Netrace: Dependency-Tracking Traces for Efficient Network-on-Chip Experimentation

Joel Hestness
hestness@cs.utexas.edu

Stephen W. Keckler
skeckler@cs.utexas.edu

Computer Architecture and Technology Laboratory
Department of Computer Science
The University of Texas at Austin
cart@cs.utexas.edu – www.cs.utexas.edu/users/cart

Technical Report TR-10-11

Abstract: Network simulation has emerged as the preferred method of evaluating new network-on-chip (NOC) designs, because of lower execution latency compared to full-system simulation and greater fidelity compared to analytical modeling. While network simulation provides quicker results than full-system simulation, it suffers from inaccuracy in the network traffic characteristics. To increase the fidelity of network simulation, this report describes the implementation of a new trace-based network simulation methodology that captures the dependencies between network messages observed in full-system simulation. These dependencies can be enforced in network simulation to ensure a proper ordering of packets and yield more accurate evaluation results. In particular, we show that this method closely mimics network-level characteristics of full-system application traffic, and provides a means of evaluating the relative application-level performance differences between network organizations. Our library, called Netrace, is comprised of source code and network packet traces that can be easily incorporated into new or existing network simulators. The Netrace library API described in this report consists of a set of functions for handling trace files, reading packets from the traces, and tracking the dependencies between packets. The traces we have released are collected from the M5 simulator [3] modeling a 64-core system executing multithreaded applications from the PARSEC v2.1 [2] suite. These traces aim to increase the fidelity of experimental results, while standardizing the experimental methodology used for testing emerging NOC designs.

1 Overview

Chip multiprocessors (CMPs) are scaling toward hundreds of processing cores, and increased diversification and integration of components. Due to the increasing degree of chip complexity, evaluation of on-chip communication presents a major challenge in the design of future systems. While some components of a CMP, such as a processor core or a cache bank, can be designed and evaluated in isolation, many aspects of a network-on-chip (NOC) architecture can only be assessed in the context of numerous interacting components. For instance, evaluating network topology, routing algorithm, or quality-of-service architectures on a small-scale network that can be efficiently simulated may lead to incorrect conclusions about a full-scale network's ability to handle the expected load.

We surveyed existing NOC evaluation methodologies and we observe that today's approaches have major drawbacks limiting their effectiveness or efficiency. For instance, full-system simulation offers high-fidelity performance results, but suffers from long simulation time, application-level scheduling variability, and simulator development complexity. On the other hand, trace-based network simulation approaches improve simulation speed, but they distort traffic injection rates and the effects of congestion due to lack of dependency enforcement between network transactions.

In response, we propose Netrace, a new trace-based NOC evaluation methodology that captures and enforces the dependencies between network messages from a full-system. We show that compared to trace-based NOC simulation without dependencies, Netrace more accurately tracks full-system network-level performance metrics. Further, it predicts relative application-level performance given different NOC designs, and can even be a good predictor of absolute application performance. Finally, the Netrace methodology allows for simplification of trace collection within full-system simulation as we describe in Section 3. This simplification and the subsequent dependency-driven, trace-based NOC simulation reduce simulation time overheads by more than 2.5x.

This technical report is organized as follows. In Section 2, we discuss the need for tracking dependencies in network traffic traces and we define the terminology for classifying dependencies. In Section 3, we describe our method for collecting traces and detecting dependencies. Sections 4 and 5 detail our evaluation of the dependency-enforcing methodology by comparing the performance and fidelity against full-system simulation with a network, and network simulation without dependencies. We describe the Netrace library and application programming interface for using the traces in Section 6. Finally, Section 7 concludes this report, briefly discusses the current status of Netrace, and discusses opportunities for future work.

Along with this technical report, we have made Netrace and a first set of network trace files with dependencies available on the following website:

`http://www.cs.utexas.edu/~netrace`

2 Packet Dependencies

In a survey of current network-on-chip (NOC) research, we found that the majority of NOC research utilizes network simulation as the primary means of testing and evaluating NOC performance. Indeed, NOC simulation returns evaluation results much quicker than full-system simulation, but it fails to provide the same network-level fidelity. For instance, a full-system simulator can run multithreaded applications and extract application-level performance metrics, such as IPC and end-to-end runtime. On the other hand, NOC simulation provides only network-level evaluation metrics, such as average packet latency and throughput for an offered load. NOC simulation has rarely been used to model network traffic that is more realistic than synthetic workloads or simplistic traces from full-system simulation, and this traffic fails to accurately reflect the characteristics of a full-system running an application.

To address these challenges, we propose a new network evaluation methodology that utilizes trace-driven network simulation with added expressiveness. We collect traces from a single execution of a full-system simulator, analyze and track dependencies between network messages, and encode them in the trace. Dependencies can include simple relations between requests and responses as well as more complex chains that span multiple parallel messages, as in the case of coherence traffic. These dependencies can be enforced in network simulation to gain more realistic insights about the impact of network design on application-level performance. Replaying dependency-based traces provides higher fidelity than dependency-oblivious approaches, while yielding results in a fraction of the time of a full-system simulation.

In this section, we describe packet dependencies, their causes, and how they fit into network simulation. Then, to make the dependency discussion concrete, we show the dependencies that exist within the target system we used for collecting a first set of network traces.

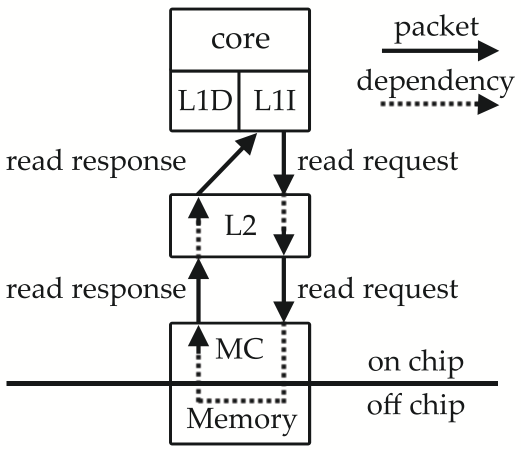


Figure 1: Example Memory Access.

2.1 Dependency Discussion

Abstractly, a network message j is dependent on network message i if either of the following holds:

1. receipt of message i must occur before the sending of message j
2. message j is dependent on message k, and k is dependent on message i.

The second condition is transitivity and stipulates that a message is dependent on another if there exists a chain of dependencies connecting them.

Memory Access:

We adopt the term "memory access" to denote the set of network messages and dependencies that result due to a single core issuing a single memory request. A memory access is a chain of network messages that flow through the memory hierarchy of the system.

As an example of a memory access, consider Figure 1. If a core issues a read request for data that is not currently on-chip, the cache hierarchy initiates a chain of communication that flows from the requesting core's L1, through the L2 to a memory controller, and back up the hierarchy when the data comes on-chip. Each message is dependent on the previous, and the memory access is complete when the data gets back to the requesting core. More complex memory accesses involve the cache coherence protocol and result in longer or wider chains of packets and dependencies.

The concept of a memory access is a useful abstraction for understanding what a network traffic trace with dependencies looks like. For the target system in this study, a network traffic trace consists of a set of memory accesses that are linked end-to-end through response-request dependencies, which we describe below. The packets and dependencies represent a directed, acyclic graph on which a partial order can be defined. The longest chain of dependencies in the graph, along with associated latencies, represents the network critical path for the application from which the traces were collected.

2.2 Dependency Classes

Dependencies between network messages arise as a result of three distinct causes: architectural, microarchitectural, and program-based. We classify network message dependencies based on these causes and describe them in more detail below. For a thorough discussion of these dependency classes in the context of microprocessor design, we refer the reader to Fields [4].

