which also has a latency of one cycle (see Space-Time Diagram in Figure1(a)) will indicate that the program completes at time 27 (one cycle after packet 4 is sent). If the simulated network has a four cycle latency, then the program will complete at time 30 (four cycles after packet 4 is sent). As expected, there is a difference in completion time, and there will also be a change in average latency. This is shown graphically in Figure1(b).

But what if node C was actually gathering information from nodes A and B, calculating a sum, and then sending the result off to node D? And what if the sum generated by D was then sent back to node A? In this case there would be dependencies within the trace – packet 3 cannot be sent until both packets 1 and 2 arrive, for example, and packet 4 cannot be sent until it receives packet 3. There will also be some amount of “computation” time (in this case, to perform the addition) that must elapse between the reception of the last dependent packet and the transmission of the next one. Thus, in this example, if the computation time is 1 cycle then packet 3 can be sent 1 cycle after both packet 1 and 2 have been received, and packet 4 can be sent 1 cycle after receiving packet

3.

A full system trace generated using a single cycle latency network will contain none of this information. In this trace packet 1 will arrive at node C at time 21, packet 2 at time 23, a single cycle will be spent performing the addition, at time 24 node C will transmit its value, and at time 27 the simulation will complete (see Figure1(c)). However, if the latency used by the full system simulator is changed to four, packet 3 will not be sent until time 27 (since packet 2 is received at time 26), which delays the reception of packet 3 until time 31. This in turn delays the transmission of packet 4 until time 32, and the completion time climbs to 36 (see Figure1(d)). The fact that there are dependencies in the trace that are not explicitly identified means the trace based simulation will report a completion time that is artificially low, because it is injecting packets into the network at too high of a rate. This simple example highlights the importance of including reception dependencies in traced based network simulation.

### 2.2 Real World Example

The results of the intuitive example from the previous subsection can be corroborated by comparing full system simulation results (which is the true indicator of performance) with the output of a trace based simulation, where the trace is generated on one network topology, and then used in a network simulation of a different topology. The full system simulator Simics 3.0 [12] and the memory and on-chip network timing model GEMS 2.1.1 [13] were used to perform this experiment.

We configured Simics to model a 16 core processor using a fully connected single cycle network, and ran a 1 million complex data point FFT benchmark from the SPLASH-2 parallel benchmark suite [14] and created a trace of network activity. This trace was used by Garnet [15] (the network simulator inside GEMS) in network-only standalone mode connected single cycle latency network. In this simulation each packet was injected into the network at the timestamp specified in the trace, and as expected the results of this simulation matched the results obtained from Simics. The same trace was then used to re-run the network-only simulation on three different topologies: a torus, a mesh, and a fattree. Additionally, the actual FFT full system simulation was re-run on a torus, mesh, and fattree. A comparison of the trace results and full system simulation results are presented in 2.

As Figure 2(a) shows, the total execution time reported by the trace based simulations change very little for different topologies because each packet is always injected into the network at a fixed time. Figure 2(b) shows that average network latency was not severely affected until the trace was run on a network with a low enough bandwidth that congestion began to occur. These results indicate that a trace by itself represents the packet injection distribution for the specific network configuration on which the trace was collected - however, when it is

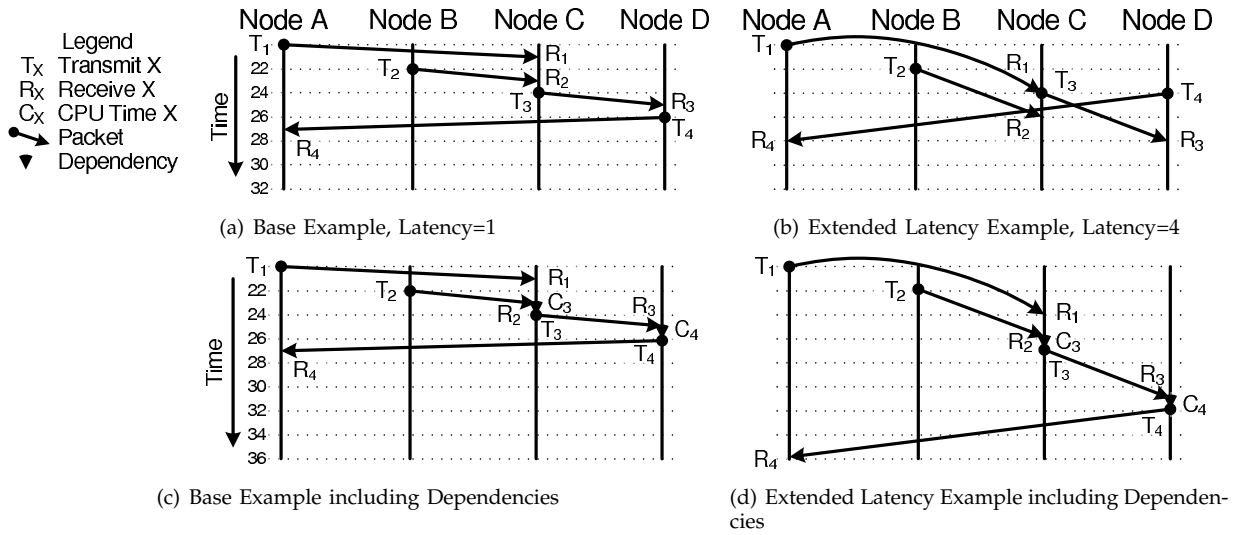


Fig. 1. Example space-time diagram (without dependencies (a), (b), and with dependencies (c) and (d))

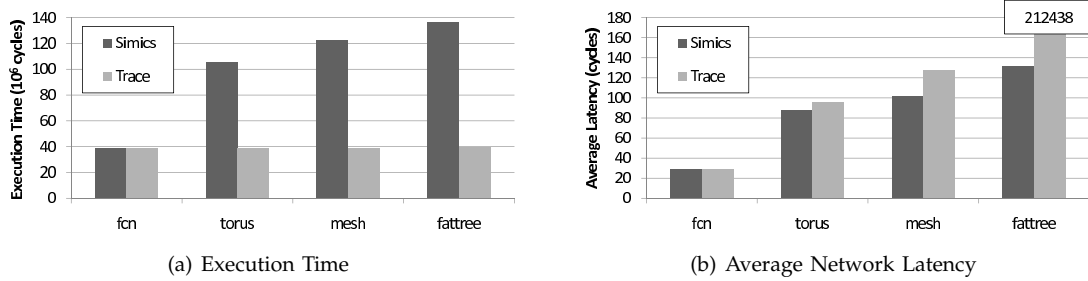


Fig. 2. 1M point FFT execution time (a) and average network latency (b) for Simics full system simulation and trace based network simulation

used on a different network configuration, it no longer represents the actual packet injection distribution for the application and hence can yield erroneous results.

### 3 PROPOSED SOLUTION

Augmenting a trace with information about the packet dependencies inherent in the application would make the packet injection times closer to what would actually happen if the application was re-run on each new network configuration, and would lead to simulation results that are more meaningful and reliable. An obvious approach is to modify the existing simulator to explicitly extract the dependencies when generating the trace; unfortunately, this approach is unrealistic because dependency tracking in a shared memory system requires maintaining dependency lists for each memory location. Furthermore, the dependency list for an operation result is the union of the operands' dependency lists. While it is clear that explicit dependency information cannot be extracted, it can be inferred from a series of full system simulations of an application of interest, and then adding this information to the trace. Using a different network configuration for each full system run exposes information about the dependency relationships between different packets, which can be extracted by comparing the different traces. Using this approach we are able to create a *Packet Dependency Graph* (PDG), which is essen-

tially a trace augmented with dependency information. This PDG can be used by a traditional cycle-accurate network simulator with only minor code changes, and allows researchers to utilize fast, reconfigurable network simulators in their research while retaining much of the fidelity of full system simulators.

The algorithm that produces a PDG from a set of input traces will be referred to as the PDG\_GEN algorithm, and it has been validated against the SPLASH-2 parallel benchmark suite running on the Simics full system simulator using the GEMS memory and on-chip network timing model. PDG\_GEN achieves an average error rate of 19.9% in end-to-end execution time and an average error rate of 1.91% in average packet latency, compared to full system simulations. This is a substantial improvement over traditional timestamp traces, which have average error rates of 45.4% and 7550% in execution time and average packet latency, respectively.

