Inferring Packet Dependencies to Improve Trace Based Simulation of On-Chip Networks

Christopher Nitta
cjnitta@ucdavis.edu
University of California, Davis
Davis, CA 95616

Kevin Macdonald
klmacdonald@ucdavis.edu
University of California, Davis
Davis, CA 95616

Matthew Farrens
farrens@cs.ucdavis.edu
University of California, Davis
Davis, CA 95616

Venkatesh Akella
akella@ucdavis.edu
University of California, Davis
Davis, CA 95616

ABSTRACT

With the advent of large scale chip-level multiprocessors, there is a growing interest in the design and analysis of on-chip networks. The use of 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. In order 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 messages (packets), trace based simulation can lead one to draw incorrect conclusions about network performance metrics such as latency and overall execution time. In this paper we will demonstrate the importance of including dependency information in traces, as well as present an inference-based technique for identifying and including dependencies, and show that using these augmented traces results in much better simulation accuracy without excessively extending simulation time.

Categories and Subject Descriptors

C.2.m [**Computer-Communication Networks**]: Miscellaneous; I.6.5 [**Simulation And Modeling**]: Model Development—*Modeling methodologies*

General Terms

Design, Performance

1. INTRODUCTION

There is a general consensus in the architecture community that the best way (in terms of performance per watt) to harness the benefits of Moore's law is through *parallelism*. Consequently, from servers to mobile phones [1], future chips will contain dozens (if not hundreds) 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 is not appropriate for designing on-chip networks - a thorough exploration of the design space is required. For example, the amount of heterogeneity, buffer sizes, number of virtual channels, topology of the network, arbitration and flow control schemes 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. In order 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 researchers from doing full system simulations with a large number of nodes. In our review of the current literature we found that only one study featured full-system simulations with as many as 32 nodes [2], one used 24 nodes [3], and the majority restricted their simulations to 16 nodes or less [4–8].

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 simulator in order to evaluate various network configurations. Trace based simulations run much faster than full system simulations, and are widely used [9–17]. 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 we will refer to them 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 - to look at different topologies, arbitration schemes, flow control mechanisms, buffer sizes, etc. 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. Thus, drawing any meaningful conclusion about system parameters (such as overall speedup or ideal buffer size) based on these results is exceedingly difficult, if not impossible.[1]

Unfortunately, many studies that use dependency-free traces include results relating to network speedup [9, 10, 12], normalized execution time [11], or network latency [13–16]. In this paper we will show that one has to be very careful when making predictions about the performance of on-chip networks based on the results of trace-driven simulations.

In order to increase the accuracy of trace-based simulation we have developed a technique that allows us to add dependency information to traces, which we accomplish by inferring dependency information based on a series of full-system runs using different latencies. Using these augmented traces allows one to gain further insight into the true behavior of on-chip networks, provides more accurate results, and can help guide real on-chip network system design.

The rest of the paper is organized as follows. In Section 2 we provide motivation for this work by providing a simple demonstration of the disparities that can occur between full system simulations and trace based simulations. We discuss related works from the literature in Section 3. In Section 4 we present a formal model for representing dependency information in a trace, and describe our algorithm for automated dependency inference, called PDG_GEN. Section 5 presents the validation methodology and experimental results.

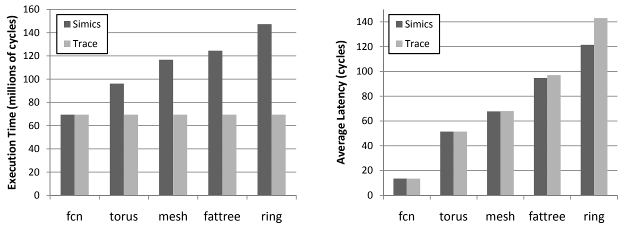
2. MOTIVATION

We will illustrate the pitfalls of using trace-based-simulation without packet dependency information by comparing the full-system simulation results (which is the true indicator of performance) with the output of a trace-based simulation, where the traces are generated from a network topology different from the one being simulated. We used Simics 3.0 [18] and GEMS 2.1.1 [19] for the experiment.

