Netrace: Dependency-Driven Trace-Based Network-on-Chip Simulation

Joel Hestness* hestness@cs.utexas.edu
Boris Grot* bgrot@cs.utexas.edu
Stephen W. Keckler*† skeckler@nvidia.com

*Department of Computer Science, The University of Texas at Austin
†Architecture Research Group, NVIDIA

Abstract

Chip multiprocessors (CMPs) and systems-on-chip (SOCs) are expected to grow in core count from a few today to hundreds or more. Since efficient on-chip communication is a primary factor in the performance of large core-count systems, the research community has directed substantial attention to networks-on-chip (NOCs). Current NOC evaluation methodologies include analytical modeling, network simulation, and full-system simulation. However, as core count and system complexity grow, the deficiencies of each of these methods will limit their ability to meet the demands of developers and researchers. Developing efficient NOCs requires high-fidelity, low-overhead NOC evaluation techniques and metrics. To address these challenges, this paper describes a new trace-based network simulation methodology that captures dependencies between network messages observed in full-system simulation of multithreaded applications. We also introduce Netrace, a library of tools and traces that enables targeted NOC simulators to track and replay network messages and their dependencies.

1. INTRODUCTION

As chip multiprocessors (CMPs) scale toward hundreds of processing cores, experimentation becomes evermore complex. Diversification of cores in heterogeneous systems will require added effort in building and debugging CMP simulators. Further, simulation time will increase as CMPs scale and integrate more components onto a single chip. Researchers must find novel methods of testing emerging systems in ways that provide quick performance estimates without sacrificing (too much) fidelity or confidence.

Networks-on-chip (NOCs) promise efficient communication between cores, caches, and memory controllers of future CMPs. Current NOC research assumes that hundreds of processing cores will be available on a chip, and indeed, Tilera has already produced such a design. Existing NOC research covers traditional topics, such as topologies and routing algorithms, as well as more complex ones, like incorporation of emerging manufacturing technologies, quality of service, and interactions of cache architecture, protocols, and network.

In this paper, we survey existing NOC evaluation methodologies and observe that today's approaches have major drawbacks limiting their effectiveness or efficiency. For instance, full-system simulation offers high fidelity but suffers from long runtimes and variability, while trace-based approaches improve simulation speed but distort the injection rates and effects of congestion due to loss of dependency information between network transactions. In response, we propose Netrace, a new trace-based NOC evaluation methodology that captures and enforces the dependencies between network messages. This methodology can offer insight not only into network-level performance, but also application-level performance characteristics and bottlenecks in a fraction of the time of full-system simulation.

Our methodology includes (1) augmentations to the M5 simulator [6] to capture network traces with dependencies, (2) a collection of resulting trace files for a range of benchmarks that are currently drawn from the PARSEC benchmark suite [5], and (3) a trace file reader library that can be incorporated into new and existing network simulators with little effort. We expect that Netrace can help to standardize a set of NOC benchmarks and enable results to be compared across different NOC studies.

Approach	Fidelity	Runtime	Cite
Analytical modeling	Low	Zero	[4, 18]
Simulation: synthetic traffic	Low	Short	[1, 15, 16]
Simulation: Full-system	"Perfect"	Long	[2, 7]
Simulation: application traces	Medium	Medium	[12, 13]
Simulation: application traces w/ dependency tracking	High	Medium	Netrace

Table 1: Network evaluation methodologies

2. EXISTING METHODOLOGIES

Table 1 gives an overview of four existing network evaluation methodologies, followed by the methodology that we are proposing in this paper. Here we detail the biggest benefits and disadvantages of each approach.

Analytical Modeling: Analytical modeling provides a baseline for quick, back-of-the-envelope NOC evaluation, and researchers have developed models to gain specific insights about an NOC, such as power and area footprints, and worst-case traffic patterns [4] [18]. However, as CMP complexity grows, and designs become more dynamic, modeling the complex interplay of architectural design decisions without detailed simulation becomes difficult.