### 4 RELATED WORK

The relevant related work can be classified into three broad categories - software based functional/timing simulation, FPGA based emulation of either the functional or the timing model (or both), and high-level workload modeling using statistical techniques. BookSim 2.0 [16] is one of the first and most basic network-on-chip simulators. It does not use traces from real applications,

but instead uses synthetic traffic to predict the average latency of a network. Garnet [15] is the successor to BookSim, and it incorporates detailed timing and power models. In its *stand-alone* configuration it also uses traces without any dependency information, so it suffers from the pitfalls described previously.

Simics [12] is a commercial tool that allows full-system functional simulation. However, as the benchmarking data in [17] shows, it is very slow and does not scale beyond approximately 16 cores due to prohibitively long simulation times. Furthermore, it does not have any support for modeling on-chip networks and lacks a timing model for the underlying architecture of the network. GEMS [13] provides a timing model and network model on top of Simics, making it one of the most widely used simulators in the architecture community today. However, since it runs on top of Simics, it is (obviously) even slower and less scalable, and unsuitable for fast design space exploration. Graphite [18] is a recent effort from MIT to take advantage of multiple machines to accelerate functional simulation of as many as 1024 cores. However, Graphite does not maintain strict ordering of events in the system, and as a result it is unsuitable for evaluating on-chip networks (a point the authors themselves mention in their paper). Hestness et. al [19] recognize the necessity for dependency information within traces, and propose a technique that uses a trace of memory references from a single full system simulation to generate a trace of cache coherency traffic which includes the direct dependencies between coherence messages. While their technique shares many similarities with the contributions of this work, it is more limited in scope than the proposed PDG\_GEN algorithm because it does not have the ability to infer complex dependencies between memory references, which necessitates multiple full system runs.

In parallel with these developments, in the design automation community (where fast design space exploration and application-specific customization of networks is important) researchers are exploring the possibilities of high-level network traffic models [20]. Marculescu [21] was the first to propose a mathematical characterization of node to node traffic for the MPEG-2 application. Soteriou et. al [22] generalized this to a comprehensive network traffic model based on hop-count, burstiness and packet injection distributions. Kim et. al [23] take a somewhat different approach by proposing a closed-loop network simulation using synthetic traffic patterns representing a certain pre-determined amount of work. Wei et. al [24] propose dynamically mapping realistic multi-threaded applications to a cycle-accurate NOC simulator in order to gather detailed information about network performance.

Gratz and Keckler [25] provide a detailed analysis of why existing approaches to simulation are not appropriate, and make a case for realistic workload characterization that includes the temporal and spatial imbalances in network traffic distribution. Their work supports the

central argument in this paper, which is that failing to properly model packet injection rate leads to substantial inaccuracies. The solution advocated by Gratz and Keckler is to provide synthetic traffic models enhanced by the network traffic characteristics, while we propose the creation of traces from full-system simulation that are augmented with a model for packet injection that is application specific.

## 5 PDG\_GEN IMPLEMENTATION

The PDG\_GEN algorithm uses a two-step approach, which consists of a sampling step followed by the use of an inference heuristic to infer the PDG from the samples generated in the first step. Before describing these steps, we present the formal model of the computing environment assumed by PDG\_GEN. Pseudocode for the algorithms described in this section can be found in [26].

### 5.1 Formal Model

#### 5.1.1 Traces

A *trace* is defined as a time-ordered list of *events*. An event  $E_i$  is a 4-tuple  $\langle T_i, L_i, R_i, P_i \rangle$ , where  $i$  is the entry number in the list,  $T_i$  is the time stamp of the global clock,  $L_i$  is the local node,  $R_i$  is the remote node, and  $P_i$  is the unique packet ID. If  $L_i$  is the sender of the packet and  $R_i$  is the receiver of the packet, then  $E_i$  is a *transmit* event. Each transmit event results in one or more *receive* events. For example, the transmit event,  $E_i = \langle T_i, L_i, R_i, P_i \rangle$  results in a receive event at node  $E_j = \langle T_j, R_i, L_i, P_i \rangle$ , where  $T_j$  is the clock cycle at which the packet is received by node  $R_i$ . Note that  $T_j - T_i$  is the network latency for the packet  $P_i$ .

It is assumed that if an application is run with network configuration A, and then with network configuration B, the two resulting traces will have the same set of packets being sent and received, albeit at different times.<sup>1</sup> More formally, for each event  $E_a = \langle T_a, L_a, R_a, P_a \rangle$  in trace A, there will be a corresponding event  $E_b = \langle T_b, L_b, R_b, P_b \rangle$  in trace B in which  $L_a = L_b$ ,  $R_a = R_b$ , and  $P_a = P_b$ , but  $T_a$  may be different than  $T_b$ . This assumption also implies that the mapping of the application to network nodes is fixed, and that in both run A and run B each node was performing the same set of tasks.

#### 5.1.2 Computation Model

Each node  $n$  in the on-chip network is modeled as a single computational element, which generates packet transmit events and consumes packet receive events. A node transmits the same packets in the same order each time an application is run. Each time a node transmits a packet, it is modeled as the result of some computation that has a set of input values. Inputs could be values previously calculated at the node, or be values received from other nodes. Each transmit event is therefore modeled as having a *computation time* and a set of 0 or more

1. In practice this assumption does not hold and is addressed in Section 5.4.2

receive dependencies, which are receive events that must occur at a node before the computation for the transmit can begin.

Node  $n$  will transmit packet  $p$  once the following 3 requirements have been satisfied: 1)  $n$  has already transmitted the packet that it is supposed to transmit before  $p$ , 2) each packet in  $p$ 's set of receive dependencies has been received by  $n$ , and 3)  $p$ 's computation time has elapsed since both 1) and 2) have been satisfied.

## 5.2 Sample Trace Generation

The PDG\_GEN algorithm takes as input a set of traces of the form described in Section 5.1.1. The traces are generated by running an application of interest multiple times, each time using a different network configuration. Each trace will therefore capture the behavior of the same application under a different set of network conditions.

The first trace (referred to as the *base trace* is generated by running the application of interest on a Fully Connected Network (FCN) with single cycle link latencies. This configuration was chosen because it represents an "ideal" network and will expose computational delays and hence the real dependencies that may otherwise be hidden due to queuing, routing and other non-deterministic mechanisms. Using information from the base trace, the nodes are partitioned into  $m$  sets with pairs of nodes that are communicating the most placed in different sets.

An additional  $m$  traces are then generated by running the application of interest on FCNs in which the outgoing links of the nodes in one of the  $m$  partitions have a large latency,  $p$ , while the remaining links have single cycle latency. If there are  $n$  nodes, then each partition will contain  $\frac{n}{m}$  nodes. The lopsided latencies in the partitioned network configurations serve to expose information about dependencies between packets - some packets will be delayed while waiting for dependencies from slow nodes, while others with no dependencies on packets from slow nodes will not be delayed.

## 5.3 PDG\_GEN Algorithm

The PDG\_GEN algorithm has two tasks: to infer a dependency set and to calculate a computation time for each packet. In general, for a given transmit event  $E_i$ , any event at that node that has occurred before  $E_i$  could be in its dependency set. This is a problem, since there are usually millions of packets in a trace. To deal with this, two simplifying assumptions are made. The first is that transmit events at a given node are ordered (as described in Section 5.1.2). This allows each transmit event  $E_i$  at a node to always be dependent on exactly one previous transmit event at that node, namely the immediately preceding one. While this assumption may not accurately model all real situations, it is important because it simplifies the PDG\_GEN algorithm and also allows for simple and efficient implementation of the code necessary to support PDGs in a network simulator. It should also be noted that traditional trace-based

simulations also make this assumption.

The second simplification is to only consider a window of previous receive events, instead of the entire history. We evaluated two windowing techniques - one that uses a fixed window size  $w$  in which only the  $w$  receive events immediately preceding a transmit event are considered for its dependency set, and another that uses a dynamic window in which a transmit event only considers packets received during the window of time since the  $k$  previous transmits at that node (for example, if  $k=1$ , a transmit can only depend on packets that have arrived since the previous transmit event at the same node, if  $k=2$  a transmit can only depend on packets that have arrived since the second most recent transmit event, etc.).