First we modified Garnet [20] (which is the network simulator inside GEMS) to generate a trace consisting of all messages that enter and exit the network. We then configured Simics to model a 16 core processor with a fully connected single cycle network, and ran a 1 million point FFT benchmark. Once we had the trace generated by the Simics FFT simulation, we used Garnet to run a network-only simulation in which each message was injected into the network at the timestamp specified in the trace. The results of this simulation matched the Simics results, which is what we would expect; given the same network configuration and the same messages injected at the same times, the network's behavior should match the actual Simics simulation. We then used the same trace to re-run the network-only simulation on three different topologies: a torus, a mesh, and a ring. The results of these trace-based simulations are shown as the lighter bars in Figure 1.

As Figure 1(a) shows, the execution times for the trace-based simulations hardly change for different topologies, because each message is always injected into the network at a fixed time. Figure 1(b) shows that network latency is not affected until the trace is run on a network with a low enough bandwidth that congestion begins to occur. Basically, a trace by itself represents the packet injection distribution for the specific network configuration on which the trace was collected. When it is used on a different network configuration, it no longer represents the actual packet injection distribution for the application and hence could yield erroneous results. If the trace had knowledge of the inherent packet dependencies in the application, then the packet injection would be closer to what actually happens on a given network configuration, and the simulation results would be more meaningful and reliable.

[1]Simulating a trace taken from a slower network on a higher performing network is also a problem, because it may not show the simulated networks true potential since packets are injected at a lower rate than they would be in a real system.



(a) Execution Time (b) Average Network Latency

Figure 1: 1M point FFT execution time (a) and average network latency (b) for Simics full system simulation and trace-based network simulation

3. 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 [21] 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 [20] 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 above.

Simics [18] is a commercial tool (recently acquired by Intel) that allows full-system functional simulation. However, as the benchmarking data in [22] shows, it is very slow and does not scale beyond approximately 16 cores. 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 [19] 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 [23] 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 [30] recognize the necessity for dependency information within traces, and propose a technique for inferring dependencies based upon the ordering of memory references from a single full system simulation.

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 [24]. Marculescu [25] was the first to propose a mathematical characterization of node to node traffic for the MPEG-2 application. Soteriou et. al [26] generalized this to a comprehensive network traffic model based on hop-count, burstiness and packet injection distributions. Gratz and Keckler [27] 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 neglecting 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.

4. MODELS AND ALGORITHM

There are two ways of capturing packet dependencies - a top-down approach which *instruments* the source code of the application to record dependency information during a full-system run, or a bottom-up approach which *infers* the dependencies. We feel the latter is the better approach for a number of reasons. The source code of applications is frequently unavailable, for example, especially if a system has hard IP blocks. In addition, a detailed knowledge of the source code is required in order to instrument it correctly. An inference based approach is more practical, although it will not be quite as accurate as the top-down approach.

Problem Formulation

Our goal is to generate a packet dependency graph (PDG) for an application. To do this, we assume a computing system with N nodes (processors), and that the application has already been partitioned to fit on the N nodes. The generation of the PDG is a onetime activity for a given application and partition, and these PDGs will be used to design and optimize on-chip networks just like SPEC, PARSEC, and EEMBC benchmarks are used today to explore processor and memory configurations. Thus, the *generation* of the PDG will require an investment of time and computing resources equal to that required by full-system simulation, but the *use* of the PDGs will be on trace-based simulators, and will therefore be much less resource-intensive.

We use 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. Sampling involves running a number (m) of full system simulations, where $m \geq 1$.

There are several important questions that must be addressed - for example, what network topology and parameters/conditions are used to generate the samples? How many samples are needed? How does one validate that the inferred PDG is correct? We start by describing a formal model for a PDG, then describe the heuristics we use with an example. In the next section, detailed results and metrics are discussed.

Abstract Model

A *trace* is defined as a time-ordered list of *events*. An event E_i is a 4-tuple $< T_i, L_i, R_i, P_i >$, 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, $E_i =< T_i, L_i, R_i, P_i >$ results in a receive event at node $E_j =< T_j, R_i, L_i, P_i >$, 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. Each transmit event E_i can be modeled as a complex gate C [2] with $t \geq 1$ inputs. This set of t inputs is defined as the *dependency set* for E_i, which implies event E_i happens after **all** the t inputs have arrived, and some delay D_i has elapsed. (The dependency set is primarily the set of reception dependencies, although a transmit can depend upon the previous transmit or the start of the simulation.)