Architectural Dependencies:

Architectural dependencies exist as a result of architectural component interaction. For example, on an L1 cache miss, a request message is issued to the L2 cache. The L2 cache will issue a response message only after receipt of the request, so the response message is dependent on the request. Detecting architectural dependencies only requires knowledge about the architecture of the system, such as that the L2 responds to L1 requests, but does not require knowledge of the microarchitectural implementation of the components.

Microarchitectural Dependencies:

Microarchitectural dependencies are due to microarchitectural implementation details, including buffer capacity, protocols and others. As an example, consider an L1 cache that has a request buffer of fixed size. When this buffer is full of outstanding requests, further requests cannot be issued from the L1. However, when the L1 receives a response message that frees a buffer position, another request can be issued. The resulting request message is dependent on the receipt of a response to a prior outstanding request. In general, tracking microarchitectural dependencies can be difficult as they must be considered on a per-architectural-component basis.

Program behavior dependencies:

Program behavior dependencies are a result of both data- and control-flow dependencies within an application, and they are realized by the microarchitecture of the processor core. An example is when a branch or jump instruction in the control-flow graph causes a new region of instruction memory to be loaded into the L1I cache. The instruction load is dependent on executing an instruction that was loaded previously. Data-flow dependencies, such as read-after-read (RAR) and write-after-read (WAR), cause similar network message dependencies between a load and other loads/stores.

2.3 Target System

Cores	64 on-chip, in-order, Alpha ISA, 2GHz
L1 cache	32KB instruction/32KB data, 4-way associative, 64B lines, 3 cycle access time, MESI coherence protocol
L2 cache	64 bank fully shared S-NUCA, 16MB, 64B lines, 8-way associative, 8 cycle bank access time
Memory	150 cycle access time, 8 on-chip memory controllers

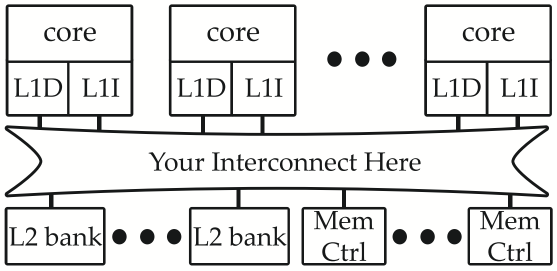


Figure 2: Target System “target:1”

Here, we walk through a case study of dependencies for a particular target system in the next subsection. Figure 2 summarizes this target system, which is comprised of in-order cores with private L1 instruction and data caches, a shared, banked L2 cache, and eight on-die memory controllers. The cache coherence protocol is MESI at the L1 level. On an address interleaved basis, an L2 bank acts as the owner of shared lines and as the backing store for writebacks from the L1s. When an L1 has exclusive or modified access to a line, the L2 may not contain a copy but tracks the coherence state for that line. Since it is the first target system that Netrace supports, we call this system “target:1” for the purposes of tracking.

2.4 Message Dependency Case Study

Table 1 lists the message dependency types that stem from the design of the target machine. The list may omit dependencies that require detailed knowledge of the underlying microarchitecture of the system, similar to the example given in the microarchitectural dependencies discussion in Section 2.2.

Architectural Dependencies:

The first three types of dependencies in the table - request-request, request-response, and response-response - are due to the cache organization of the system, and are thus classified as architectural dependencies. Request-request dependencies are a result of an L2 miss when an L1 cache has issued a request. As an example, when an L1 cache issues a read request that misses in the L2, it triggers another read request from the L2 to a memory controller. The last request-request dependency type, between two writebacks, results when a dirty cache line is written from an L1 cache to the L2, and then later evicted from the L2 to be written back to memory.

The second type of dependency that falls under the architectural classification is the request-response dependency. When a cache architecture component receives a request that it can service, it issues a response back to the requestor. The release of the response is dependent on the receipt of the request at that cache resource.

Corresponding to request-request dependencies is a set of analogous response-response dependencies. When requested data is returned to the chip, a memory controller sends it to the L2, which then forwards the data to the

dependency class	dependency type	src	source message type	intermed. node	dependent message type	same addr	dst
architectural	request-request	L1I	read request	L2	read request	Y	MC
		L1D	read request	L2	read request	Y	MC
		L1D	read exclusive request	L2	read exclusive request	Y	MC
		L1D	read exclusive request	L2	upgrade request	Y	MC
		L1D	upgrade request	L2	upgrade request	Y	MC
		L1D	writeback	L2	writeback	Y	MC
	request-response	L1I	read request	L2	read response	Y	L1I
		L1I	read request	L2	read response + invalidate	Y	L1I
		L1D	read request	L2	read response	Y	L1D
		L1D	read request	L2	read response + invalidate	Y	L1D
		L1D	read exclusive request	L2	read exclusive response	Y	L1D
		L1D	upgrade request	L2	upgrade response	Y	L1D
		L2	read request	MC	read response	Y	L2
		L2	read exclusive request	MC	read exclusive response	Y	L2
		L2	upgrade request	MC	upgrade response	Y	L2
	response-response	MC	read response	L2	read response	Y	L1I
		MC	read response	L2	read response + invalidate	Y	L1I
		MC	read response	L2	read response	Y	L1D
		MC	read response	L2	read response + invalidate	Y	L1D
		MC	read exclusive response	L2	read exclusive response	Y	L1D
		MC	upgrade response	L2	read exclusive response	Y	L1D
		MC	upgrade response	L2	upgrade response	Y	L1D
microarchitectural	coherence	0.L1D	read request	L2	downgrade request	Y	1.L1D
		0.L1D	read exclusive request	L2	invalidate request	Y	1.L1D
		0.L1D	upgrade request	L2	invalidate request	Y	1.L1D
		L2	downgrade request	1.L1D	downgrade response	Y	L2
		L2	downgrade request	1.L1D	read response	Y	0.L1D
		L2	invalidate request	1.L1D	read exclusive response	Y	0.L1D
		L2	invalidate request	L1D	read request	Y	L2
		L2	invalidate request	L1D	read exclusive request	Y	L2
		L2	downgrade request	L1D	upgrade request	Y	L2
		L2	downgrade request	L1D	read request	Y	L2
		L2	downgrade request	L1D	read exclusive request	Y	L2
	response-request	L2	read response	L1D	writeback	N	L2
		L2	read exclusive response	L1D	writeback	N	L2
		L2	read response	L1D	upgrade request	Y	L2
		L2	read exclusive response	L1D	writeback	Y	L2
		L2	upgrade response	L1D	writeback	Y	L2
program-based		L2	read response	L1I	read request	N	L2
		L2	read response	L1D	read request	N	L2
		L2	read response	L1D	read exclusive request	N	L2
		L2	read response	L1D	upgrade request	N	L2
	instruction-data	L2	read response	L1D/L1I	read request	N	L2
		L2	read exclusive response	L1D/L1I	read request	N	L2
		L2	upgrade response	L1D/L1I	read request	N	L2
		L2	read response	L1I/L1D	read request	N	L2
		L2	read response	L1I/L1D	read exclusive request	N	L2
		L2	read response	L1I/L1D	upgrade request	N	L2

Table 1: Network Message Dependency Types for Target System.

requesting L1. The response from the L2 to the L1 is dependent on the response from the memory controller.