Synthetic Workloads: A common experimental methodology employs an NOC simulator driven by synthetically generated network traffic. Commonly-used patterns include uniform random and transpose, among others, which are designed to stress an NOC in different ways and can provide insight into network bottlenecks through metrics such as throughput and latency across different injection rates. Unfortunately, synthetic workloads do not represent expected

NoCArc '10, December 4, 2010, Atlanta, Georgia, USA

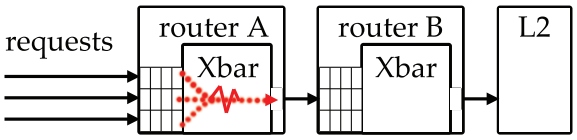


Figure 1: Throttling by Timestamp

traffic within a system running real applications. As such, they fail to offer the designer application-level performance insights, such as memory access time or end-to-end runtime.

A related approach is to develop synthetic traffic generators that mimic actual application-level behavior as in an SOC [15, 17]. These methods require statistical analysis of network traffic traces to tune multiple different generators for appropriate injection rates, causal dependencies between messages, and application phase changes. Such tuning requires substantial effort and is often specific to a particular class of architectures or applications.

Full System Simulation: Full-system simulation provides the highest evaluation fidelity by explicitly modeling the interaction of application and architecture. The insights gained through full-system simulation can reveal NOC bottlenecks with respect to actual applications and the impact of network-level optimizations on application performance.

Despite the considerable benefits, the approach suffers from two important drawbacks. First, the simulation time of a complete system running an operating system and benchmark applications can be on the order of many weeks to months, depending on the level of detail and system size of the simulation, and the duration of the application region of interest. Slow simulation hinders the ability of the designer to get quick feedback and to test a large design space. Second, even with deterministic full-system simulation, the issue of variability arises between simulations spanning a design space. For instance, comparing performance of even small modifications to the NOC can result in large differences in thread-to-core assignment, synchronization contention and operating system effects due to slight timing differences and different execution paths [3]. Such non-determinism makes exploration of the architectural design space of NOCs difficult.

Trace-driven NOC Simulation: As an alternative to full-system simulations, researchers have proposed driving an NOC simulator with traces of network traffic that have been collected from execution of an application. The motivation behind these trace-based approaches is to capture the network-level behavior of real applications while benefitting from the speed of isolated network simulation. Speedups of up to 50x have been reported for trace-driven NOC evaluation over full-system simulation [16].

While application traffic tends to be more realistic than synthetic workloads, trace-driven methodologies have largely ignored the dependencies between network messages. Ignoring dependencies allows interleavings of messages that would not occur in full-system simulation and can cause misleading injection rates due to a lack of dependency-based throttling.

As an example, Figure 1 shows multiple requesters issuing consecutive memory requests to the L2 bank on the right. In a trace-driven configuration, the rate at which the requests enter the network is controlled by the trace packet timestamps that reflect the cycle in which the corresponding memory accesses occurred in full-system simulation. If the packet latency in the simulated NOC is greater than that modeled in full-system mode, the rate at which the packets enter the network can exceed the NOC's ability to deliver them, causing pockets of congestion. Congestion tends to increase network latency and diminish throughput, distorting these metrics and compromising simulation fidelity.

This behavior is an artifact of incorrectly throttled packet injection that ignores transaction ordering and dependencies among messages. As a result of such distortions, existing trace-driven methodologies do not provide meaningful application-level performance metrics [10]. Without such feedback, designers are not able to evaluate the impact of NOC optimizations on application performance.

Dependency-driven NOC Simulation: We propose a new evaluation methodology for trace-based NOC simulation that captures and obeys the dependencies between messages. Our approach is to construct a directed acyclic graph (DAG) between network messages based on the ordering and dependencies among memory transactions recorded during full-system simulation. The dependency information is stored along with packet data in the network trace. By enforcing the ordering constraints in a network simulator, the proposed technique can greatly increase the fidelity of trace-driven evaluation with little impact on simulation speed.