A transmit event is modeled by the firing of the complex gate C. The intrinsic delay D_i is used to model the computation or processing delay required before producing the data being transmitted. We will exploit this property when developing the algorithm PDG_GEN (described below) to infer dependencies. A PDG which models the on-chip network is an interconnection of these gates. Network latency is modeled as the propagation (or wiring) delay between the output of one gate and the input of another gate.

Transmit event E_i in node N_i generally depends on "some" transmit events in other nodes N_j, where $j \neq i$. These events form the reception *dependency set* for the event E_i. Thus the task of inferring the PDG boils down to determining the reception dependency set, and an estimate of the intrinsic delay D_i, for each transmit event E_i in the trace.

To determine the reception dependency set for each event, we first need to ensure that causality is not violated, since a given transmit event cannot depend on a transmit event that is going to happen in the future (in other words, E_i cannot depend on any event E_j where $T_j \geq T_i$.) Hence, we need at least one trace that is generated on a network that exposes causality of events. We can accomplish this by modeling it as a network of gates with zero wire delay. Such a network is called speed-independent in asynchronous logic literature. A speed-independent network can be emulated using a fully connected topology with single cycle latency and infinite receive buffers, so there is no delay due to routing, arbitration, switching, or flow control. Thus any delay observed can be attributed solely to the computation or processing delay. We use a trace generated on such a network as the initial *base trace*.

PDG_GEN Algorithm

In general, for a given transmit event E_i, *any* event at that node that has occurred before T_i could be in its dependency set. This is a problem, since there are usually millions of packets in a trace. In order to deal with this we make two simplifying assumptions - that transmit events in a given node are *ordered*, and that a given transmit can only depend on the packets received in a window of time since the k previous transmits at that particular node. For example, if k=1, it means that 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 transmit event before the most recent one.

In addition to the base trace, m other traces are created by partitioning the nodes into m sets such that pairs of nodes with the most communication between them are put in different sets. The m additional traces are generated using networks with skewed link latencies, such that the outgoing links of the nodes in one of the m partitions has high latency, while the remaining links have a latency of one.

The highly skewed latencies in the partitioned network configurations serve to expose the dependencies between packets, since some packets will be delayed while waiting for dependent packets arriving on slower links, while others with no dependencies on packets from slower links will not. The PDG_GEN algorithm has 3 steps:

[2]The analogy with a Muller C element in asynchronous logic is intentional as it helps us define parameters such as network latency more intuitively and accurately

Table 1: 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 #	T_{13}	T_6	T_7	T_8	T_9
1	1000	900	950	980	990
2	1050	1020	1000	1030	1100
3	1100	1045	1050	1075	1095

STEP 1: For each transmit event E_i, add all the receive events within the window of W to the 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 computation time associated with a transmit event is calculated as follows: The initial computation delay for each event is computed using the base trace. Let $E_i =< T_i, L_i, R_i, P_i >$ be a transmit event in node (or processor) L_i. Recall that this event will happen in 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 $T_i - T_j$. We define two properties:

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 as follows. 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.

Note that the choice of m is up to the developer of the PDGs. A small value of m will require a lower upfront investment in terms of simulation time, because $m+1$ is the number of full system simulations that are needed. However, it will also lead to less accurate results, since it might not be possible to expose many packet dependencies.

PDG_GEN Example

The following is an example of how we determine the computational delay and reception dependency set for a given packet. Table 1 gives the transmission and reception times for a packet (P_{13}) in three different traces. **STEP 1** of the algorithm would result in adding events E_6, E_7, E_8, and E_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, E_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 computational delay is estimated first, which in this case is 20 (E_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 computational delay exhibited in trace sample 3 is 25 (E_8 is received 25 before transmission of P_{13}). Therefore, E_8 is removed from the reception dependency set. The third step repeats by calculating the new computational delay, now estimated to be 50 (E_7 is received at 950). Property 1 does not hold now, since event E_6 occurs within the window of the transmission of P_{13} and 50 before that in trace sample 2. E_6 is removed from the dependency set, and at this point all Properties hold for the third step for all traces. Thus the PDG_GEN algorithm will indicate P_{13} is dependent upon only E_7, with a computational delay of 50.