The PDG\_GEN algorithm uses the base trace and  $m$  additional sample traces (as described in Section 5.2) to generate a receive dependency set ( $S_i$ ) and a computation time ( $D_i$ ) for each packet. PDG\_GEN follows 3 steps:

**STEP 1:** For each transmit event  $E_i$  in each of the traces, add all the receive events within the window ( $k$  or  $w$ ) to  $E_i$ 's set of potential receive dependencies,  $S_i$ .

**STEP 2:** Remove all receive events from  $S_i$  that violate causality, i.e. arrive after the transmit event  $E_i$ , in any of the  $m$  traces.

**STEP 3:** Find the computation time ( $D_i$ ) associated with the transmit event  $E_i$ .

The initial computation delay for each event is computed using the base trace. Let  $E_i = \langle T_i, L_i, R_i, P_i \rangle$  be a transmit event at node  $L_i$ . Recall that this event will occur at cycle  $T_i$ , after all the packets in its reception dependency set have arrived. Let  $T_j$  be the clock cycle at which the last member of the reception dependency set arrives. The initial computation delay  $D_j$  is then calculated as  $T_i - T_j$ . The following two properties are then used:

*Property 1:* If node  $N$  transmits a packet  $P_i$  at time  $T_i$  and if the initial computation delay as computed above is  $D_i$ , then any packet received by node  $N$  at time  $T_j > T_i - D_i$  cannot be a reception dependency for  $P_i$ .

*Property 2:* If node  $N$  transmits packet  $P_i$  at time  $T_i$ , and  $P_i$ 's computation time is  $D_i$ , then any packet received by  $n$  at time  $T_j < T_i - D_i$  cannot be  $P_i$ 's last reception dependency.

These properties are used to prune the set of possible dependencies. The last reception dependency for each packet is found in each of the  $m$  traces, and if it violates Property 1 in any trace, it is removed from the set  $S_i$ . If any of the elements in  $S_i$  violate Property 2, it means that the estimated computation time  $D_i$  was too small, so the corresponding reception dependency is removed from  $S_i$ . This process continues until no pruning occurs for an entire iteration of all the traces.

### 5.3.1 PDG\_GEN Example

Table 2 shows transmission times ( $P_{13}$ ) and reception times ( $P_6, P_7, P_8, P_9$ ) for a node in three different traces. **STEP 1** of the algorithm will result in adding packets  $P_6$ ,

TABLE 2

Trace Fragment - TX denotes transmit event and RX denotes a receive event. For simplicity only the time is shown, the rest of the details of the packet are omitted

	TX	RX	RX	RX	RX
Sample #	$P_{13}$	$P_6$	$P_7$	$P_8$	$P_9$
1	1000	900	950	980	990
2	1050	1020	1000	1030	1100
3	1100	1045	1050	1075	1095

$P_7$ ,  $P_8$ , and  $P_9$  as potential reception dependencies for packet  $P_{13}$ . In **STEP 2** of the algorithm any events that violate causality will be removed - in this example,  $P_9$  will be removed from the reception dependency set since it arrives after the transmission of  $P_{13}$  in trace sample 2.

In **STEP 3**, the computation time is estimated first, which in this case is 20 ( $P_8$  is the last in the set to be received before transmission of  $P_{13}$  in the first trace). Next, Properties 1 and 2 are used to iteratively prune the set of initial dependencies generated in **STEP 1**. At this point Property 1 holds - however, trace sample 3 violates Property 2, because the computation time exhibited in trace sample 3 is 25 ( $P_8$  is received 25 before transmission of  $P_{13}$ ). Therefore,  $P_8$  is removed from the reception dependency set. The third step repeats by calculating the new computation time, now estimated to be 50 ( $P_7$  is received at 950). Property 1 does not hold now, since  $P_6$  is received only 30 before the transmission of  $P_{13}$  in trace sample 2.  $P_6$  is removed from the dependency set, and all Properties hold for the third step for all traces. Thus the PDG\_GEN algorithm will indicate  $P_{13}$  is dependent upon only  $P_7$ , with a computation time of 50.

#### 5.4 Adapting PDG\_GEN to Real Traces

In a modern multicore processor, communication between cores working on a parallel task occurs through reads and writes to memory locations within a shared memory space. This means that on-chip network traffic actually consists of cache coherence protocol messages associated with cache line requests generated by cores performing memory operations. A major implication of this is that no unique packet ID exists to allow for correlation of a packet across the different traces.

##### 5.4.1 Packet Matching

This complication can be addressed by storing additional metadata about each packet in the trace, such as the cache coherence message type and the memory location that the message pertains to. By combining this information with the source and destination of the packet, it is usually possible to find a packet that matches in each of the traces. The situation where it is not possible to find a match is discussed in the following section.

##### 5.4.2 Different Length Traces

Another challenge to dealing with real traces is the fact that the number of packets in each trace may differ. Consider a situation in which core A writes to a memory location (X) two times, and core B writes to that same

memory location (X) once. In one trace, the writes may occur in the order  $A \rightarrow X$ ,  $A \rightarrow X$ ,  $B \rightarrow X$ , while in another trace the order might be  $A \rightarrow X$ ,  $B \rightarrow X$ ,  $A \rightarrow X$ . In the first case traffic will be generated by the first write by A (but not the second, since X will be in A's cache) and by the write by B. In the second case, however, network traffic will be generated for all three memory operations because the write by B will invalidate the cache line in A containing X, and it will no longer be present when A writes a second time. Unfortunately, no perfect solution to this problem exists. If only packets present in all traces are used, then the total number of packets in the final PDG will be lower than in any of the input traces. This is an undesirable result because it means that a smaller volume of traffic will be injected into the network when the PDG is used. For this reason, every packet from the base trace is ensured to be present in the final PDG, so that the overall traffic volume in the PDG matches that of the base trace.

Including unmatched packets from the base trace brings up the issue of how the PDG\_GEN algorithm should handle these packets. There are two separate sub-problems: how to handle an unmatched packet's dependency set and computation time, and how to handle adding it to other packet's dependency sets. When dealing with real traces, it is not unusual for one of the  $m$  sample traces to have significantly different traffic than the rest, causing very few packets to match across every single trace. The PDG\_GEN algorithm therefore allows a packet to generate and prune a dependency set if it matched in a majority of the traces. Similarly, a packet will only be added to other dependency sets if it matched in a majority of the traces. A much more detailed description of this problem and how it is dealt with is available in [26].

##### 5.4.3 Very Large Traces

Benchmarks found in common parallel benchmark suites such as SPLASH-2 [14] or PARSEC [27] typically have regions of interest that involve the execution of many billions of instructions. Even benchmarks with modest amounts of communication can generate network traces consisting of hundreds of millions of packets. For example, each trace from an FFT benchmark with a problem size of 16 million complex data points contains more than 300 million packets, which results in a trace file size of about 28 GB. Running the PDG\_GEN algorithm with a base trace and four sample traces ( $m = 4$ ), at least  $28GB * 5 = 140GB$  of RAM would be required to keep all of the packet information from all of the traces resident in memory. To mitigate this large memory footprint, the PDG\_GEN algorithm can be modified to stream through each trace with a selectable packet window size,  $W$  (not to be confused with  $w$  or  $k$ , the static or dynamic windows of consideration for potential receive dependencies for each packet).

Since every packet from the base trace ends up in the output PDG, the base trace can be read through in a

linear fashion, and the only packets that must be retained in memory are those that are still within the window of consideration for upcoming transmits from each node. The PDG algorithm can be modified to run one packet at a time, because all the necessary information will come from packets that have been encountered and matched earlier in the base trace.<sup>2</sup>

This assumption allows the introduction of a packet window  $W$ , which is the maximum number of packets “ahead of” and “behind” the current position in the base trace that must be kept track of for each of the other traces. Each time a new packet is encountered in the base trace 3 actions occur: 1) if  $W$  packets from the base trace are already being stored, the oldest one is discarded, 2) if  $2W$  packets from each the other traces are already being stored, discard each oldest one, and 3) read forward in each of the other traces to find a new packet.

This allows an upper bound of  $2W*m+W$  packets that must be stored at any time while running the PDG\_GEN algorithm. Using a window size of  $W$  does introduce a trade-off between memory footprint and matching accuracy, because if  $W$  is too small, packets from the base trace may not find their match in some of the sample traces simply because they are outside the window of consideration. Similarly, if the base trace is shorter than any of the sample traces and if  $W$  is small enough, some packets at the end of those traces will never be considered because the packet window never reaches them.

