# On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-Core Interconnects

George Nychis†, Chris Fallin†, Thomas Moscibroda§, Onur Mutlu†, Srinivasan Seshan†
† Carnegie Mellon University
{gnychis,cfallin,onur,srini}@cmu.edu
§ Microsoft Research Asia
moscitho@microsoft.com

## ABSTRACT

In this paper, we present network-on-chip (NoC) design and contrast it to traditional network design, highlighting similarities and differences between the two. As an initial case study, we examine *network congestion* in bufferless NoCs. We show that congestion manifests itself differently in a NoC than in traditional networks. Network congestion reduces system throughput in congested workloads for smaller NoCs (16 and 64 nodes), and limits the scalability of larger bufferless NoCs (256 to 4096 nodes) even when traffic has locality (e.g., when an application's required data is mapped nearby to its core in the network). We propose a new source throttling-based congestion control mechanism with application-level awareness that reduces network congestion to improve system performance. Our mechanism improves system performance by up to 28% (15% on average in congested workloads) in smaller NoCs, achieves linear throughput scaling in NoCs up to 4096 cores (attaining similar performance scalability to a NoC with large buffers), and reduces power consumption by up to 20%. Thus, we show an effective application of a network-level concept, congestion control, to a class of networks – bufferless on-chip networks – that has not been studied before by the networking community.

## Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiprocessors – Interconnection architectures; C.2.1 [**Network Architecture and Design**]: Packet-switching networks

**Keywords** On-chip networks, multi-core, congestion control

## 1. INTRODUCTION

One of the most important trends in computer architecture in the past decade is the move towards multiple CPU cores on a single chip. Common chip multiprocessor (CMP) sizes today range from 2 to 8 cores, and chips with hundreds or thousands of cores are likely to be commonplace in the future [9, 55]. Real chips exist already with 48 cores [34], 100 cores [66], and even a research prototype with 1000 cores [68]. While increased core count has allowed processor chips to scale without experiencing complexity and power dissipation problems inherent in larger individual cores, new challenges also exist. One such challenge is to design an efficient and scalable interconnect between cores. Since the interconnect carries all inter-cache and memory traffic (i.e., all data accessed by the programs running on chip), it plays a critical role in system performance and energy efficiency.

Unfortunately, the traditional bus-based, crossbar-based, and other non-distributed designs used in small CMPs do not scale to the medium- and large-scale CMPs in development. As a result, the architecture research community is moving away from traditional centralized interconnect structures, instead using interconnects with distributed scheduling and routing. The resulting Networks on Chip (NoCs) connect cores, caches and memory controllers using packet switching routers [15], and have been arranged both in regular 2D meshes and a variety of denser topologies [29, 41]. The resulting designs are more network-like than conventional small-scale multi-core designs. These NoCs must deal with many problems, such as scalability [28], routing [31,50], congestion [10,27,53,65], and prioritization [16, 17, 30], that have traditionally been studied by the networking community rather than the architecture community.

While different from traditional processor interconnects, these NoCs also differ from existing large-scale computer networks and even from the traditional multi-chip interconnects used in large-scale parallel computing machines [12, 45]. On-chip hardware implementation constraints lead to a different tradeoff space for NoCs compared to most traditional off-chip networks: chip area/space, power consumption, and implementation complexity are first-class considerations. These constraints make it hard to build energy-efficient network buffers [50], make the cost of conventional routing and arbitration [14] a more significant concern, and reduce the ability to over-provision the network for performance. These and other characteristics give NoCs unique properties, and have important ramifications on solutions to traditional networking problems in a new context.

In this paper, we explore the adaptation of conventional networking solutions to address two particular issues in next-generation bufferless NoC design: *congestion management* and *scalability*. Recent work in the architecture community considers bufferless NoCs as a serious alternative to conventional buffered NoC designs due to chip area and power constraints[1] [10,20,21,25,31,49,50,67]. While bufferless NoCs have been shown to operate efficiently under moderate workloads and limited network sizes (up to 64 cores), we find that with higher-intensity workloads and larger network sizes (e.g., 256 to 4096 cores), the network operates inefficiently


*SIGCOMM'12,* August 13–17, 2012, Helsinki, Finland.


[1] Existing prototypes show that NoCs can consume a substantial portion of system power (28% in the Intel 80-core Terascale chip [33], 36% in the MIT RAW chip [63], and 10% in the Intel Single-Chip Cloud Computer [34]).

and does not scale effectively. As a consequence, application-level system performance can suffer heavily.