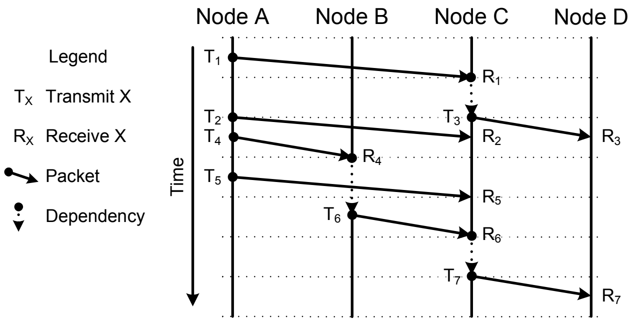


Figure 2: Space-Time Diagram to Illustrate Quasi-dependencies

5. METHODOLOGY AND RESULTS

5.1 Validation

In the previous section, we described how a PDG is generated for a given application mapped to N processors. In this section we describe a methodology to validate the proposed approach.

We validate our approach by comparing a reference *PDG* to the PDG *inferred* by the PDG_GEN heuristic. Since reference PDGs do not exist for benchmarks such as SPLASH or PARSEC, we created our own set of reference PDGs using a traffic generator which takes a wide range of user selectable parameters as inputs, and generates traffic based on well-known synthetic traffic models for on-chip networks. Next we created m trace samples by simulating the *PDG* on a modified version of Garnet that tracks dependencies, and fed them into PDG_GEN to create PDG'. We then compared PDG' to *PDG* to see how many dependencies were missed, as well as the number of additional "quasi-dependencies" that were added (we define quasi-dependencies as packets that are classified as dependencies in PDG', but are not explicitly stated as dependencies in *PDG*.) We also compared the execution time for *PDG* and PDG', as well as the average network latency predicted from PDG' to the average network latency in *PDG*, when the two PDGs were simulated on the modified Garnet.

Figure 2 illustrates the two types of quasi-dependencies. In this figure packet 7 is actually dependent on packet 6, while packets 2 and 5 are quasi-dependencies. The PDG_GEN algorithm will identify packets 2 and 5 as dependencies 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, packet 2 is highly unlikely to ever be the last dependency met since it is sent before packet 4. 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

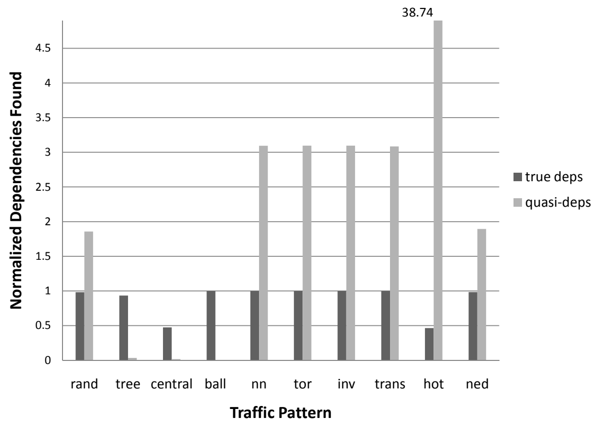
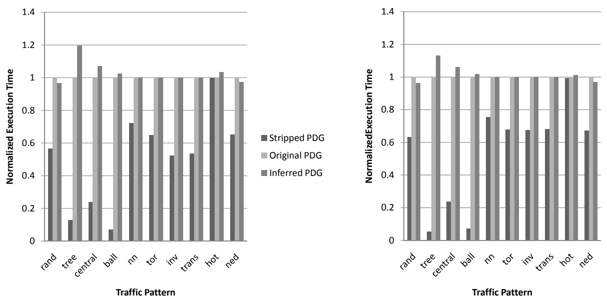
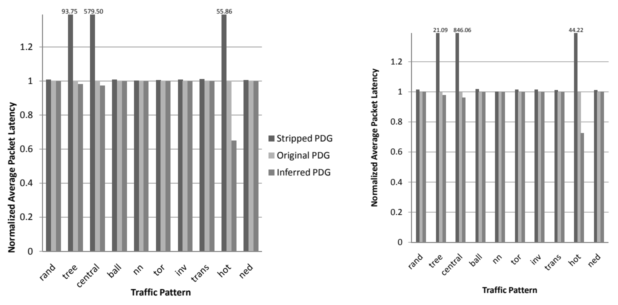