## 5.5 Traffic Specific Modifications

While the PDG\_GEN algorithm as described thus far only uses timing information in its analysis, it is possible to include other traffic-specific information into the process. For example, in the case of our Simics + GEMS simulation setup described earlier in Section 2.2, the traffic is made up of coherence messages for a MOESI directory based cache coherence protocol. Much of the traffic consists of request and response messages sent between cache controllers and directories that have straightforward dependency relationships, which are easily discernable by using information about the message’s source, destination, coherence message type, and cache line in question. This knowledge enabled us to create a modified version of PDG\_GEN specific to the MOESI directory coherence protocol. It uses coherence message types and memory locations to infer all protocol-specific dependencies, and only performs the PDG\_GEN algorithm on initial requests. These modifications were straightforward to implement, and it would be possible to create a modified version of PDG\_GEN for any other type of traffic.

2. Each packet that is encountered in the base trace could theoretically match to a packet that is transmitted at *any* time in each of the other traces - realistically, however, the ordering of packet transmits in each trace should be fairly similar, meaning that a matching packet should usually be located at roughly the same fraction of the way through each trace.

## 6 EVALUATION

Two separate testing environments were employed to evaluate the PDG\_GEN algorithm. The first approach was to generate a *reference* PDG and compare it to a PDG *inferred* by the PDG\_GEN algorithm, which is described in Section 6.1. The second approach was a real-world validation, in which traces captured from benchmarks running in a full system simulator were used by PDG\_GEN to create a PDG for the benchmark. This approach is described in Section 6.2 To see a more detailed evaluation, refer to [26].

### 6.1 Reference PDG Evaluation

In order to evaluate the PDG\_GEN algorithm’s performance in terms of metrics like the number of dependencies found or missed, there must be some known reference PDG to compare to the inferred PDG created by the algorithm. This cannot be done using real applications, because no reference PDG exists for these applications. Not only that, a PDG reflects a simplified *computational model* of what is actually happening as a parallelized application is running, so it is unclear what a correct reference PDG for an application would even be.

Our solution was to generate reference PDGs using synthetic traffic patterns, and run the reference PDGs on a network simulator to create traces that can be used as inputs to PDG\_GEN. The resulting inferred PDG was then compared to the original reference PDG to determine how closely it matches.

#### 6.1.1 Simulation Environment

There are two major components to the simulation environment: a traffic generation tool that creates reference PDGs from synthetic traffic patterns, and a cycle-accurate on-chip network simulator that reads a PDG and injects packets into the network based upon its reception dependencies and computation times.

The traffic generator can create any of the following well known traffic patterns: uniform random (rand), nearest neighbor (nn), tornado (tor), transpose (trans), bit inverse (inv), hotspot (hot), and negative exponential distribution (NED) [28]. It also supports three additional traffic patterns - *Ball*, *Central* and *Tree* for further stress testing. The *Ball* pattern simulates a selectable number of tokens that are randomly sent to the next node based on NED, modeling passing a beach ball in a crowd. The *Central* pattern simulates a central or hotspot node that receives requests and responds to the source node, designed to model a central memory controller or a master node. The *Tree* pattern models a barrier synchronization, whose performance is critical in many parallel applications.

There are also two other traffic generator parameters of note. One is the average packet injection rate, where a rate of 0.1 indicates each node transmits a packet on average once every 10 cycles. The other is the dependency rate, which is the probability that a packet transmitted by node  $n$  is dependent on each previously generated packet received by  $n$ . This probability compounds as

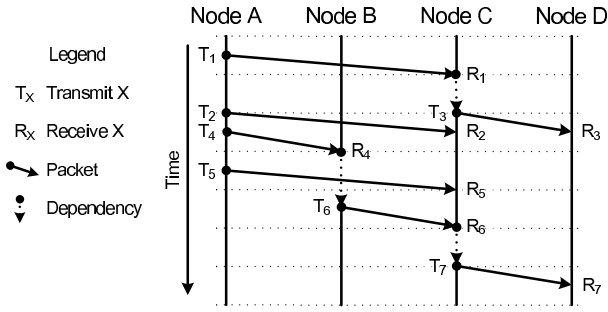


Fig. 3. Space-Time Diagram to Illustrate Quasi-dependencies

each previously generated packet is considered - a dependency rate of 0.5 means that a packet has a 50% chance of being dependent on the most recent receive, and a 25% chance of being dependent on the second most recent, etc.

The output of the traffic generator is a known PDG—a list of packets which each have a source, a destination, a unique Packet ID, a computation time and a list of reception dependencies. We modified the traffic injection code for the popular cycle-accurate network simulator BookSim 2.0 [16] so that it could inject the packets in the PDG at the appropriate time based upon when dependencies are met and when computation times finish for each packet (see [26] for details). However, any network simulator can be similarly modified with relatively small effort.

### 6.1.2 Sensitivity Analysis

In each of the following sections, the inferred PDG is compared to the reference PDG to determine how many dependencies were found, as well as the number of additional *quasi-dependencies* that were added. Quasi-dependencies are defined as packets that are classified as dependencies in the inferred PDG, but are not explicitly stated as dependencies in the reference PDG. Figure 3 illustrates the two types of quasi-dependencies. In this figure packet 7 is actually dependent only on packet 6, but the PDG\_GEN algorithm will also identify packets 2 and 5 as dependencies as well since the partitioning is unlikely to be able to make either packet violate causality. For example, a trace generated with Node A using slow outbound links will also slow delivery of packet 4 and hence delay packet 6, and slowing Node B in a partitioning will further exacerbate the problem. Fortunately, regardless of topology, quasi-dependent packet 2 is highly unlikely to ever be the last dependency met since it is transmitted before packet 4, which the true dependency (packet 6) is itself dependent upon. However, quasi-dependencies of the form of packet 5 are of greater concern, since it is possible that for some topology packet 5 will be received after packet 6 (the true dependency). Fortunately, Property 1 and Property 2 (Section 5.3) used in the PDG\_GEN algorithm are more likely to prune packet 5 from the set of reception dependencies than packet 2, thus reducing the potential

impact on predicted execution time.

### 6.1.3 Number of Partitions, $m$

As described in Section 5.2,  $m$  is the number of partitions into which the nodes should be placed, as well as the number of additional simulations to run and acquire traces from. Figure 4(a) shows the number of true and quasi-dependencies (normalized to the reference PDG) found for values of  $m$  varying from 0 (only base FCN trace) to 64 (base FCN trace plus 64 sample traces each with a slow node). Figure 4(b) shows the average percent error in packet computation time for varying values of  $m$ . The reference PDG that was used consisted of 64 node NED traffic with a dependency rate of 0.5 and an injection rate of 0.01. PDG\_GEN was run with a window size of  $k=1$ , which was able to identify most of the dependencies in the base trace ( $m=0$ ). Figure 4(b) does show that when  $m = 0$  the computation time error is fairly high, but increasing  $m$  beyond 2 yields almost no gain in either dependency identification or computation time accuracy.

These results indicate that the main reason to use a larger value of  $m$  is to reduce the number of quasi-dependencies in the inferred PDG. As later analysis in Section 6.1.6 shows, even with a large number of quasi-dependencies present, the inferred PDG and reference PDG still tend to perform similarly during actual network simulations. Furthermore, when full system simulations are being used to generate traces, increasing  $m$  means increasing the number of computationally expensive full system simulations that are required to generate the PDG. Therefore,  $m$  should be picked according to the details of the situation at hand - how long the initial full system simulations take (and how many can be performed in parallel), how sensitive the network simulator's performance is to tracking extra dependencies, and how many times the PDG will be re-used.

### 6.1.4 Partitioned Link Latency, $p$

Section 5.2 also describes  $p$ , which is the latency of all the outgoing links from the nodes in one "slow" partition. Each sample simulation has a different partition selected as the slow partition. Figures 5 and 6 show the accuracy of PDG\_GEN for varying values of  $p$ . Figures 5(a) and 5(b) show the number of dependencies found for NED and Tree traffic patterns, respectively, while Figures 6(a) and 6(b) show the average percent error in the computation time that PDG\_GEN calculated for each packet. PDG\_GEN was run with  $m=4$  and  $k=1$ , and both NED and Tree consist of 64 node traffic with an injection rate of 0.01 and a dependency rate of 0.5.