Microarchitectural Dependencies:

The next type of dependency listed in the table is a result of a memory access that causes coherence traffic. For instance, if a cache line is shared between multiple L1 data caches, and one of the L1s issues an upgrade request for that cache line, the receipt of that upgrade request at the L2 will trigger the release of invalidation requests to all other sharers of that line. The memory access is completed when the requesting L1 receives an upgrade response from the L2 cache.

In addition to coherence dependencies, the first five response-request dependencies in the table are also a result of the cache microarchitecture, so we classify them as microarchitectural. The first two are due to cache evictions. When a read or read exclusive response is received by an L1 cache that does not have space in the corresponding cache set, it must evict a line. If the line being evicted is dirty, the L1 must issue a writeback to the L2. The writeback is triggered by receipt of the read or read exclusive response. Note here that the address of the cache line being written back is not the same as the address of the received cache line, as shown in the table.

The third response-request dependency represents the case when the L2 responds to a read request from an L1 cache that, after receiving the cache line, would like to write to it. In order to proceed with the write, it needs to request exclusive access to the line by issuing an upgrade request to the L2. The upgrade request is dependent on the read response. The next two lines are a result of similar behavior: a cache line must be in the modified state in the

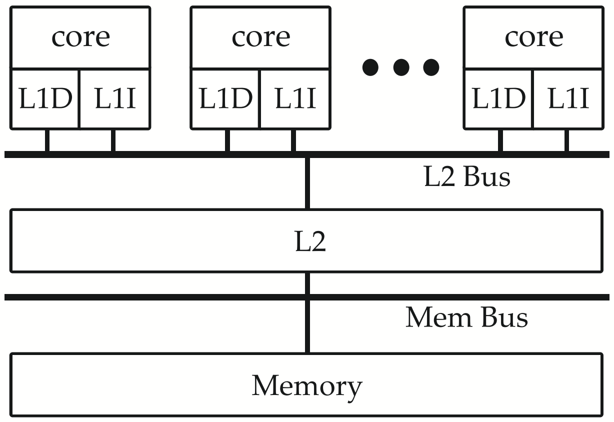


Figure 3: Simulated System.

cache before a writeback can occur.

Program Behavior Dependencies:

The last four response-request dependencies are due to program structure. For instance, the first type in the table describes a branch or jump in the control-flow graph, which might cause a new region of instruction memory to be loaded into the L1I cache. The instruction load is dependent on executing an instruction that was loaded previously. Data-flow dependencies, such as read-after-read (RAR) and write-after-read (WAR), cause similar network message dependencies between a load and other loads/stores. Note here again that the address of the dependent packet is not the same as the address of the received cache line.

In addition to response-request dependencies, there are dependencies that arise due to the interaction of data and instructions in the core. For instance, after data is returned to the L1 data cache, execution of instructions can continue, causing instruction fetches. Similarly, when instruction memory is returned to the L1 instruction cache and executed in the core, it may contain load or store instructions that cause data stream requests through the L1D.

Program behavior dependencies are generally the hardest to track. First, tracking program-based dependencies requires inspection of the machine code of an application or instrumentation of a microprocessor simulator. Second, aside from being dependent on data- and control-flow graphs, these dependencies can be due to microarchitecture of the processor. For instance, a microarchitecture that prefetches down an instruction stream may cause dependencies between instruction memory loads that would not be accessed by a design without prefetching. Third, complicating the detection of these dependencies is the fact that the number of cycles between the receipt of a source message and the release of a dependent message is determined by program structure and processor performance, rather than a more deterministic latency like that of a cache bank access or memory read.

3 Network Packet Traces

This section describes the way that we collect network traffic traces and the post-processing involved with identifying and enforcing dependencies. We conclude the section with a brief discussion of the structure of the packets in the trace file.

3.1 Trace Collection

To collect cache-to-cache communication for the traces, we use the M5 full-system simulator [3] running a modified version of Linux 2.6.27 that supports up to 64 cores. We modified the "Bus" tracing functionality of M5 in a couple ways to support trace collection. First, we eliminated tracing that does not include information about cache-to-cache communication. Second, to collect communication caused by cache coherence, we modified the tracing functionality of the caches and tag stores. When we map this communication to network traffic, these modifications allow us to extract the dependencies between packets and handle the requests caused by coherence.

Figure 3 shows a diagram of the system that we simulate within M5. This simulated system has the same cores and caches as the target system, but they are connected via zero-contention buses. We designed the simulated system in this way to avoid any implied physical realization of the processor and to eliminate any implied or artificial delays or contention in the network traces. To that end, in the simulated system, we model a zero-contention bus between the L1

```
Format:
<inject_cycle>: <bus_name>: src <port_id> dst <port_id> <msg_type> <address>

A       36: tol2bus: src 96 dst -1 ReadReq 0xd040
B       50: tol2bus: src 46 dst -1 ReadReq 0x1b40
C       60: tol2bus: src 16 dst 96 ReadResp 0xd040
D       74: tol2bus: src 16 dst 46 ReadResp 0x1b40
```

Figure 4: Example Simulation Trace Records.

and L2 caches, and another between the L2 cache and memory controllers. By modeling messages without contention, packets are not artificially spaced through time due to scheduling conflicts with other requests. When replayed from the trace with dependencies, the network simulator can be assured that any delay beyond the latency specified in the trace that is experienced by a packet is a result of the network design and not an artifact of the simulated system. Further, use of a bus with no contention helps build an abstraction from any physical implementation of the system since each core is the same "distance" from the L2 cache and memory controller.

To further develop this abstraction, the simulated system uses a unified shared L2 cache, while the target system has a 64 bank distributed shared L2. This organization simplifies post-processing of the simulation trace by focusing L1-to-L2 communication through a single point for an implied global ordering of packets. At the same time, a unified L2 again avoids fixing any implied physical implementation of the system since all L1 requests will be serviced uniformly rather than sent to separate L2 cache banks, possibly of varying latency. We make a similar decision to unify memory in the simulated system. In post-processing, we map the packet sources and destinations to cache banks or memory controllers on an address interleaved basis. This mapping can be modified by the network simulator as desired.

The toughest decision in setting up the simulated system involves modeling component latencies. Some current CMPs have non-uniform cache architectures, and it is likely that an actual implementation of the target system would also feature non-uniform access latencies to various components. However, decisions about non-uniform access latencies in the simulated system would be approximate at best, since the actual latencies of cache components are highly dependent on floorplanning and layout. To maintain abstraction of the simulated system, we avoid assumptions about variable latencies of caches by fixing the access latency across cache banks.

Another aspect of modeling component latency is deciding how to model network latency. Actual network latencies are dependent on the network design, such as the number of routers between source and destination of a message, as well as the amount of contention in the network at a given time. Due to the high variability in network latency, we again assume a fixed latency for network traversal. To decide this latency, we used an analytical model to calculate the average number of hops, and thus average packet latency, across a chip with an aggressive concentrated mesh-like network topology using taxi-cab distance and assuming no contention. Assuming contention-free network traffic is intended to avoid artificial timing delays between packets that might be a result of contention in the simulated system. Instead, the contention should be reflected in the timing of packets in the NOC simulation that obeys dependencies between packets.

Finally, we desire to collect traces from a simulation that is representative of normal executions of the benchmark. M5 is a deterministic simulator, so if the simulated system, operating system and benchmark are unchanged from one simulation to the next, the results will be the same. To ensure that the trace collection execution is representative of normal execution, we run tests that perturb the L2 and memory access latency of the simulated system to verify that the trace collection run does not incur unexpected cache behavior, or result in an unusually long or short runtime.