Figure 3: Comparison of true and quasi-dependencies for the various traffic patterns (normalized to number of depencies present in the reference PDGs)



(a) 8x8 Mesh (b) 3 Level FatTree

Figure 4: 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



(a) 8x8 Mesh (b) 3 Level FatTree

Figure 5: Normalized Latency for different traffic patterns for Stripped PDG, Reference PDG and Inferred PDG on 8x8 mesh (a) and 3 level FatTree (b) networks

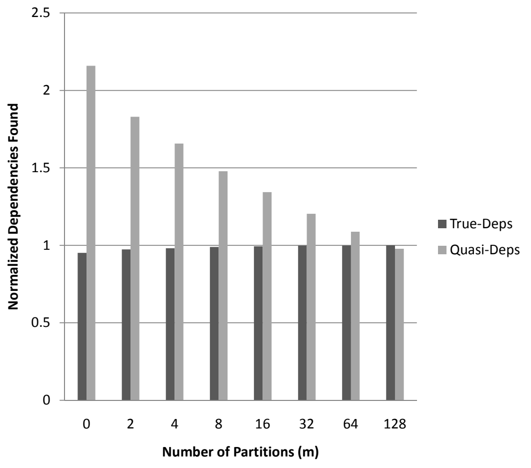


Figure 6: Accuracy vs. Number of Partitions

received after packet 6 (the true dependency). Property 1 and Property 2 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.

The reference PDGs were created for the following traffic patterns - uniform random (rand), nearest neighbor (nn), tornado (tor), transpose (trans), bit inverse (inv), hotspot (hot), and NED [28]. We also used 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.

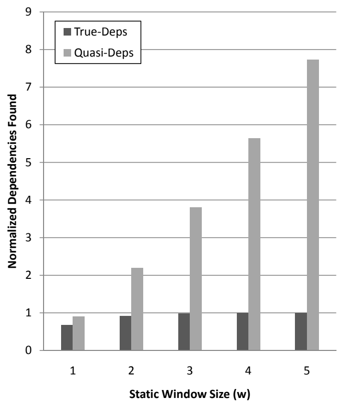
Figure 3 shows that PDG_GEN discovers almost all of the true dependencies for most of the traffic patterns. With the exception of *ball*, PDG_GEN also finds a large number of quasi-dependencies, but we will next show that they have little impact on execution time and network latency.

Execution time and average network latency are other important metrics to evaluate the accuracy of the inferred PDGs. To perform this analysis, we created reference PDGs of 1M packets for a 64 node system. We ran our algorithm with *k*=1 and *m*=4 to generate the inferred PDGs. The reference and inferred PDGs were then simulated on an 8x8 mesh with two virtual channels. We compared the total execution time (shown in Figure 4(a)) and average packet latency (shown in 5(a)) of the inferred PDG and the reference PDG. We also compared the inferred PDG performance to that of traces with all dependencies removed, which are referred to as Stripped PDGs. In the figures, the execution times and average packet latencies are normalized to the original PDG, and they show that on average the execution times of the inferred PDGs were within 3% of the reference PDGs.

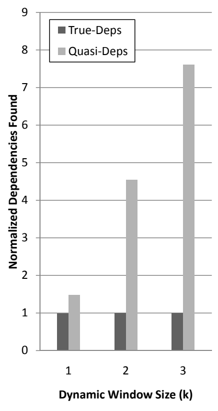
Figures 4(b) and 5(b) show the same PDGs running on a 3 level FatTree network. The average difference of execution time between the inferred and reference PDGs in this case was 1.5%, while the execution time of the stripped PDGs varied widely between different traffic patterns (for both the mesh and the FatTree) - a further demonstration of the importance of augmenting the traces with dependency information. The largest discrepancy in execution time for the inferred PDG was 13.1%, seen in the *tree* traffic pattern. This may seem large, but it pales in comparison to the Stripped PDG, which exhibited a 94.8% difference.