For NED, the initial increase of  $p$  from 1 to 2 yields large improvements, because when  $p = 1$  the sample traces are identical to the base trace. Increasing  $p$  beyond 2 yields small improvements in the number of true dependencies found initially, at the cost of small increases in the number of quasi-dependencies found. However, further increases in  $p$  slowly but steadily increases the number of quasi-dependencies found. This is because



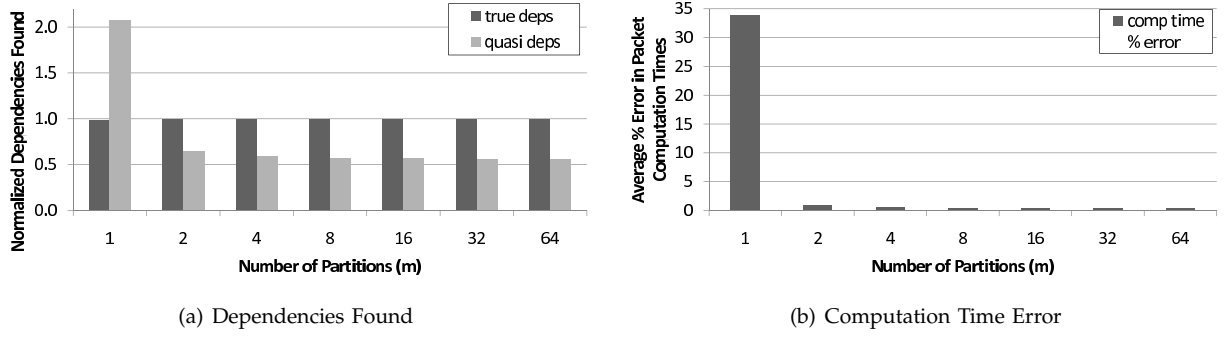


Fig. 4. Normalized number of dependencies found (a) and average percent error in computation time (b) for NED traffic pattern for varying values of  $m$ .

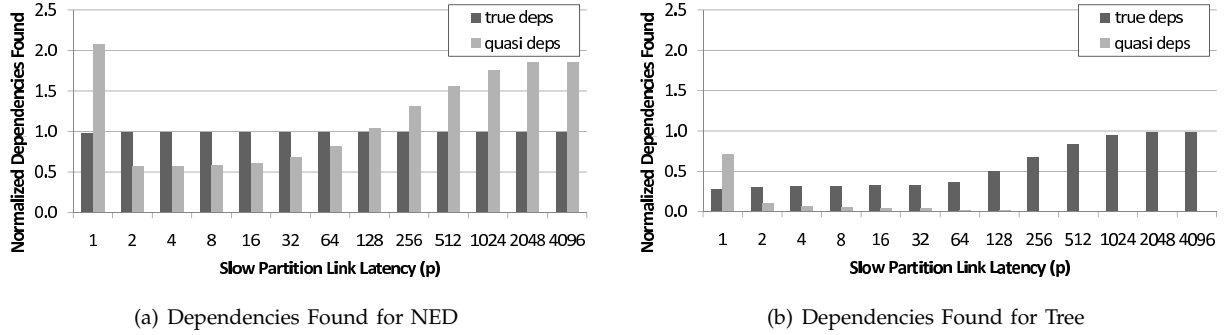


Fig. 5. Normalized number of dependencies found for NED (a) and Tree (b) traffic patterns for varying values of  $p$ .

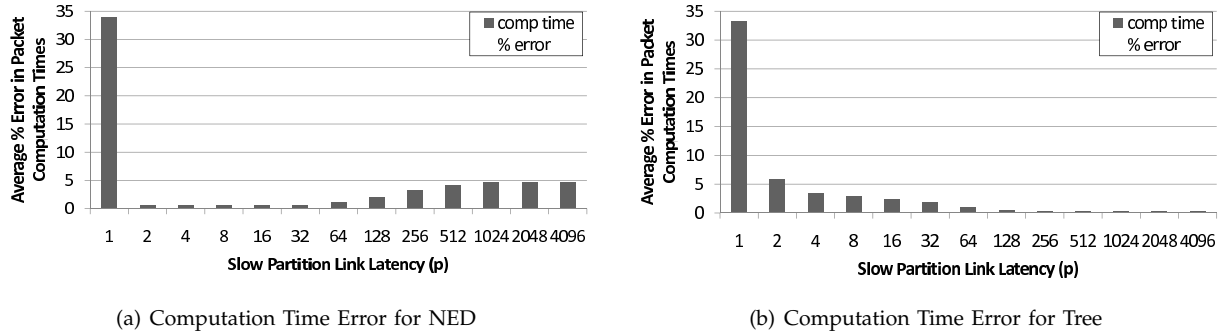


Fig. 6. Average percent error in computation time for NED (a) and Tree (b) traffic patterns for varying values of  $p$ .

some packets will start out with a larger set of initial receive dependencies when  $p$  increases, and any of these packets that cannot be pruned will increase the number of quasi-dependencies found. Note that as the number of quasi-dependencies become very large, the error in computation time begins to grow significantly as well.

The other traffic patterns studied exhibit behavior similar to NED as  $p$  is increased, with one notable exception. For Tree, increasing  $p$  steadily increases the number of true dependencies found, while simultaneously decreasing both quasi-dependencies and computation time errors. This result highlights one of the challenges faced by PDG\_GEN; different traffic patterns may require different parameters, but it is not clear how to determine the correct values for these parameters without a reference PDG to compare against.

### 6.1.5 Window Sizes, $w$ and $k$

As Section 5.3 describes,  $w$  is a static window size containing the number of receive events at a node preceding a transmit event that will be considered as possible dependencies for the transmit, while  $k$  is a dynamic window size containing the number of previous transmit events to go back when considering potential receive dependencies for a transmit. PDG\_GEN can use either a static or dynamic window ( $w$  or  $k$ ).

Figure 7(a) shows the performance of PDG\_GEN on 64 node NED traffic for varying values of  $w$ . The results demonstrate that as the window size grows the number of true dependencies detected increases, but the number of quasi-dependencies climbs even faster. In Figure 7(b), varying values of  $k$  are used, and the results show that using a dynamic window of  $k=1$  tends to perform well due to its adaptive nature.

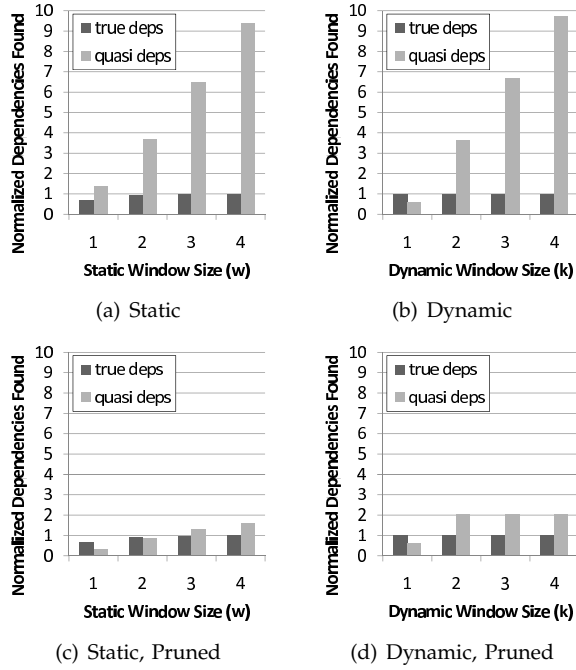


Fig. 7. Effect of window size on accuracy for NED traffic pattern with Static (a) and Dynamic (b) window sizes. (c) and (d) show the effect of pruning all receive dependencies appearing in dependency sets of previous transmits

The reason the number of quasi-dependencies increases so quickly in Figure 7(b) is that as the window size increases, each receive is initially present in more and more transmit's receive dependency sets. For example, for  $k=1$ , each receive begins in one transmit's dependency set. For  $k=2$ , each receive begins in two transmit's dependency sets, and so on. Due to the computational model's constraint that transmits must occur in order, if a transmit is dependent on a receive event, then any subsequent transmits from the same node can have that receive pruned from their dependency set without affecting the PDG's behavior in a simulation. If a large window size is used, then pruning this way can decrease the number of dependencies in the PDG, which can speed up simulation times. Figures 7(c) and 7(d) show the effect of pruning all receive dependencies appearing in dependency sets of previous transmits. The number of quasi-dependencies are greatly reduced without affecting the number of true dependencies, because most of the packets in the reference PDG have at most one packet that depends on them.