3.2 Post-processing Simulation Traces

Once we have collected the cache-to-cache communication in the simulated system, we perform post-processing to generate network traffic traces. We have written an application that first syntactically parses the simulation trace to build network packets that include the injection cycle, source node, destination node, and the network command type. The second phase of the application detects and tracks dependencies between network messages.

To make the post-processing step more concrete, consider the example records from a simulation trace as shown in Figure 4. The syntactic parsing of record `A` builds the packet data structure shown in Figure 5. The injection cycle of the packet is read directly from the record. The source and destination ports are translated using the mapping defined within simulation. In this example, source port 96 maps to the L1 data cache at node 47. A destination port of `-1` indicates that a packet is broadcast on the bus. Record `C` shows the response to record `A`, which originates at port 16,

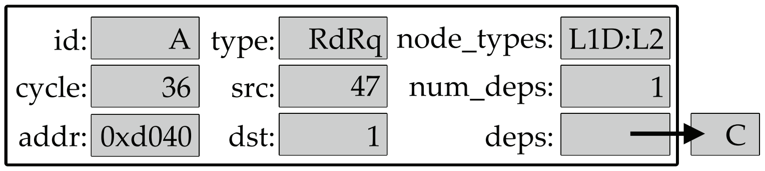


Figure 5: Example Packet

the L2 cache. In this case, we use an address-interleaved mapping to calculate the L2 home node of the cache line based on the address of records `A` and `C`. Packet `A` is added to a window of packets that can be inspected later to detect dependencies.

In the second phase, the application detects and tracks dependencies between packets in the trace. When record `C` is parsed, the application searches through the window of previously parsed records to find packets on which `C` depends. In this example, `C` is the response to `A` and thus dependent on `A`, so the number of dependencies for packet `A` is incremented, and `C` is added to `A`s list of downward dependencies. Similarly, packet `D` is dependent on `B`, so a dependency would be set up between them.

3.3 Packet Structure

This subsection describes the packet structure, `nt_packet_t`, that is returned when the network simulator calls the `nt_read_packet` function of the Netrace reader. Figure 5 shows an example packet data structure in action, and this subsection describes each of the data members of the packet.

`nt_packet_t`:

`id`	An identifier for this packet. This value is not necessarily unique, but it is used by Netrace to identify packets when tracking dependencies.
`cycle`	The timestamp of this packet. This member signals the first cycle at which this packet can be injected. However, it does not account for possible packet dependencies that may delay the injection of the message.
`addr`	The memory address of this packet.
`type`	The network command type of this packet. See the discussion below for more information.
`src`	The source node where this packet is to be injected.
`dst`	The destination node of this packet.
`node_types`	Holds the cache or memory component type for the source and destination nodes of the packet. Example types include "L1 Data Cache" "L2 Cache" or "Memory Controller". See also `nt_get_src_type` and `nt_get_dst_type` API functions.
`num_deps`	The number of downstream packets that are dependent on this packet. See section 2 for more information about dependencies.
`deps`	A pointer to an array of packet IDs which depend on this packet. To ensure proper tracking of dependencies by Netrace, these dependencies should not be modified by the network simulator.

Packet Types

Due to its cache hierarchy and coherence protocol, the caches of the target system use a number of different network packet types for requests and responses. Table 2 lists the packet types and their integer encoding in the packet structure. In our target system, packets of a particular type have a fixed size, which is also noted in the table. The packet type field of the packet structure is 1 byte, which allows for 256 separate encodings to be defined as necessary in future development. In particular, this field could be utilized for more expressive packet size encoding.

Val	Packet Type	Size (B)	Val	Packet Type	Size (B)
0	Invalid Command	-1	14	Upgrade Response	8
1	Read Request	8	15	Read Exclusive Request	8
2	Read Response	72	16	Read Exclusive Response	72
3	Read Response With Invalidate	72	25	Bad Address Error	8
4	Write Request	72	27	Invalidate Request	8
5	Write Response	8	28	Invalidate Response	8
6	Writeback Request	72	29	Downgrade Request	8
13	Upgrade Request	8	30	Downgrade Response	72

Table 2: Packet type encoding

		Traces			Netrace			Full-system with network		
Benchmark	Input	Cycles	Packets	Load	Cycles	Packets	Load	Cycles	Packets	Load
blackscholes	simlarge	894M	89.5M	10.0%	984M	89.5M	9.09%	1.23B	110M	8.96%
canneal	simmedium	300M	74.2M	24.7%	463M	74.2M	16.0%	419M	67.3M	16.0%
x264	simsmall	1.48B	31.3M	2.12%	2.06B	31.3M	1.52%	2.09B	33.7M	1.61%

Table 3: PARSEC ROI Runtime, Packet Counts and Injection Rates.

4 Methodology

In Section 5, we present an evaluation of the fidelity and performance of Netrace. Here, we describe the infrastructure that we have used to perform this evaluation.

For this evaluation, we use two simulators: a custom network simulator and the M5 full-system simulator. In Section 3, we described the simulated system from which we collect the memory traffic traces. After collecting and post-processing the traces, we use them to drive our network simulator. The first configuration neglects the dependencies encoded in the traces and simply injects packets into the network according to the timestamp encoded into the trace. We refer to this trace-based simulation without dependency enforcement as "conventional traces," or simply "traces." The second network simulator configuration is driven by traces that enforce dependencies, and we refer to this configuration as "Netrace."

In order to evaluate the fidelity of Netrace performance estimation at the network- and application-level, we run simulations using two different full-system configurations. First, we aim to evaluate the fidelity of absolute performance numbers predicted by Netrace. For this, we compare results against M5 full-system simulation that models the target system, including the network and an S-NUCA L2. We refer to this configuration as "full-system." Second, we aim to evaluate the relative performance of applications running on systems with varying network latency. For this, we configure M5 similarly to the simulated system, but we vary the fixed network latency from 8 cycles up to 32 cycles to mimic a range of topology options. We refer to this configuration as "full-system + ideal network."

Our NOC traffic traces with dependencies are collected from M5 simulation of the PARSEC v2.1 benchmark suite [2]. We utilize the work of Gebhart et al. [5] for running PARSEC in the M5 simulator [3]. The PARSEC suite contains multiple input sets for each benchmark, and we collect traces for simulation with the simsmall, simmedium and simlarge input sets for all benchmarks that work with up to 64 cores. For all simulations, we the portion of execution that is denoted as the region-of-interest (ROI) defined within the PARSEC suite. The ROI is a portion of the benchmark that executes multiple threads in parallel and exercises a large portion of the program functionality.

5 Evaluation

Chip architects and NOC researchers want to evaluate how well a network performs given realistic application traffic patterns. In evaluating the Netrace methodology, we first wish to validate the full-system realism of traffic traces that obey dependencies. In prior sections, we have argued that dependencies enforce the packet orderings that are seen in full-system simulation, and in this section, we start by showing quantitatively that Netrace mimics network-level characteristics of full-system application traffic.

Beyond ensuring realistic application traffic patterns, NOC researchers also wish to evaluate the impact of network design choices on application performance without the need for time-consuming full-system simulation. Our aim with Netrace is to facilitate exploration of application performance through network-only studies, thus avoiding the need for full-system simulation. We show that under limited conditions, Netrace can accurately predict absolute application runtime. In the absence of these limited conditions, Netrace still does a good job of precisely predicting relative application performance trends, as we describe in a topology case study below.