5.2 Sensitivity Analysis

Figure 6 shows the performance of PDG_GEN for different values of *m* (the number of sample traces used to generate the inferred



(a) Static



(b) Dynamic

Figure 7: Effect of Window Size on Accuracy for NED traffic pattern on 128 nodes with Static ((a)) and Dynamic ((b)) Window Sizes. Data is normalized to the number of true dependencies in the reference PDG for the NED traffic pattern

PDG.) Notice that even for relatively small values of *m*, there are very few missed true dependencies. The biggest impact of using a larger value of *m* is that it lowers the number of quasi-dependencies detected, and the quasi-dependencies do not appear to have a significant impact on execution time or average latency. This is encouraging, because it means that a small number of full system simulations will be necessary to generate the PDGs, even for a large number of nodes. This ensures that the proposed approach is scalable, which is one of the goals of the work.

Figure 7(a) shows the performance of PDG_GEN on a system with 128 nodes using a NED traffic pattern and *m*=8. In this figure only *w* previous packets are examined when inferring dependencies. The results show that as the window size grows the number of true dependencies detected increases, but the number of quasi-dependencies increases even faster. Figure 7(b) shows the results for the same setup using various dynamic window sizes, where all received packets since the *k* previous transmits are considered. The results are normalized to the number of true dependencies in the reference PDG, and they show that using a dynamic window of *k*=1 tends to perform well due to its adaptive nature. The figure also shows that increasing *k* does not dramatically improve the detection of true dependencies, but does dramatically increase the number of quasi-dependencies detected.

5.3 PDG Results for Shared Memory Benchmarks

By using the simulation setup described in Section 2, we were able to acquire traces from real shared memory benchmarks running on Simics for use with the PDG algorithm. We ran Simics simulations in a 16 core configuration for FFT, LU, and CHOLESKY from the SPLASH 2 benchmark suite (other benchmarks from the suite were omitted due to lack of adequate simulation resources). We chose to use *m*=4 and *k*=1, which Figures 6 and 7 indicate will yield a good tradeoff between accuracy and required simulation overhead.

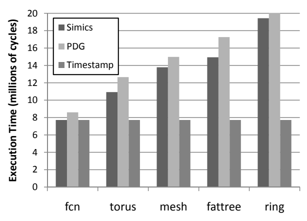
The traces obtained from the Simics simulations cannot be directly used by the PDG_GEN algorithm because the packets (or messages) in each of the *m* runs of the simulator are not identical. This is due to the fact that the traffic in shared memory systems consists of coherence messages generated by cache misses, and conflicts resulting from each processor making requests to read/write memory locations in the shared address space. Each trace will be of a different length due to a slightly different sequence of coherence events occurring when a given processor-to-directory latency is higher or lower than it was originally.

We addressed this issue by developing a matching algorithm that correlates the messages that correspond to each other across all traces. Most of the matching can be done by using only the message contents (source, destination, memory address reference, coherence message type, etc.). Any messages that are not matched across all the traces cannot remain in the trace as-is, because the PDG_GEN algorithm requires each trace to consist of identical messages. However, if only the successfully matched messages are used, the overall traffic volume will be lower, thus changing the trace's impact on the network it is injected into. In order to keep the volume of traffic consistent, we "fill out" the traces with messages that are in the base trace but unmatched, by inserting the messages into the traces that are missing them. The base trace was chosen as the comparison since it represents the theoretical ideal (the base trace should be closest to the true data dependencies).

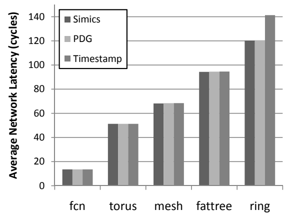
After running the matching algorithm on the Simics traces, we ran the PDG_GEN algorithm on them and generated PDG traces for each benchmark. The PDG traces corresponding to each benchmark were simulated on a modified version of Garnet, using various network topologies; these results were then compared with the results of Simics simulations on the same topologies. Figure 8 shows the execution times and average network latencies for Simics, PDG, and basic timestamp simulations for a 64K point FFT on various topologies. Figures 9 and 10 show results for 256x256 LU and CHOLESKY with TK14.O input, respectively. The PDG trace averaged 11.4%, 23.4%, and 5.2% error in execution time and 0.16%, 0.34%, and 1.3% error in average network latency for FFT, LU, and CHOLESKY, respectively. On average across all the benchmarks, the PDG was 2.5 times more accurate than the basic timestamp for execution time and 5.7 times more accurate for latency. The PDG was the least accurate on the LU benchmark, a problem that most likely could be remedied through increasing the number of samples (*m*) used by PDG_GEN. These results show that the PDG_GEN algorithm can be applied to real world benchmarks that are commonly employed by researchers today.