As with the sensitivity analysis of  $p$ , the Tree traffic pattern exhibits different behavior than NED when window size is varied. Figures 8(a) and 8(b) show the effects of  $w$  and  $k$  on accuracy for Tree with  $p=10$ , while  $p=100$  in Figures 8(c) and 8(d). The figures show that to find most of the true dependencies for Tree, one would need to use  $p=10$  and  $k=4$ , or  $p=100$  and  $k=3$ . This is very different than the rest of the traffic patterns, which perform much better with  $p=10$  and  $k=1$ .

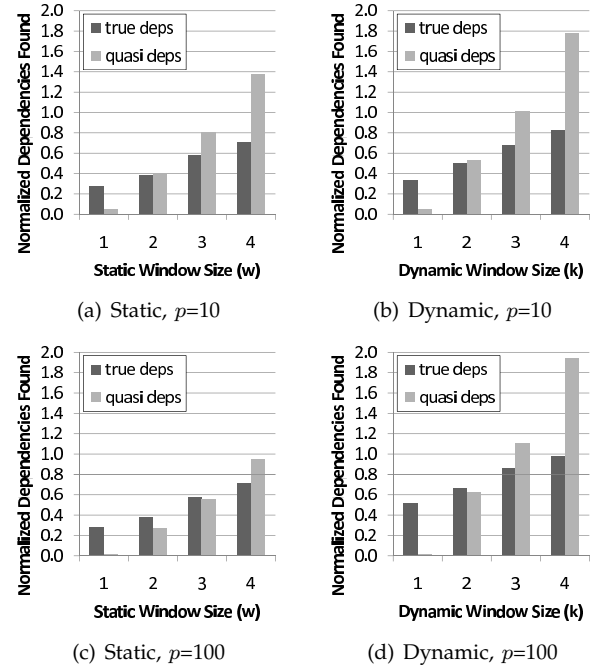


Fig. 8. Effect of window size on accuracy for Tree traffic pattern with Static ((a), (c)) and Dynamic ((b), (d)) window sizes, with  $p=10$  and  $p=100$

Note that if there are a large number of dependencies on packets that tend to arrive well before each transmit, then a larger window size becomes necessary. With the exception of Tree, the traffic generator creates such dependencies with a low probability, which seems to match our observations of real traffic patterns. For instance, cache coherence protocol traffic in a shared memory multiprocessor predominantly has a request-response communication pattern, which will result in most messages being dependant only on a single message that was received very recently.

#### 6.1.6 Performance Comparison

In order to evaluate the performance, the traffic generator was used to produce 64 node reference PDGs with an injection rate of 0.01 and dependency rate of 0.5 for each previously described traffic pattern. Based on the results of the sensitivity analyses, parameter values of  $m = 4$ ,  $k = 1$ , and  $p = 10$  were used by the PDG\_GEN algorithm and an inferred PDG was created for each reference PDG.

Figure 9(a) shows that PDG\_GEN discovers almost all of the true dependencies for most of the traffic patterns, but often also finds a large number of quasi-dependencies. Figure 9(b) shows that PDG\_GEN is fairly accurate in calculating computation time as well (with the exception of *nn* and *tor*).

To compare the overall performance of an inferred and reference PDG, both were run through different network configurations to compare overall performance statistics such as total execution time and average packet latency. The performance of the inferred PDG was also compared to that of a modified reference PDG from which all of

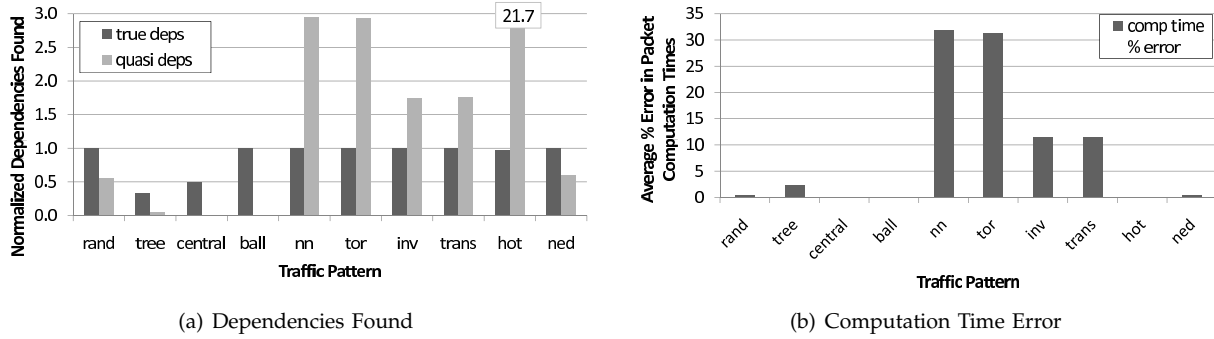


Fig. 9. Normalized number of dependencies found (a) and average percent error in computation time (b) for each traffic pattern

the reception dependencies have been removed (referred to as a *stripped* PDG).

BookSim was used to run a simulation of an 8x8 mesh with two virtual channels, for each reference and inferred PDG. The total execution time and average packet latency of the inferred and stripped PDGs were then normalized to the reference PDG, and the results are shown in Figure 10(a) and Figure 11(a). These figures show that on average the execution times of the inferred PDGs were within 0.55% of the reference PDGs, and average packet latencies were within 0.27%. In contrast, the stripped PDG varied widely from the reference PDG, with average errors of 89.18% in execution time and 27464% in latency. The stripped PDG results show once again that failing to accurately model packet injection rates can lead to incredibly inaccurate conclusions about how a traffic pattern will perform on a network.

The largest discrepancies for the inferred PDG were a 2.25% error in execution time and 1.59% error in latency, both seen in the Tree traffic pattern. This is to be expected, as the previous sections have shown that the Tree traffic pattern performs best with different PDG\_GEN settings than the others. Figures 10(b) and 11(b) show the same PDGs running on a 3 level FatTree network, and the inferred PDG performed very similarly to the mesh case, with an average execution time error of 0.32% and average latency error of 0.30%. These results indicate that an inferred PDG can match a reference PDG very closely in overall network behavior, despite the presence of quasi-dependencies and computation time error.

## 6.2 Simics Evaluation

After performing direct quantitative comparisons between reference and inferred PDGs, the next step was to use the PDG\_GEN algorithm on traces from full system simulations of real applications. As with the initial motivating experiment in Section 2.2, Simics 3.0 [12] and GEMS 2.1.1 [13] were used as the full system simulation framework, and Garnet (the network simulator within GEMS) was modified to record a trace of each packet transmit and receive event that occurs during a full system simulation.

With this modification, multiple Simics simulations of an application of interest can be run to acquire the nec-

essary input traces for use by the PDG\_GEN algorithm.

However, the traces acquired in this manner are not directly usable by PDG\_GEN. As described earlier, there are no explicit packet IDs to identify packets across the traces, and the traces are of different lengths. To accommodate this, the matching algorithm described in Section 5.4.1 was used to generate usable traces. The PDG\_GEN algorithm was also modified to use the windowing scheme described in Section 5.4.3 in order to accommodate the very large traces that are generated by running real applications.

The PDGs obtained from the setup described above can be used by any network simulator that has been modified to support the injection of a PDG file into the network. We modified Garnet to support PDGs, in order to allow a direct comparison between full-system Simics simulations (which use the Garnet simulator) and network-only Garnet simulations using PDGs.

### 6.2.1 Simics Comparison Results

In this section overall performance results are presented for the PDG\_GEN algorithm and each benchmark from the SPLASH-2 parallel benchmark suite. To generate the results Simics and GEMS were configured with 16 directories using the MOESI\_SMP\_directory cache coherency protocol, and 16 1 CPI in-order processors each with 16KB L1 instruction and data caches and private 4MB L2 caches (for a total of 32 distinct nodes in the on-chip network). Pipelined out of order processors were not modeled because we are only looking at the messages out the back side of the memory heirarchy, and any out of order behavior should be absorbed by the various levels of the cache. PDGs were generated for each benchmark using  $m = 4$  and  $p = 50$ , and then simulated on Garnet running in standalone network mode using two new network topologies—a mesh and a fattree, with the network clock speed cut in half and a link pipeline depth of 10 cycles (to simulate a resource-constrained network). The networks had 4 virtual channels and 16 byte flits. These results were then compared to Simics simulations of the same benchmarks on the same two network configurations.