5.1 Netrace Accuracy

We first evaluate the accuracy of Netrace performance estimates against full-system simulation. As explained in Section 3, our trace collection infrastructure is based on M5 full-system simulation with an ideal network model. For this study, we compare network- and application-level performance metrics under (a) network-only execution of conventional traces, (b) dependency-encoding (Netrace) traces, and (c) full-system simulation under M5 with an accurate network model and static non-uniform access (S-NUCA) L2 cache with cache lines interleaved across multiple cache banks [7]. For this accuracy analysis, in all configurations, we model a mesh topology with a 4-cycle router pipeline and single-cycle link latency.

We note that there are a few differences between full-system simulations used for trace collection and the network-enabled full-system simulations. First, the mapping of cache lines to L2 home nodes is slightly different, though in most cases, this difference does not noticeably change the traffic source and destination distributions. Second, differences exist in the precise modeling of the cache coherence protocol, which can have the effect of increasing or decreasing the absolute number of messages.

Table 3 summarizes the results of the evaluation. We consider three applications from the PARSEC benchmark suite, *blackscholes*, *canneal*, and *x264*, because they show diverse parallelism profiles and dataset sizes. For each application, the table shows the number of simulated cycles, completed packet count, and the effective network load for each simulated setup in the ROI. The average injection rate ("Load" column) here is calculated as the packet count divided by the cycle count.

At the network level, we observe that trace-based simulation without dependency tracking inflates the network load by at least 11.6% and up to 54.4% compared to full-system simulation. In contrast, the average injection rate with Netrace closely tracks the load under M5, with a maximum deviation of 5.8% across the three applications. Thus, Netrace provides excellent network-level correlation relative to full-system execution.

At the application level, we note that Netrace is always a better predictor of execution time than conventional trace-based simulation. The highest accuracy is achieved on *x264*, where the number of simulated cycles under Netrace is within 1.5% of full-system execution. Upon closer inspection, we observe that *x264* exhibits very consistent behavior in repeated full-system simulations with respect to thread placement, instruction counts, memory references and network-level behavior. The minimal variability between runs allows accurate prediction of application-level performance with Netrace.

Variability can have a detrimental effect in validating Netrace accuracy. For instance, *canneal* shows considerable variability in core utilization across multiple M5 executions. When collecting traces from M5, 53 of the 64 cores had threads scheduled to them and the remaining 11 cores were unutilized during the execution. In comparison, due to differences in the way Linux scheduled threads across runs, the network-enabled full-system simulation of the same workload utilized 62 cores. Due to this lower degree of parallelism captured in the trace, Netrace predicts extension of the runtime by 10.5% compared to full-system execution. In contrast, trace-based simulation without dependency enforcement artificially compresses the execution time by over 28% versus full-system simulation (35% versus Netrace) by ignoring dependencies among network transactions. The difference in load balancing also explains the absolute difference in packet counts between the simulations; since multiple threads were competing for fewer cores in the trace-collection simulation, L1 locality was poorer, causing more data requests to L2 caches.

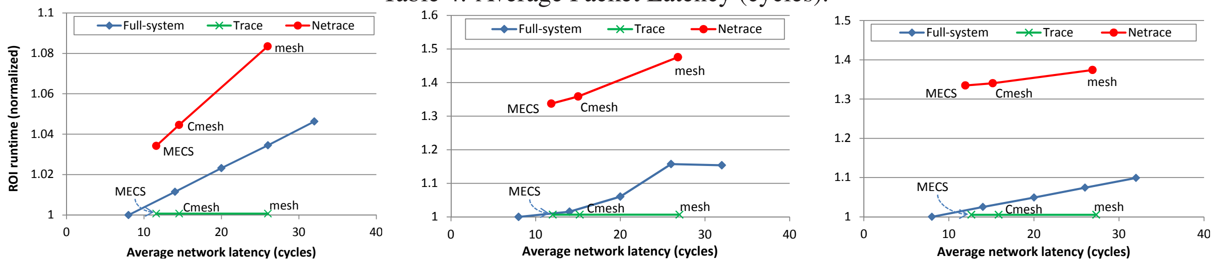
Finally, Netrace underestimates the ROI runtime of *blackscholes* by 20%. This benchmark highlights the difficulty of ensuring that simulation models are the same. In particular, here we see the effects of the different home node mapping for L2 cache lines. In this case, full-system execution shows one of the L2 cache banks receiving a disproportionately large share of requests. Because of the contention, the number of cache misses increases, which thus extends the application runtime. This contention has the added effect of increasing the number of packets seen during the ROI. We do not observe such behavior under Netrace due to a slightly different address mapping, and the execution time is correspondingly shorter. Further, we expect that if the cache line mapping were the same, we would see similar results between the executions. Here again, conventional trace-based simulation is even less accurate, executing 28% fewer cycles compared to full-system and 9% fewer cycles than Netrace.

5.2 Topology Case Study

The previous subsection showed that Netrace is successful at mimicking the network traffic characteristics of a full-system simulation. It also showed that Netrace can be a good predictor of absolute application-level performance. In many cases though, NOC architects only require a relative performance metrics while evaluating a set of design

Benchmark	Method	mesh4	CMesh	MECS
blackscholes	Traces	32.4	14.6	11.6
	Netrace	32.3	14.5	11.6
canneal	Traces	33.5	15.2	12.0
	Netrace	33.4	15.0	11.9
x264	Traces	33.8	15.9	12.6
	Netrace	33.3	15.2	12.0

Table 4: Average Packet Latency (cycles).



(a) blackscholes-large (b) canneal-medium (c) x264-small

Figure 6: Comparison of different methodologies.

candidates. To that end, we evaluate Netrace on its ability to predict relative application performance trends across different network organizations.

Our study looks at the effect of network topology on both network and application performance. We consider three topologies: mesh, four-way concentrated mesh (Cmesh) [1], and Multidrop Express Channels (MECS) [6]. All topologies feature 16-byte links, and as a result, have vastly different latency and throughput characteristics due to differences in connectivity and bisection bandwidth.

Table 4 shows the average packet latency for the three topologies across our application set. Here we see that the mesh, which generally has more hops on average between source and destination nodes has large average packet latencies, while the Cmesh and MECS networks tend to have fewer hops and lower packet latencies. Of particular note in this table is the fact that packet latencies are reduced when Netrace is employed. When ignoring dependencies in NOC simulation, the traces introduce artificial contention for network resources, which elongates average packet latency by as much as 5Netrace eliminates this artificial contention, resulting in more realistic application-level traffic characteristics and average packet latencies.

We next compare the relative application-level performance trends to full-system simulation. For the purposes of this study, our full-system simulator is configured with a fixed-latency interconnect, but the latency is varied from 8 to 32 cycles to mimic a range of topology options. Figure 6 plots PARSEC ROI runtime normalized to full-system simulation with an 8-cycle ideal network. As before, we compare three simulation modes: conventional trace-based NOC simulation without dependency tracking (Trace), dependency-driven trace-based simulation (Netrace), and full-system with fixed-latency network (Full-system). For both trace-based configurations, individual points correspond to the modeled topologies as labeled.

As expected, under full-system simulation, longer network latencies increase the runtime of the application by delaying the completion of cache and memory accesses. In contrast, the runtime of network simulation without dependencies remains unchanged across the different topologies and is always equal to the runtime of the full-system configuration under which the traces were collected. This is because simulation without dependencies fails to provide feedback to throttle packet injection and elongate runtime.

Netrace accurately reflects the relative elongation in application runtime experienced by full-system simulations with longer network delays, as the slopes of Full-system and Netrace lines in all three graphs are quite similar. For instance, for *canneal*, Netrace reveals that a MECS topology yields an 11% speed-up relative to a mesh network. The relative speedup precisely matches that observed in full-system simulation with an equivalent average network latency.