5.4 Discussion

Several questions may have come to the reader's mind regarding this approach. For example, isn't running *m* full system simulations increasing the amount of work? Is a gate level model necessary? Are there other approaches to solving this problem, and if so, why was this one chosen? Can taint analysis techniques from the security field be used to infer packet dependencies? Is the static

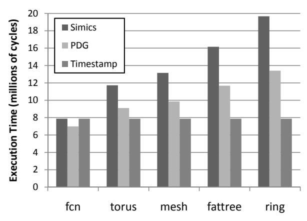


(a) Execution Time

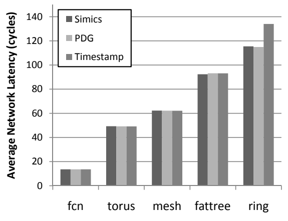


(b) Average Network Latency

Figure 8: 64K point FFT execution time (a) and Average Network Latency (b) for Simics full system simulation and PDG trace-based network simulation

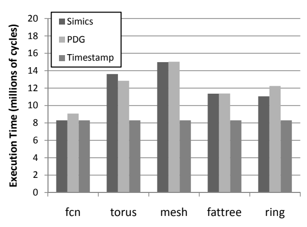


(a) Execution Time

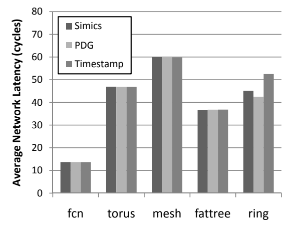


(b) Average Network Latency

Figure 9: 256x256 LU execution time (a) and Average Network Latency (b) for Simics full system simulation and PDG trace-based network simulation



(a) Execution Time



(b) Average Network Latency

Figure 10: TK14.O CHOLESKY execution time (a) and Average Network Latency (b) for Simics full system simulation and PDG trace-based network simulation. (Note: Simics simulation errors forced us to use lower link latencies for fattree and ring for the CHOLESKY experiments)

application partitioning in the full system simulation a limitation due to this approach? Why does the heuristic perform poorly on the hotspot traffic based reference PDG? Here we will attempt to address these questions.

It is likely that the *m* full system simulations will require a substantial amount of work on useful (long) benchmarks, but the simulations only have to be performed once - the inferred PDG derived from the full system runs will be used for future NoC exploration. In addition, the required full system simulations will be run on fully connected networks, and therefore will execute somewhat faster than more complex multi-hop networks.

There are other approaches to developing the PDG from full system simulations. While modifying the full system simulator to produce the packet dependencies may seem to be the most intuitive approach, there are many pitfalls that would need to be addressed. Tracking dependencies within an application would require not only modification of the full system simulator, but also the application source code as well. This fact alone may make the modified simulator approach infeasible, since source code is not always available for both the applications and the full system simulators. Furthermore the tracking of the dependencies within the simulator becomes so cumbersome that this approach is unrealistic. Unlike taint tracking, commonly used in security [29] which requires only a logical *or* of the taint tag bit, dependency tracking requires creating list unions for each operation. Consequently, a taint tracking approach would be unpractical for any application that executes for a reasonable length of time. In addition "taint explosion" would result in unwieldy dependency lists for most memory locations.

Recall that the PDG is generated for a given application mapping. Clearly, if the application mapping changes, a new PDG has to be generated. However, this limitation is shared by all trace based simulation techniques, and we consider it a fair tradeoff for the potential gains in simulation speed and accuracy.

The PDG_GEN algorithm performs poorly on the hotspot and central traffic patterns since many packets are sent to a hotspot node, and few are transmitted from it to the other nodes. This results in the high rate of quasi-dependencies (for hotspot), as well as the low detection rate of the true dependencies. The way we partition the *m* runs (all outgoing links from a node are slowed, for example) does not help to expose the true dependencies in this case, and a method that marks only some outgoing links of a particular node as slow may provide for more accurate results.

6. CONCLUSION AND FUTURE WORK

In this paper we have shown that using network traces which have only total ordering information to predict system level performance (such as execution time or network latency) can lead to erroneous conclusions, because in a real system there are reception dependencies between packets - i.e., some packets cannot be sent until some other packets have been received. We have presented a methodology to infer PDGs from traces that can be used for future design space exploration. The proposed PDG_GEN algorithm has been shown to accurately detect the true dependencies over a wide range of traffic patterns.