Dependency-tracking trace-driven simulation provides several benefits. First, enforcing dependencies ensures a proper interleaving of network messages, which increases the fidelity of the simulation by eliminating network hotspots that are a result of artificial network contention. Analysis of the remaining hotspots that arise while enforcing dependencies can offer valuable insight into the network bottlenecks that are due to either the network architecture or the communication patterns of the application. Second, since the traces are collected from a single execution of a benchmark, the same application execution paths are enforced regardless of the NOC organization being tested. This strategy makes it easier to assess the impact of different network designs on application-level performance characteristics as compared to full-system simulation, which is plagued by variability in execution paths. Third, the time to run an NOC simulation that obeys dependencies is only slightly longer than NOC simulation without dependencies, offering substantial speedup over full-system simulation. Finally, we anticipate that the rich content of the traces will enable researchers to test the interplay of numerous design points, including cache coherence and cache-to-network protocols, physical placement of cache and memory controllers on a die, and cache line owner/home mappings. We hope that this methodology can be used as a standard for network evaluation as the size and complexity of NOC designs grow.

Trace-based methods, including the one proposed here, are not without drawbacks. Trace file sizes can be large (a gigabyte or more) making distribution difficult. Because each trace is matched to the architecture of the simulated system, testing a different cache organization or processor core will require full-system simulation run to collect new traces. Despite these limitations, we anticipate that the benefits of fidelity and speed will motivate many NOC researchers to use these dependency-based traces and tools.

3. DEPENDENCIES

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 (or triggers) the sending of message j, or (2) message j is dependent on message k, and k is

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
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

Table 2: Target System

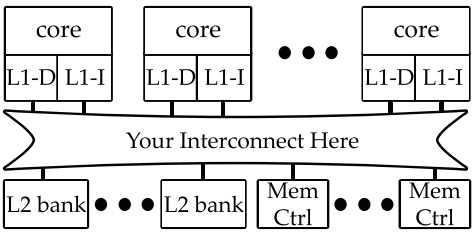


Figure 2: Abstraction in Target System

dependent on message i.

The second condition is transitivity, stipulating that a message is dependent on another if there exists a chain of dependencies linking them. Analogous to a data-flow graph, the set of packets and dependencies in a trace makes up a directed-acyclic graph where the nodes are packets and the edges are dependencies between packets.

One of the simplest examples of a chain of dependencies is a memory access. Figure 3 depicts a memory access from an L1 instruction cache for instruction memory that is not currently on-chip. The request triggers a chain of packets and dependencies that goes to memory to get the data and returns it to the L1-I. More complex memory accesses involve the cache coherence protocol and result in longer or wider chains of packets and dependencies. We use 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.

The concept of a memory access is a useful abstraction when trying to understand what a network traffic trace with dependencies looks like. For the target system in this study, a network traffic trace is made up of a set of memory accesses that are linked end-to-end through dependencies at the processor core. Since the packets and dependencies represent a directed, acyclic graph, 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. This observation indicates the link between network performance and end-to-end runtime of an application, and it is our motivation for tracking dependencies.

3.1 Dependency Classes

Dependencies between network messages arise as a result of three distinct causes: architectural, microarchitectural, and programmatic. We classify network message dependencies based on these causes and describe them in the context of a target system below. For a thorough discussion of these dependency classes in the context of microprocessor design, we refer the reader to Fields [8].

3.2 Target System

To make the message dependency discussion concrete, we walk through a case study of dependencies for a particular target system in the next subsection. Table 2 summarizes the design parameters for 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. This target system corresponds to a specific instance of the abstract architecture shown in Figure 2. The cache coherence protocol is MESI for the L1s. An L2 bank acts as the owner of shared lines and as the backing store for writebacks from the L1s.