In general, the relative performance of different topologies under Netrace correlates well with full-system measurements. However, as expected, the absolute performance as indicated by ROI runtime is different for the two approaches. In this case, the ideal fixed-latency NOC model in full-system simulation ignores the effects of variable communication latency of a realistic multi-hop network. One of these effects is the divergence in the execution rates of the different threads based on their observed memory latency, which causes some threads to finish ahead of others. In parallel systems, workload completion time is always determined by the slowest thread – an effect not fully captured in simulations with fixed network latency. In contrast, Netrace faithfully captures both network effects and application dependencies, and is a more accurate indicator of NOC's impact on application performance than full-system simulation with an idealized network.

5.3 Simulation Time

Compared to trace-based simulation without dependencies, we find that Netrace slows down network simulation by 1.1–3x, with 2.2x slowdown on average. Most of the slowdown is due to the need to simulate significantly more cycles in a dependency-tracking network, since dependencies serialize packet injection relative to a timestamp-driven execution. The Netrace library itself is responsible for a very small share of execution time.

Compared to full-system simulation with M5, Netrace accelerates simulation time by 1.4x to 17x, with a geometric mean of 2.5x. The reason for the relatively modest difference in execution time between network-only and full-system simulation is that M5 is an event-driven platform, while our network simulator is cycle-accurate. Event-driven simulators are more efficient, since only a subset of system components is simulated on any given cycle. Our experience with the M5 full-system simulator shows that modeling out-of-order cores in M5 increases simulation time by roughly one order of magnitude; meanwhile, network simulation time under our infrastructure in unaffected by processor microarchitecture. Thus, the gap in simulation time would increase as more complex systems are modeled.

6 Application Programming Interface

Netrace provides a simple interface for tracking packet dependencies from a standardized trace file format. Here we start by describing the Netrace source library. We then describe the API functions for handling the trace files, reading packets from the trace, and checking/clearing packet dependencies. We then describe other utility functions, and wrap this section with an example use case of the Netrace library.

6.1 Netrace Library

The Netrace source code includes two major source files, netrace.h and netrace.c, which define the Netrace library.

This library can be compiled into new or existing C and C++ simulators. To compile Netrace into C code, follow standard conventions:

1. First, to include the Netrace library in code that will be using the API, include the header file:

```
#include "netrace.h"
```

2. Second, specify that Netrace should be built in the make file by including it in the sources list and linking when building the final binary.

In order to compile and link in C++ simulators, the following steps must be taken:

1. First, to include the Netrace library in code that will be using the API, declare it as a C external library using:

```
extern "C" {
    #include "netrace.h"
}
```

2. Second, specify that Netrace should be built in the make file by including it in the sources list and linking when building the final binary. The Netrace library should be built using a C compiler.

In the Netrace source, we include example code showing how the library can be included in a cycle-accurate simulator. This code - `main.c, queue.h and queue.c` - is described in the last subsection of this section. We also include a utility application in the `./util/` directory that can be used to look at the information encoded in the header of the trace files, as well as to print portions of the trace.

Netrace depends on the file compression application, bzip2, for handling trace files. Trace files included with the Netrace library should remain bzipped when they are being read by Netrace. The bzip2 library is available for both Linux- and Windows-based systems.

In this section, we introduce two new terms: For a packet, i, the "upward dependencies" of i are the set of packets that i must wait for before it can be injected into the NOC. We use "upward" since this term only includes packets that come before i in time. Complementary to upward dependencies, "downward dependencies" refer to the set of packets that are dependent on packet i, and thus, must come after i in time.

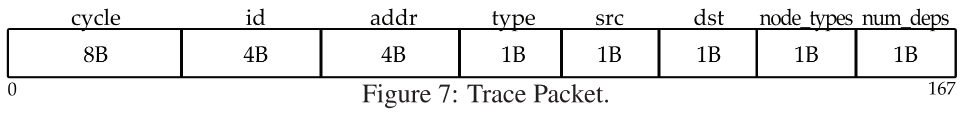


Figure 7: Trace Packet.

6.2 API Functions

`void nt_open_trfile( char* ):`

Opens the trace file specified in the argument, reads the trace header and checks that it is a valid trace file. Note that a trace file must be bzipped in order to be valid. Netrace supports one open trace file at a given time.

`void nt_disable_dependencies( void ):`

This function disables dependency tracking functionality of the reader. It can be called before tracing begins in order to use the packets from the trace without regard to dependencies. When dependencies are disabled, network simulators should inject packets on the cycle that is specified in the trace.

`void nt_seek_region( nt_regionhead_t* ):`

Seeks through the open trace file to the region that is passed as an argument to the function. Regions are defined in the header of the trace file, and they are read when the trace file is opened. To get access to the header and regions, use the `nt_get_trheader` function described below.

`nt_packet_t* nt_read_packet( void ):`

After a trace file has been opened, and its header and regions have been read into memory, the reader can start reading packets from the trace file. When the next packet has been read from the trace, Netrace adds the packet dependencies to its internal dependency-tracking data structures, and returns a pointer to the packet. If the end of the trace file has been reached, Netrace returns NULL.

In the trace file, a packet is encoded as a 21-byte block of data as shown in Figure 7. First, Netrace reads a 21-byte block into a packet data structure. Then it reads `num_deps` 4-byte integers from the trace file that represent the IDs of downward dependent packets from that packet. These dependencies are added to the Netrace internal data structures, and they enable the dependency tracking functionality. Note that the first version trace files are generated and supported on little-endian architectures, so multibyte data are stored least significant bytes first.

`int nt_dependencies_cleared( nt_packet_t* ):`

Tests whether the packet in the argument has any remaining upward dependencies. If the packet in the argument has remaining dependencies, the function returns 0, otherwise it returns a non-zero value indicating that all upward dependencies have been cleared. When enforcing dependencies, a packet should not be injected into the network until this function indicates that dependencies have been cleared. Upward dependencies can be cleared using the `nt_clear_dependencies_free_packet` function.

`void nt_clear_dependencies_free_packet( nt_packet_t* ):`

Signals to the trace reader that the packet in the argument has completed traversal of the network. This function clears downward packet dependencies which cause the `nt_dependencies_cleared` function to return 0. Note that for correct tracking of dependencies, the `deps` member of the packet structure should *not* be modified by the network simulator. Memory allocated for the packet is freed upon completion of the call.

`void nt_close_trfile( void ):`

Closes the currently open trace file and clears all packet dependency data structures. For performance and ease of handling trace files, if there are active packets when the trace file is closed, their dependencies will not be enforced by the reader.

`void nt_init_cleared_packets_list( void ):`

When simulating long traces, much time could be spent checking if dependencies for each packet are cleared on each cycle. This function initializes a list that the trace reader uses to specify to the network simulator which packets have cleared all of their upward dependencies. This function can be used to eliminate excessive calls to `nt_dependencies_cleared`. Section 6.4 includes details on the use of the cleared packets list.

`void nt_init_self_throttling( void ):`

When the cleared packets list is in use, it is possible for the trace reader to throttle reads to the trace file automatically, relieving the network simulator from having to call `nt_read_packet`. The network simulator simply needs to call this function at the beginning of the simulation. After simulation has started, the network simulator must use the `nt_get_cleared_packets_list` and `nt_empty_cleared_packets_list` functions to handle the set of packets whose dependencies have been cleared. See section 6.4 for details on the use of trace reader self-throttling.