Figure 12(a) shows the execution times and Figure 12(b) shows the average packet latencies for Simics,

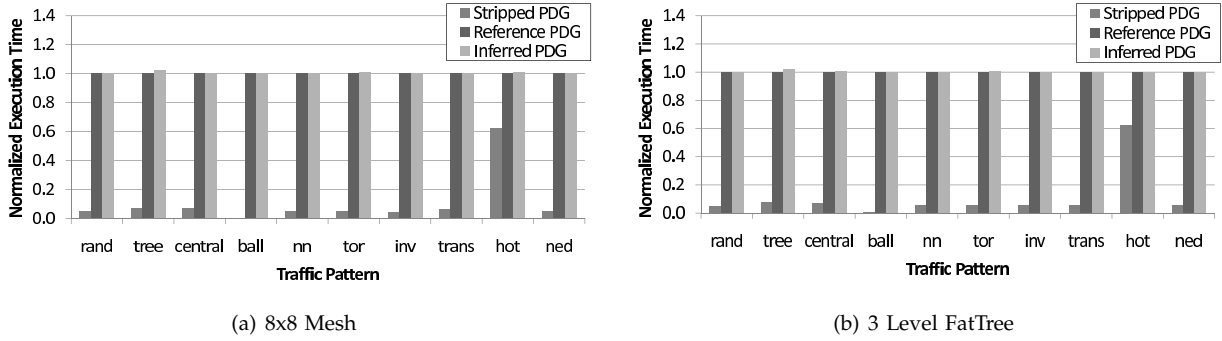


Fig. 10. Normalized execution time for different traffic patterns for Stripped PDG, Reference PDG and Inferred PDG on 8x8 mesh (a) and 3 level FatTree (b) networks

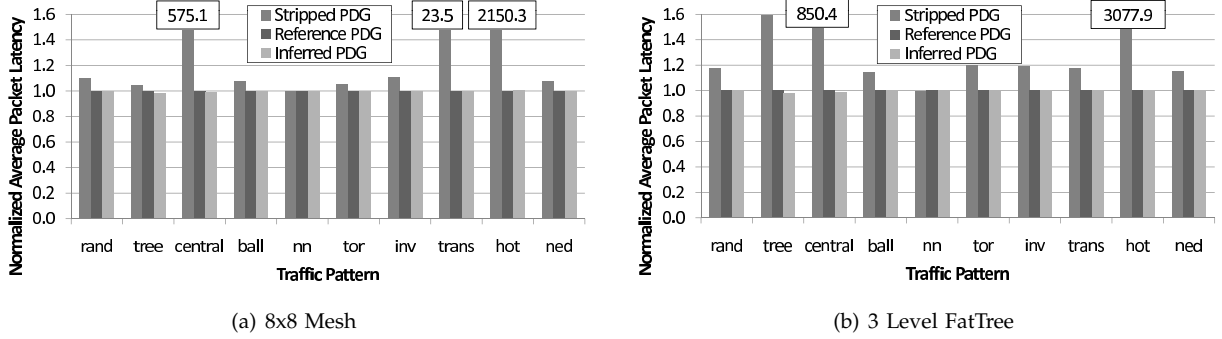


Fig. 11. Normalized latency for different traffic patterns for Stripped PDG, Reference PDG and Inferred PDG on 8x8 mesh (a) and 3 level FatTree (b) networks

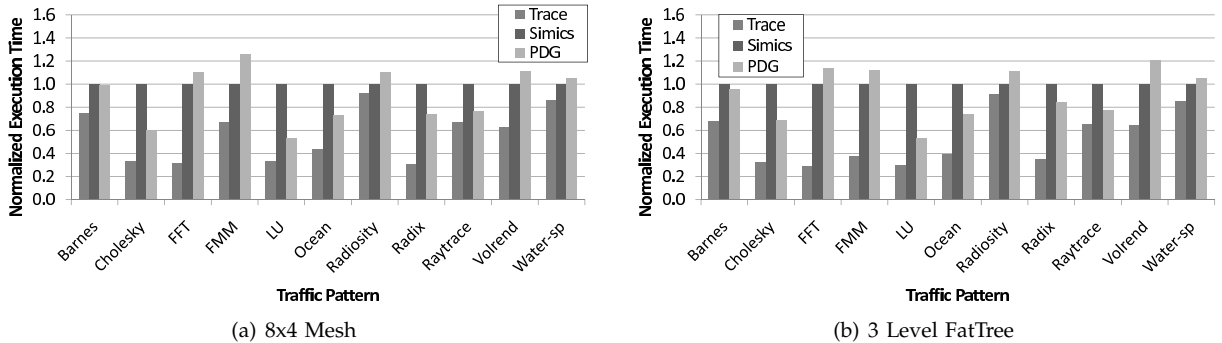


Fig. 12. Normalized execution time for different SPLASH-2 benchmarks for traditional trace, Simics simulation, and PDG on 8x4 mesh (a) and 3 level FatTree (b) networks

PDG, and traditional trace simulations for each benchmark (normalized to the execution time of the Simics simulations) running on an 8x4 mesh network. The base trace was used for the traditional trace simulations. These results show that while the PDGs are not perfect at predicting execution time, they are far superior to the traditional method of using a timestamp-based trace. Notice also that even when the PDGs significantly misspredict execution time, they still result in very accurate packet latency estimates. Both the FFT and FMM benchmarks on fattree show situations where using a traditional trace leads to highly inaccurate packet latency estimates.

Another interesting thing to note is that sometimes the PDG will underestimate execution time due to a

significant increase in the number of packets generated by a benchmark on one of the new networks compared to the base trace. For example, the Cholesky benchmark generated 33% more packets when run on the mesh network than when run on the base FCN network. It is quite possible that this difference in trace size is the true reason for the PDG underestimating execution time (Figure 12(a)), and not that it failed to infer enough dependencies or accurately calculate computation times. And even in these cases the PDG still did a good job of predicting average packet latency (Figure 13(a)).

However, some PDGs did misspredict execution time due to inaccuracies in the PDG\_GEN algorithm. For example, PDG significantly underestimated LU's execution time on fattree, despite only a 15.1% trace size dif-

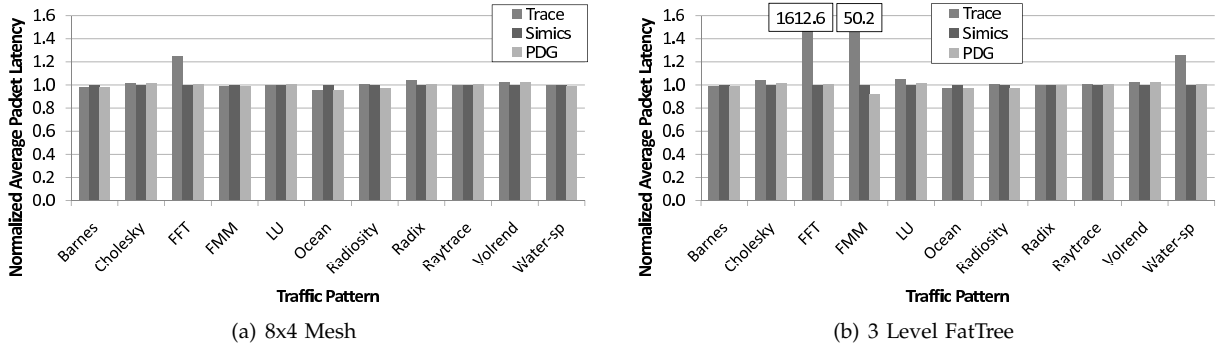


Fig. 13. Normalized latency for different SPLASH-2 benchmarks for traditional trace, Simics simulation, and PDG on 8x4 mesh (a) and 3 level FatTree (b) networks

ference. PDG also significantly overestimated Volrend's execution time on fattree even though there was only a 4.7% trace size difference. This highlights an issue first uncovered by the NED and Tree synthetic traffic patterns in Section 6.1.4: different traffic patterns may require different PDG\_GEN settings.