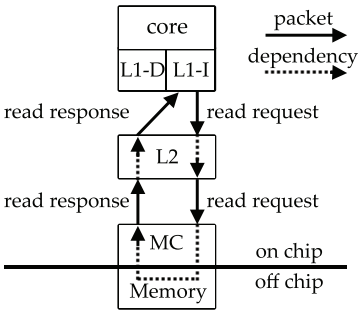


Figure 3: Example Memory Access

3.3 Message Dependency Classes

Architectural dependencies arise as a result of architectural component interaction, such as messages between cores, caches, and memory controllers. For our target system, there are 3 types of architectural dependencies: request-request, request-response and response-response. Each of these different types is depicted in Figure 3. For instance, request-request dependencies occur when a request for data causes a miss in the L2 and thus, a subsequent request to the memory controller for the data. Analogously, when the response comes back on-chip, it first returns to the L2 before being forwarded to the requesting L1. Request-response dependencies occur when an architectural component can service a request, so after it receives the request, it can send the response data. In this target system, the L2 and memory controllers can service requests.

Microarchitectural dependencies are due to microarchitectural implementation details, including buffer capacities, protocols and others. In general, tracking microarchitectural dependencies can be difficult as they must be considered on a per-architectural-component basis.

For the target system, microarchitectural dependencies are caused by cache implementations and the cache coherence protocol. For instance, the L2 cache line size, associativity, replacement policy and capacity affect when and which cache lines are evicted. When the L2 receives a packet i containing new data, it may trigger a writeback of the old data via a message to a memory controller. The writeback packet depends on the receipt of i, causing the eviction.

The coherence protocol is also a source of many dependencies. 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. A similar situation occurs when read requests cause a downgrade request to the owner.

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 L1-I 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.

In practice, program behavior dependencies are difficult to track without inspection of machine code of an application. For the purpose of investigating the efficacy of dependency tracking, our initial target system uses in-order cores, which allow us to make the assumption that memory requests from a core must proceed serially. We have started investigating dependencies in the context of an out-of-order core, and we plan to complete a full analysis in future work.

4. TRACE COLLECTION

We use a two step process for collecting network traffic traces. First, we use full-system simulation to collect memory request traces, which we refer to as the "simulation traces." Then we post-process these simulation traces to extract network messages and detect dependencies between them. This data is output to an encoded network traffic trace. While we describe the process in the context of a particular target system, our methodology can be extended and applied to a variety of system configurations.

4.1 Simulated System

For full-system simulation, we use the M5 simulator [6] running a modified version of Linux 2.6.27 that supports up to 64 cores. We choose to simulate a system that is, in a couple ways, more abstract than the target system described in Section 3. First, the simulated system models a unified L2 cache and a single memory controller. This allows us to concentrate the L2 and memory traffic through a single point, simplifying trace post-processing and eliminating variable access time to physically distinct components.

Second, the simulated system models a fixed-latency bus between the L1 and L2 caches, and another between the L2 and memory controller. Packets are not artificially spaced through time due to contention for communication resources. The combination of these two abstractions avoids assuming any physical layout of the chip by giving the illusion that each core is the same "distance" from the L2 or memory. This approach ensures that the varying latencies that are realized in dependency-enforcing network simulation are not a result of artificial timing effects from the full-system simulation.

While future systems may feature non-uniform cache access latencies, the actual latencies of these components would be highly dependent on floorplanning and layout, so we fix the access latency to a given cache. Another aspect of modeling component latency is deciding how to model network latency. Actual network latencies are dependent on the amount of contention in the network at a given time, as well as the network implementation, such as the number of routers between source and destination of a message. Due to the high variability in network latency, we again assume a fixed latency for message communication. 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 a mesh-like network topology using manhattan distance and assuming no contention. The assumption of contention-free network traffic reflects communication behavior of high-performance applications, which are typically optimized for local communication in an actual system.

```
Format:
<inject_cycle>: <bus>: src <port_id>
         dst <port_id> <type> <addr>

A 36: sys.tol2bus: src 96 dst -1 ReadReq  0xd040
B 50: sys.tol2bus: src 46 dst -1 ReadReq  0xdb40
C 60: sys.tol2bus: src 16 dst 96 ReadResp 0xd040
D 74: sys.tol2bus: src 16 dst 46 ReadResp 0xdb40
```

Figure 4: Example Simulation Trace Records

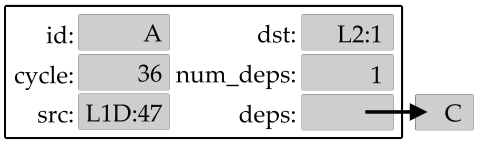


Figure 5: Example Packet

4.2 Post-processing Simulation Traces

After collecting simulation traces from the simulated system, we post-process them to generate network traffic traces. Our post-processing application first syntactically parses the simulation trace to build network packets that include the injection cycle, source and destination node, and others. The second phase inspects a window of packets from the previous 1,000,000 trace cycles to detect and track dependencies between them. The window depth was chosen empirically to contain at least 95% of each dependency type that we track.