`nt_packet_list_t* nt_get_cleared_packets_list( void ):`

This function returns the current list of packets whose dependencies have been cleared. This can only be used after `nt_init_cleared_packets_list` has been called. When the `nt_clear_dependencies_free_packet` function is called on a packet, the trace reader will check to see if all dependencies for any of its downward dependent packets are cleared. If it sees that all dependencies for a packet have been cleared, it will add the packet to the cleared packets list to signal to the network simulator that it can be injected. The cleared packets list includes only those packets whose dependences have been satisfied. Checking this list avoids the need to call `nt_dependencies_cleared` on each waiting packet each cycle.

`void nt_empty_cleared_packets_list( void ):`

After calling `nt_get_cleared_packets_list` and queueing all of the packets whose dependencies have been cleared at the appropriate injection ports, use this function to empty the list for the next stage of network component simulation.

6.3 Utility Functions

`void nt_print_trheader( void ):`

Prints the contents of the trace file header to standard out. The trace_viewer application in the `util/` directory uses this function to print the header for easy access to information about trace files.

`void nt_print_packet( nt_packet_t* ):`

Prints the data members of a packet, including IDs of dependent packets, to standard out.

`nt_header_t* nt_get_trheader( void ):`

Returns a pointer to the header of the currently open trace file or NULL if one is not currently open. The header includes region data for the trace file. Use the region data to seek to a particular position in the trace using the `nt_seek_region` function.

`int nt_get_src_type( nt_packet_t* ):`

Returns the node type of the source of the packet passed as an argument. This function enables the analysis of the way packets flow through the cache hierarchy. Use this with `nt_node_type_to_string` to get a string describing the source node type.

`int nt_get_dst_type( nt_packet_t* ):`

Returns the node type of the destination of the packet passed as an argument. This function enables the analysis of the way packets flow through the cache hierarchy. Use this with `nt_node_type_to_string` to get a string describing the destination node type.

`char* nt_node_type_to_string( int ):`

Returns a string description of the node type passed as an argument, indicating a cache hierarchy component or memory controller. For more detail, see the `nt_node_types` array in `netrace.h`.

`char* nt_packet_type_to_string( nt_packet_t* ):`

Returns a string description of the packet type passed as an argument. See Table 2 and the `nt_packet_types` array in `netrace.h`.

`int nt_get_packet_size( nt_packet_t* ):`

Returns the size, in bytes, of the packet passed as an argument. See the `nt_packet_sizes` array in `netrace.h`.

6.4 Example Use of Netrace Library

In this subsection, we describe three intended use options for Netrace within a network simulator. Each option has different flexibility and overheads as explained. Appendix A contains example source code for these intended uses, and we refer to it in the description here. For these use cases, the network simulator should enforce latencies between dependent packets, and we describe latency enforcement below.

Option 1: Trace Reader Self-Throttling:
This option is the simplest to leverage in a network simulator, since it places much of the burden of packet handling on the Netrace library. Thus, it is the recommended option, especially for simulating long trace runs. In this option, the trace reader tracks the set of packets which have no remaining upward dependencies, and can use this information to automatically throttle the packets that are offered to the network simulator for injection into the simulated network.

To initalize self-throttling, the network simulator should call `nt_init_self_throttling` at the beginning of simulation and use the following process for enforcing dependencies:

1. Each cycle, get the list of cleared packets from the Netrace reader by calling `nt_get_cleared_packets_list`. Enumerating the list, schedule each packet for injection at the appropriate source. *Note: latencies should be imposed as we describe below.
2. Inject packets where possible.
3. Step all of the network components, traversing routers and links, ejecting any packets at their destinations using `nt_clear_dependencies_free_packet`.
4. Repeat from step 1.

Option 2: Flexible Throttling:
In the event that the user would like to change the rate at which packets are read from the trace file, Netrace supports more flexible throttling using the `nt_read_packet` function, while still relying on the Netrace library to offer packets with no upward dependencies to the network simulator. This method may be useful if the user wishes to ignore the packet timestamp that is encoded in the trace. Since this option is a hybrid of options (1) and (3), it is not included in the example code in the Appendix.

The process is as follows:

1. Each cycle, use `nt_read_packet` to read all packets for this cycle.
2. Get the list of cleared packets from the Netrace reader by calling `nt_get_cleared_packets_list`. Enumerating the list, schedule each packet for injection at the appropriate source. *Note: latencies should be imposed as we describe below.
3. Inject packets where possible.
4. Step all of the network components, traversing routers and links, ejecting any packets at their destinations using `nt_clear_dependencies_free_packet`.
5. Repeat from step 1.

Option 3: Flexible Dependencies:
Finally, in the event that the user wishes to selectively enforce dependencies between messages, Netrace supports a mode to allow the network simulator complete control over packet throttling and dependency testing. Dependency testing is handled by checking each active packet in each cycle to see if its dependencies have been cleared using the `nt_dependencies_cleared` function. Note that in this use case, the `nt_dependencies_cleared` function can incur large simulation time overheads.

1. Each cycle, use `nt_read_packet` to read all packets for this cycle.

2. Call `nt_dependencies_cleared` on each new packet.
    (a) If dependencies are cleared, schedule the packet for injection. *Note: latencies should be imposed as we describe below.
    (b) Else, queue the packet to wait for its dependencies to clear.
3. Inject packets where possible.
4. Step all of the network components, traversing routers and links, ejecting packets at their destinations.
5. If any packets were ejected from the network, they may have cleared dependencies, so check each packet in the waiting queues using `nt_dependencies_cleared` to see if it can be injected. If so, schedule the packet for injection in the appropriate inject queue. *Note: latencies should be imposed as we describe below.
6. Repeat from step 1.

We encourage the reader to scan the code in the appendix to see the difference between the use of options (1) and (3). In particular, note that by setting the `reader_throttling` variable to 1 (true) causes a call to `nt_init_self_throttling`, and offloads much of the work of reading and tracking packets to the Netrace reader, which simplifies the code in the network simulator.

Enforcing Latencies
Regardless of the option chosen by the network simulator for enforcing dependencies, it is responsible for enforcing appropriate latencies between dependent messages. For instance, when an L2 cache bank receives a request for data that it can service, a dependency is cleared for the subsequent response packet. The response should not be injected into the network immediately, but held at the L2 for the latency to access the cache bank. The network simulator should schedule the response packet to be injected into the network after the L2 latency has passed.

For requests destined for the L2, we model an 8-cycle bank access latency before the corresponding response is injected. On an L2 miss, we model a 2-cycle tag access latency before the subsequent request to a memory controller is injected. Similarly, for requests destined for the memory controller, we model a latency of 150 cycles to access memory. To evaluate the latency between memory accesses in the core, we calculate the difference in timestamps between the response to the last memory access and the request of the next memory access. This latency represents the time during which the core was computing or accessing the L1 caches. These latencies can be modified by the network simulator, but we recommend using these values, since they are consistent with the latencies enforced during collection of the traces.

The calculation to decide the cycle when a packet should be injected into the network is as follows:

$$inject = max(packet\_timestamp, current\_cycle + latency)$$

where

$$latency = \begin{cases} L2\_TAG\_LATENCY & : src = L2, dst = MEM \\ L2\_DATA\_LATENCY & : src = L2, dst = L1 \\ MEM\_ACCESS\_LATENCY & : src = MEM \\ \Delta TIMESTAMPS & : src = L1, type = request \end{cases}$$

7 Conclusion

As the degree of chip-level integration increases, the time required for full-system CMP simulation will become prohibitive. Researchers want to evaluate application-level performance while only simulating specific components in the system in order to avoid these prohibitive simulation times. In this report, we introduce Netrace, a trace-based network simulation platform that encodes dependencies between network messages. Unlike prior NOC simulation methodologies, Netrace enables application-level performance feedback through network simulation at a fraction of the time required for full-system execution. This allows NOC architects to rapidly evaluate a large design space, and to identify the most promising candidates based on both absolute and relative network- and application-level performance estimates.