The number of quasi-dependencies detected by the current implementation of the PDG_GEN algorithm has limited impact on the execution time or average latency of the inferred PDG, but they do increase the time necessary to perform the simulation. An investigation into techniques to reduce the number of quasi-dependencies should be completed in the future. Different partitioning schemes, as discussed in the previous section, may also be useful in reducing the number of quasi-dependencies and may increase the detection of true dependencies. In addition, the matching algorithm that we use to correlate messages between shared memory traces has room for improvement. It may be possible to identify sections of each trace that correlate well, and in some way account for the sections of each trace in which memory access and coherence events played out differently. We plan to address these and other issues in the future.

Despite the shortcomings of the matching algorithm, our experiments show that inferred PDGs can predict execution time for applications running on shared memory systems with an average error of 13.4% (2.5 times more accurate than trace based simulation without dependencies). We expect our approach to yield even more accurate results for message passing systems, where there is no layer of indirection caused by cache coherence protocols. We plan to investigate the application of this approach to message passing systems in the future.

7. REFERENCES

[1] C. H. van Berkel, “Multi-core for mobile phones,” in *DATE*, 2009, pp. 1260–1265.

[2] A. Adriahantenaina *et al.*, “Spin: A scalable, packet switched, on-chip micro-network,” in *DATE ’03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 20070.

[3] A. Pullini *et al.*, “Fault tolerance overhead in network-on-chip flow control schemes,” in *SBCCI ’05: Proceedings of the 18th annual symposium on Integrated circuits and system design*. New York, NY, USA: ACM, 2005, pp. 224–229.

[4] P. Abad, V. Puente, and J.-A. Gregorio, "Mrr: Enabling fully adaptive multicast routing for cmp interconnection networks," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb. 2009, pp. 355–366.

[5] P. Abad *et al.*, "Rotary router: an efficient architecture for cmp interconnection networks," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 116–125.

[6] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti, "Circuit-switched coherence," in *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 193–202.

[7] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti, "Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence," in *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 35–46.

[8] N. E. Jerger, D. Vantrease, and M. Lipasti, "An evaluation of server consolidation workloads for multi-core designs," in *IISWC '07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 47–56.

[9] 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.

[10] 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.

[11] Y. Pan *et al.*, "Firefly: illuminating future network-on-chip with nanophotonics," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 429–440, 2009.

[12] M. J. Cianchetti, J. C. Kerekes, and D. H. Albonesi, "Phastlane: a rapid transit optical routing network," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 441–450, 2009.

[13] N. Eisley, L.-S. Peh, and L. Shang, "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.

[14] 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.

[15] 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.

[16] N. E. Jerger, L.-S. Peh, and M. Lipasti, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 229–240, 2008.

[17] J. Kim, J. Balfour, and W. Dally, "Flattened butterfly topology for on-chip networks," *Microarchitecture, IEEE/ACM International Symposium on*, vol. 0, pp. 172–182, 2007.

[18] P. Magnusson *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[19] 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.

[20] 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.

[21] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco: Morgan Kaufmann, 2004.

[22] 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. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815999

[23] 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.

[24] R. Marculescu *et al.*, "Outstanding research problems in noc design: system, microarchitecture, and circuit perspectives," *IEEE Transactions on Computer Aided Design of Integrated Ciruits and Systems*, vol. 28, no. 1, pp. 3–21, 2009.

[25] 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.

[26] V. Soteriou, H. Wang, and L. Peh, "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*, Sep. 2006, pp. 104 – 116.

[27] P. V. Gratz and S. W. Keckler, "Realistic workload characterization and analysis for networks-on-chip design," CMP-MSI, 2010, available at http://cegroup.ece.tamu.edu/~pgratz/papers/CMP-MSI2010-workload.pdf.

[28] A.-M. Rahmani *et al.*, "Negative exponential distribution traffic pattern for power/performance analysis of network on chips," in *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 157–162.

[29] G. E. Suh *et al.*, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XI. New York, NY, USA: ACM, 2004, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024404

[30] 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*. ACM, New York, NY, USA, 2010, pp. 31–36.