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 a packet data structure similar to the one shown in Figure 5. The packet injection cycle is read directly from the record. The source and destination ports are translated using the mapping defined within simulation, which in this case maps port 96 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, the L2 cache. The L2 home node of the cache line is calculated on an address-interleaved basis. Packet `A` is added to the packet window to be inspected later to detect dependencies.

When record `C` is parsed, our application searches through the window of previously parsed records to find packets on which `C` depends. Here, `C` depends 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` depends on `B`, so a dependency is established between them.

4.3 Benchmarks

Our first set of on-chip network traffic traces with dependencies was collected from M5 simulation of the PARSEC v2.1 benchmark suite [5] [9]. 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 the benchmarks that work with simulation up to 64 cores.

5. NETRACE

Netrace is a C/C++ library that includes functionality to read a standardized network traffic trace file and track the dependencies between network messages. This library can be easily incorporated into existing and new network simulators as a means of driving the simulation. A network simulator can employ the API calls of the Netrace system to read packets from a trace and to detect and enforce depen-

dencies. This section outlines the Netrace API as shown in Table 3, and we describe the flexibility of the traces.

Netrace API: After a Netrace file has been opened, the `nt_read_packet` function will return the next packet in the file. A packet, i contains a list of downstream dependent packets that should not be injected into the network until i has been ejected. Netrace adds these dependencies to its internal data structures to power the next two functions.

The `nt_dependencies_cleared` method queries the Netrace data structures to check if, for a packet j, all of its upstream dependent packets have been ejected from the network. If so, the method returns `TRUE` to indicate that j can be injected into the network, otherwise, it returns `FALSE`. The network simulator *must* clear these upward dependencies by calling the `nt_clear_dependencies_free_packet` method on all packets as they are ejected from the network. A call to `nt_clear_dependencies_free_packet` removes the downward dependencies of a packet and frees the packet memory. For packet, i, which is being ejected from the network, if i is the last upward dependency for packet j, then after the call `nt_clear_dependencies_free_packet(`i`)`, subsequent calls to `nt_dependencies_cleared(`j`)` will return `TRUE` indicating that all dependencies have been cleared.

Netrace Flexibility: Figure 2 shows the abstraction of the target system as we discussed in Section 4. Thanks to this abstraction, an NOC simulator can be used to evaluate a wide range of network configurations within this target system using traces with message dependencies.

First, Netrace is designed to support comparisons between traditional NOC design points, such as topologies and routing algorithms. At a detailed level, it can provide application-level insight into bottlenecks caused by the NOC, similar to what we describe in Section 6.

Second, the expressiveness of Netrace traces also supports emerging research directions. For instance, it allows for evaluations of on-chip floorplanning and layout in future systems that may incorporate numerous cores or new manufacturing technologies such as die-stacking. It can be used to evaluate different cache and memory controller counts and locality simply by remapping the sources and destinations of packets aimed at particular components.

Finally, Netrace provides the flexibility to test the interplay of multiple design decisions in a system hierarchy. For example, the traces we distribute assume a particular coherence protocol that can easily be replaced with other protocols as long as the NOC simulator tracks coherence state for live cache lines. This allows for testing the interaction of different network designs and cache coherence protocols, which will be critical as the focus of NOC research comes to include broadcast/multicast support and power concerns.