Netrace includes a set of NOC traffic traces and a trace file reader library that can be incorporated into new and existing network simulators with little effort. The traffic traces are collected from numerous PARSEC v2.1 benchmarks, and they are available on our website for download:

```
http://www.cs.utexas.edu/~netrace
```

Future Updates:

We will maintain the Netrace library and trace files on the website listed above. Updates, patches and bugfix requests for Netrace or the trace files can be emailed to Joel Hestness:

```
hestness@cs.utexas.edu
```

and we will post them to the website as they become available.

Acknowledgement of Support:

This material is based upon work supported by the National Science Foundation under Grant No. 0811056. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Balfour and W. J. Dally. Design Tradeoffs for Tiled CMP On-chip Networks. In *Proceedings of International Conference on Supercomputing (ICS)*, 2006.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, January 2008.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. In *IEEE Micro*, July/August 2006.

[4] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. *SIGARCH Computer Architecture News*, 29(2), 2001.

[5] M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, and S. W. Keckler. Running PARSEC 2.1 on M5. Technical Report TR-09-32, The University of Texas at Austin, Department of Computer Science, October 2009.

[6] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[7] C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

Appendix:

A Netrace Use Example

Code can also be found in `main.c`:

```
#define L2_LATENCY 8

int i;
int ignore_dependencies = 0;
int start_region = 0;  // region to start tracing from
int reader_throttling = 0; // 1 to switch on
char* tracefile = argv[1];
int packets_left = 0;
unsigned long long int cycle = 0;
nt_packet_t* trace_packet = NULL;
nt_packet_t* packet = NULL;
queue_t** waiting;
queue_t** inject;
queue_t** traverse;
nt_open_trfile( tracefile );
if( ignore_dependencies ) {
    nt_disable_dependencies();
}
nt_print_trheader();
nt_header_t* header = nt_get_trheader();
nt_seek_region( &header->regions[start_region] );
for( i = 0; i < start_region; i++ ) {
    cycle += header->regions[i].num_cycles;
}

int x_nodes = sqrt( header->num_nodes );
int y_nodes = header->num_nodes / x_nodes;
waiting = (queue_t**) malloc( header->num_nodes * sizeof(queue_t*) );
inject = (queue_t**) malloc( header->num_nodes * sizeof(queue_t*) );
traverse = (queue_t**) malloc( header->num_nodes * sizeof(queue_t*) );
if( (waiting == NULL) || (inject == NULL) || (traverse == NULL) ) {
    printf( "malloc fail queues\n" );
    exit(0);
}
for( i = 0; i < header->num_nodes; ++i ) {
    waiting[i] = queue_new();
    inject[i] = queue_new();
    traverse[i] = queue_new();
}

if( !reader_throttling ) {
    trace_packet = nt_read_packet();
} else if( !ignore_dependencies ) {
    nt_init_self_throttling();
}

while( ( cycle <= header->num_cycles ) || packets_left ) {

    // Reset packets remaining check
    packets_left = 0;

    // Get packets for this cycle
    if( reader_throttling ) {
        nt_packet_list_t* list = nt_get_cleared_packets_list();
        for( ; list != NULL; list = list->next ) {
            if( list->node_packet != NULL ) {
                trace_packet = list->node_packet;
                queue_node_t* new_node =
                    (queue_node_t*) nt_checked_malloc( sizeof(queue_node_t) );
                new_node->packet = trace_packet;
                new_node->cycle =
                    (trace_packet->cycle > cycle) ? trace_packet->cycle : cycle;
                queue_push( inject[trace_packet->src], new_node, new_node->cycle );
            } else {
                printf( "Malformed packet list" );
                exit(-1);
            }
        }
        nt_empty_cleared_packets_list();
    } else {
        while( (trace_packet != NULL) && (trace_packet->cycle == cycle) ) {
            // Place in appropriate queue
            queue_node_t* new_node =
                (queue_node_t*) nt_checked_malloc( sizeof(queue_node_t) );
            new_node->packet = trace_packet;
            new_node->cycle =
                (trace_packet->cycle > cycle) ? trace_packet->cycle : cycle;
            if( ignore_dependencies || nt_dependencies_cleared( trace_packet ) ) {
                // Add to inject queue
                queue_push( inject[trace_packet->src], new_node, new_node->cycle );
            } else {
                // Add to waiting queue
                queue_push( waiting[trace_packet->src], new_node, new_node->cycle );
            }
            // Get another packet from trace
            trace_packet = nt_read_packet();
        }
        if( (trace_packet != NULL) && (trace_packet->cycle < cycle) ) {
            // Error check: Crash and burn
            printf( "Invalid trace_packet cycle time: %llu, current cycle: %llu\n",
                    trace_packet->cycle, cycle );
            exit(-1);
        }
    }

    // Inject where possible (max one per node)
    for( i = 0; i < header->num_nodes; ++i ) {
        packets_left |= !queue_empty( inject[i] );
        queue_node_t* temp_node = queue_peek_front( inject[i] );
        if( temp_node != NULL ) {
            packet = temp_node->packet;
            if( (packet != NULL) && (temp_node->cycle <= cycle) ) {
                printf( "Inject: %llu ", cycle );
                nt_print_packet( packet );
                temp_node = queue_pop_front( inject[i] );
                temp_node->cycle = cycle + calc_packet_timing( packet );
                queue_push( traverse[packet->dst], temp_node, temp_node->cycle );
            }
        }
    }

    // Step all network components, Eject where possible
    for( i = 0; i < header->num_nodes; ++i ) {
        packets_left |= !queue_empty( traverse[i] );
        queue_node_t* temp_node = queue_peek_front( traverse[i] );
        if( temp_node != NULL ) {
            packet = temp_node->packet;
            if( (packet != NULL) && (temp_node->cycle <= cycle) ) {
                printf( "Eject: %llu ", cycle );
                nt_print_packet( packet );
                nt_clear_dependencies_free_packet( packet );
                temp_node = queue_pop_front( traverse[i] );
                free( temp_node );
            }
        }
    }

    // Check for cleared dependences... or not if the reader is self-throttling
    if( !reader_throttling ) {
        for( i = 0; i < header->num_nodes; ++i ) {
            packets_left |= !queue_empty( waiting[i] );
            node_t* temp = waiting[i]->head;
            while( temp != NULL ) {
                queue_node_t* temp_node = (queue_node_t*) temp->elem;
                packet = temp_node->packet;
                temp = temp->next;
                if( nt_dependencies_cleared( packet ) ) {
                    // remove from waiting
                    queue_remove( waiting[i], temp_node );
                    // add to inject
                    queue_node_t* new_node =
                        (queue_node_t*) nt_checked_malloc( sizeof(queue_node_t) );
                    new_node->packet = packet;
                    new_node->cycle = cycle + L2_LATENCY;
                    queue_push( inject[i], new_node, new_node->cycle );
                    free( temp_node );
                }
            }
        }
    }

    cycle++;
}

// Clean up
for( i = 0; i < header->num_nodes; ++i ) {
    queue_delete( waiting[i] );
    queue_delete( inject[i] );
    queue_delete( traverse[i] );
}
free( waiting );
free( inject );
free( traverse );
nt_close_trfile();
```