Overall, the PDGs had an average error of 20.8% in execution time on the mesh network, and an average error of 18.9% in execution time on the fattree network. In contrast, the traditional traces had an average error of 43.4% and 47.3% in execution time on the mesh and fattree networks, respectively. This means that on average, the PDGs were 2.3 times more accurate at predicting execution time. The PDGs were even better at predicting average packet latency, with average errors of 1.66% and 2.16% on the mesh and fattree networks, respectively. The traditional traces had an average error of 3.81% in packet latency on the mesh network, and 15100% on the fattree network (due to the catastrophically large over-estimates for FFT and FMM; see Figure 13(b)). This means that the PDGs were 2.3 times better at predicting average packet latency on the mesh network, and 6990 times better at predicting average packet latency on the fattree network. This shows that as a network becomes more resource constrained, the PDGs become increasingly better than traditional traces at predicting packet latency because they throttle injection rates appropriately and never flood the network with packets the way traditional traces do.

### 6.2.2 Traffic Specific PDGs

In Section 5.5 we discussed the possibility of modifying the PDG\_GEN algorithm to take advantage of traffic-specific behavior. To demonstrate the viability of this approach, we developed a version of the PDG\_GEN algorithm that works specifically with the MOESI directory cache coherence traffic used in these Simics simulations. We used the modified PDG\_GEN algorithm to generate modified PDGs for each SPLASH 2 benchmark, and ran them on the same mesh and fattree networks. Averaged across all of the benchmarks on both of these networks, the modified PDGs were 22% more accurate at predicting execution time and 2% more accurate at predicting packet latency than the original PDGs. While

these gains are modest, the required effort to modify the algorithm was equally modest. Overall, these results show that the PDG\_GEN algorithm can be applied to real-world benchmarks that are commonly employed by researchers today, and afford significant accuracy gains over traditional trace based simulation methods.

## 7 CONCLUSION

The goal of this work was to improve the accuracy of trace based cycle-accurate network simulation, a commonly used on-chip network evaluation method. A full system simulation environment (Simics+GEMS) was used to demonstrate that simply recording a network trace from an application fails to incorporate key information about packet injection rates, and can lead to inaccurate results. The PDG\_GEN algorithm was presented, which infers dependency information between packets based upon a set of traces gathered from multiple full system simulations of an application. Evaluations using both synthetic and real traffic patterns show that PDG\_GEN can increase the accuracy of network simulations significantly compared to the standard technique of simple trace based simulation.

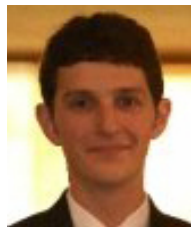
While the results presented here are encouraging, there are several avenues of future work. Different classes of traffic patterns (for example, Tree vs. NED) may call for different settings of the PDG\_GEN algorithm's parameters, or even modifications to the communication model, such as removing total ordering of transmits. Additionally, Kamil et. al [29] have shown that parallel scientific applications often go through program phases in which traffic patterns and volumes change drastically. An initial high-level traffic analysis phase could be added to PDG\_GEN, which identifies traffic characteristics and program phases to determine optimal settings. Work can also be done to apply PDG\_GEN to new communication paradigms, such as on-chip message passing traffic and traffic for emerging architectures such as System-on-Chip (SoC) or General Purpose Graphics Processing Units (GPGPU).

## ACKNOWLEDGMENTS

This research is partially supported by NSF Award # CCF-1116897.

## REFERENCES

- [1] C. H. van Berkel, "Multi-core for mobile phones," in *DATE*, 2009, pp. 1260–1265.
- [2] D. Zhao *et al.*, "Design of multi-channel wireless noc to improve on-chip communication capacity," in *Networks-on-Chip (NOCS)*, 2011 *Fifth ACM/IEEE International Symposium on*, May 2011, pp. 177–184.
- [3] D. Vantrease *et al.*, "Corona: System implications of emerging nanophotonic technology," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 153–164.
- [4] G. Hendry *et al.*, "Analysis of photonic networks for a chip multiprocessor using scientific applications," *Networks-on-Chip, International Symposium on*, vol. 0, pp. 104–113, 2009.
- [5] Y. Pan *et al.*, "Firefly: illuminating future network-on-chip with nanophotonics," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 429–440, 2009.
- [6] M. J. Cianchetti *et al.*, "Phastlane: a rapid transit optical routing network," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 441–450, 2009.
- [7] N. Easley *et al.*, "In-network cache coherence," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 321–332.
- [8] J. Kim *et al.*, "A novel dimensionally-decomposed router for on-chip communication in 3d architectures," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 138–149, 2007.
- [9] D. Park *et al.*, "Design of a dynamic priority-based fast path architecture for on-chip interconnects," in *HOTI '07: Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 15–20.
- [10] N. E. Jerger *et al.*, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 229–240, 2008.
- [11] J. Kim *et al.*, "Flattened butterfly topology for on-chip networks," *Microarchitecture, IEEE/ACM International Symposium on*, vol. 0, pp. 172–182, 2007.
- [12] P. Magnusson *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.
- [13] M. M. K. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.
- [14] S. C. Woo *et al.*, "The splash-2 programs: characterization and methodological considerations," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1995, pp. 24–36.
- [15] L.-S. Peh *et al.*, "Garnet: A detailed on-chip network model inside a full-system simulator," *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 0, April 2009.
- [16] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco: Morgan Kaufmann, 2004.
- [17] Z. Tan *et al.*, "A case for fame: Fpga architecture model execution," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 290–301.
- [18] J. Miller *et al.*, "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA)*, 2010 *IEEE 16th International Symposium on*, 9–14 2010, pp. 1–12.
- [19] J. Hestness *et al.*, "Netrace: dependency-driven trace-based network-on-chip simulation," in *Proceedings of the Third International Workshop on Network on Chip Architectures*, ser. NoCArc '10. New York, NY, USA: ACM, 2010, pp. 31–36.
- [20] R. Marculescu *et al.*, "Outstanding research problems in noc design: system, microarchitecture, and circuit perspectives," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 3–21, 2009.
- [21] G. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for mpeg-2 video applications," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 1, pp. 108–119, jan. 2004.
- [22] V. Soteriou *et al.*, "A statistical traffic model for on-chip interconnection networks," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2006. *MASCOTS 2006. 14th IEEE International Symposium on*, sept. 2006, pp. 104–116.
- [23] H. Kim, *et al.*, "On-chip network evaluation framework," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010.
- [24] G. Wei, *et al.*, "A software framework for trace analysis targeting multicore platforms design," in *Networks on Chip (NoCS)*, 2011 *Fifth IEEE/ACM International Symposium on*, may 2011, pp. 259–260.
- [25] P. V. Gratz and S. W. Keckler, "Realistic workload characterization and analysis for networks-on-chip design," *CMP-MSI*, 2010.
- [26] K. Macdonald, "Inferring packet dependencies to improve trace-based simulation of on-chip networks," Master's thesis, University of California, Davis, 2011.
- [27] C. Bienia *et al.*, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.
- [28] A.-M. Rahmani *et al.*, "Negative exponential distribution traffic pattern for power/performance analysis of network on chips," in *VLSI '09: Proceedings of the 2009 22nd International Conference on VLSI Design*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 157–162.
- [29] S. Kamil *et al.*, "Reconfigurable hybrid interconnection for static and dynamic scientific applications," in *Proceedings of the 4th international conference on Computing frontiers*, ser. CF '07. New York, NY, USA: ACM, 2007, pp. 183–194.



**Kevin Macdonald** received his Master's Degree in Computer Science from University of California, Davis. His research interests include computer architecture and on-chip networks.



**Christopher Nitta** is a postdoctoral researcher and lecturer at University of California, Davis. His research interests include network-on-chip technologies, embedded system and RTOS design, and hybrid electric vehicle control. He has a PhD in Computer Science from University of California, Davis.



**Matthew Farrens** received his PhD in Electrical Engineering from the University of Wisconsin and is a professor of Computer Science at the University of California, Davis. His research interests center on computer architecture, with special emphasis on the memory hierarchy. He is a member of ACM and IEEE and a recipient of the NSF PYI award.



**Venkatesh Akella** received his PhD in Computer Science from University of Utah and is a professor of Electrical & Computer Engineering at University of California, Davis. His current research encompasses various aspects of embedded systems and computer architecture with special emphasis on embedded software, hardware/software codesign and low power system design. He is member of ACM and received the NSF CAREER award.