6. EVALUATION

Methodology: To evaluate the effectiveness of enforcing message dependencies, we compare application-level performance metrics between M5 full-system simulation and trace-based NOC simulation. To assess the affect of varying average packet latency on application-level performance characteristics, we first run full-system simulations with a variety of fixed network latencies. The full-system that we model is the same as our simulated system, which models an ideal network, namely, the communication latency from any node to any other node is the same. We then use a custom NOC simulator to test three network topologies that represent a range of different latency and throughput characteristics: a mesh, a mesh with concentration factor of four (Cmesh) [4], and a multidrop express channel topology (MECS) [11].

Benchmark	Input	Cycles	Packets
blackscholes	simlarge	894M	89.5M
canneal	simmedium	300M	74.2M
x264	simsmall	1.48B	31.3M

Table 4: Full-system simulation ROI data

Discussion: Figure 6 plots the region of interest (ROI) runtime, normalized to full-system simulation with an 8-cycle fixed network latency for three different PARSEC benchmarks. The benchmarks, listed in Table 4, were chosen as examples of the range of application performance characteristics that Netrace can capture. Three different simulation configurations are shown in the figure – full-system, conventional trace-based NOC simulation without dependency tracking (Trace), and dependency-driven NOC simulation (Netrace). For both trace-based configurations, individual points correspond to the three modeled topologies – MECS, concentrated mesh, and mesh, respectively, going from left (lower latency) to right (higher latency) in the figure. For full-system simulations, the topology is unchanged but different fixed network latencies are modeled.

As expected, in 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.

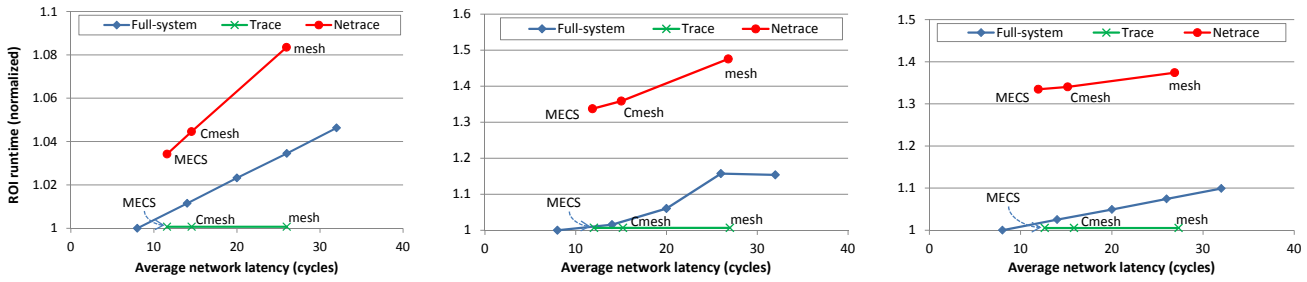
Netrace correctly tracks network dependencies and faithfully serializes transactions, reflecting the relative elongation in application runtime experienced by full-system simulations with longer network delays (i.e. the slope of the full-system and Netrace lines in the figure are quite similar). For instance, for *canneal*, Netrace reveals that a MECS topology yields an 11% speed-up relative to a mesh network while reducing the average packet latency by over 66%, from 26.8 cycles to 11.9 cycles. The relative speedup precisely matches that observed in full-system simulation with equivalent network latencies.

We can also see the effect of communication locality on ROI runtime. As an example, *x264-small* can only utilize 8 cores during the ROI, and the Linux thread scheduler does a good job of placing threads for locality. The end result is that data communication between threads is mostly local, so network topology has a minor effect on ROI runtime. On the other hand, blackscholes spreads the work among all of the cores with little communication affinity. As a result, packets travel larger distances across the NOC, exposing the latency-reducing benefits of low-diameter networks. In this case, Netrace shows that the low-diameter MECS topology improves application performance by 5% compared to the mesh network, which requires a large number of hops between source and destination.

In general, the relative performance of different topologies under Netrace correlates well with full-system measurements. However, the absolute performance as indicated by ROI runtime is quite different for the two approaches. The reason for this phenomenon is that full-system simulation uses an idealized fixed-latency NOC model, which ignores

Method	Description
`nt_read_packet( )`	Read a packet from the Netrace file
`nt_dependencies_cleared( packet )`	Check if upward dependencies have been cleared
`nt_clear_dependencies_free_packet( packet )`	Clear downward dependencies and free the passed packet

Table 3: Netrace API Methods



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

Figure 6: Comparison of different methodologies.

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 likely a more accurate indicator of NOC's impact on application performance than the full-system results here.

7. CONCLUSIONS AND FUTURE WORK

This paper introduces Netrace, a trace-based network simulation platform that encodes dependencies between network messages. Netrace includes a set of NOC traffic traces and trace file reader library that can be incorporated into new and existing network simulators with little effort. We will provide more detail about the Netrace tools and traces in our technical report [14].

Enforcing dependencies between network messages during trace-driven NOC simulation ensures that the network handles a proper interleaving of packets. This strategy increases the fidelity of NOC simulation when compared to other trace-based NOC simulation methodologies by eliminating hotspots due to artificial network contention and focusing bottleneck detection. Compared to full-system simulation, trace-driven NOC simulation with dependencies can be orders of magnitude faster and can avoid the problem of variability of execution path by collecting a trace of a single execution of a benchmark. Finally, Netrace enables the analysis of the interaction of numerous design decisions and application-level evaluation metrics, most importantly, application runtime.

In the future, we plan to extend Netrace to support systems with out-of-order cores. We will also be considering new on-chip system configurations, possibly with different numbers of network terminal nodes and cache hierarchies.

REFERENCES

[1] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving Predictable Performance Through Better Memory Controller Placement in Many-core CMPs. *SIGARCH Computer Architecture News*, 37(3):451–461, 2009.

[2] N. Agarwal, L.-S. Peh, and N. K. Jha. In-Network Snoop Ordering (INSO): Snoopy Coherence on Unordered Interconnects. In *HPCA*, pages 67–78, 2009.

[3] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *HPCA*, pages 7–18, 2003.

[4] J. Balfour and W. J. Dally. Design Tradeoffs for Tiled CMP On-chip Networks. In *ICS*, pages 187–198, 2006.

[5] 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.

[6] 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*, pages 52–60, July/August 2006.

[7] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Application-Aware Prioritization Mechanisms for On-chip Networks. In *MICRO*, pages 280–291, 2009.

[8] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. *SIGARCH Computer Architecture News*, 29(2):74–85, 2001.

[9] 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.

[10] C. Grecu, A. Ivanov, P. P, A. Jantsch, E. Salminen, and R. Marculescu. An initiative towards open network-on-chip benchmarks. `http://www.ocpip.org/socket/whitepapers/NoC-Benchmarks-WhitePaper-15.pdf`, 2007.

[11] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. In *HPCA*, 2009.

[12] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive Virtual Clock: A Flexible, Efficient, and Cost-Effective QOS Scheme for Networks-On-Chip. In *MICRO*, pages 268–279, 2009.

[13] M. Hayenga, N. E. Jerger, and M. Lipasti. SCARAB: A Single Cycle Adaptive Routing and Bufferless Network. In *MICRO*, pages 244–254, 2009.

[14] J. Hestness and S. W. Keckler. Netrace: Dependency-Tracking Traces for Efficient Network-on-Chip Experimentation. Technical Report TR-10-11, The University of Texas at Austin, Department of Computer Science, `http://www.cs.utexas.edu/~netrace`, October 2010.

[15] S. Mahadevan, F. Angiolini, J. Sparsø, L. Benini, and J. Madsen. A Reactive and Cycle-True IP Emulator for MPSoC Exploration. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 27(1):109–122, 2008.

[16] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparso, and J. Madsen. A network traffic generator model for fast network-on-chip simulation. In *DATE*, pages 780–785, 2005.

[17] A. Scherrer, A. Fraboulet, and T. Risset. Automatic Phase Detection for Stochastic On-Chip Traffic Generation. In *CODES+ISSS*, pages 88–93, 2006.

[18] B. Towles and W. J. Dally. Worst-case Traffic for Oblivious Routing Functions. In *SPAA*, pages 1–8, 2002.