Through evaluation, we find that congestion limits the efficiency and scalability of bufferless NoCs, even when traffic has locality, e.g., as a result of intelligent compiler, system software, and hardware data mapping techniques. Unlike traditional large-scale computer networks, NoCs experience congestion in a fundamentally different way due to unique properties of both NoCs and bufferless NoCs. While traditional networks suffer from congestion collapse at high utilization, a NoC's cores have a *self-throttling* property which avoids this congestion collapse: slower responses to memory requests cause pipeline stalls, and so the cores send requests less quickly in a congested system, hence loading the network less. However, congestion does cause the system to operate at less than its peak throughput, as we will show. In addition, congestion in the network can lead to increasing inefficiency as the network is scaled to more nodes. We will show that addressing congestion yields better performance scalability with size, comparable to a more expensive NoC with buffers that reduce congestion.

We develop a new congestion-control mechanism suited to the unique properties of NoCs and of bufferless routing. First, we demonstrate how to detect impending congestion in the NoC by monitoring *injection starvation*, or the inability to inject new packets. Second, we show that simply throttling all applications when congestion occurs is not enough: since different applications respond differently to congestion and increases/decreases in network throughput, the network must be application-aware. We thus define an application-level metric called *Instructions-per-Flit* which distinguishes between applications that should be throttled and those that should be given network access to maximize system performance. By dynamically throttling according to periodic measurements of these metrics, we reduce congestion, improve system performance, and allow the network to scale more effectively. In summary, we make the following **contributions**:

- We discuss **key differences** between NoCs (and bufferless NoCs particularly) and traditional networks, to frame NoC design challenges and research goals from a networking perspective.
- From a **study of scalability and congestion**, we find that the bufferless NoC's scalability and efficiency are limited by congestion. In small networks, congestion due to network-intensive workloads limits throughput. In large networks, even with *locality* (placing application data nearby to its core in the network), congestion still causes application throughput reductions.
- We propose a new low-complexity and high performance congestion control mechanism in a bufferless NoC, motivated by ideas from both networking and computer architecture. To our knowledge, this is the first work that comprehensively examines congestion and scalability in *bufferless* NoCs and provides an effective solution based on the properties of such a design.
- Using a large set of real-application workloads, we demonstrate improved performance for small (4x4 and 8x8) bufferless NoCs. Our mechanism improves system performance by up to 28% (19%) in a 16-core (64-core) system with a 4x4 (8x8) mesh NoC, and 15% on average in congested workloads.
- In large (256 – 4096 core) networks, we show that congestion limits scalability, and hence that congestion control is *required* to achieve linear performance scalability with core count, even when most network traversals are local. At 4096 cores, congestion control yields a 50% throughput improvement, and up to a 20% reduction in power consumption.

# 2. NOC BACKGROUND AND UNIQUE CHARACTERISTICS

We first provide a brief background on on-chip NoC architectures, bufferless NoCs, and their unique characteristics in comparison to traditional and historical networks. We refer the reader to [8, 14] for an in-depth discussion.

## 2.1 General NoC Design and Characteristics

In a chip multiprocessor (CMP) architecture that is built on a NoC substrate, the NoC typically connects the processor nodes and their private caches with the shared cache banks and memory controllers (Figure 1). A NoC might also carry other control traffic, such as interrupt and I/O requests, but it primarily exists to service cache miss requests. On a cache miss, a core will inject a memory request packet into the NoC addressed to the core whose cache contains the needed data, or the memory controller connected to the memory bank with the needed data (for example). This begins a data exchange over the NoC according to a cache coherence protocol (which is specific to the implementation). Eventually, the requested data is transmitted over the NoC to the original requester.

**Design Considerations:** A NoC must service such cache miss requests quickly, as these requests are typically on the user program's critical path. There are several first-order considerations in NoC design to achieve the necessary throughput and latency for this task: chip area/space, implementation complexity, and power. As we provide background, we will describe how these considerations drive the NoC's design and endow it with unique properties.

**Network Architecture / Topology:** A high-speed router at each node connects the core to its neighbors by links. These links may form a variety of topologies (e.g., [29, 40, 41, 43]). Unlike traditional off-chip networks, an on-chip network's topology is statically known and usually very regular (e.g., a mesh). The most typical topology is the two-dimensional (2D) Mesh [14], shown in Figure 1. The 2D Mesh is implemented in several commercial processors [66, 70] and research prototypes [33, 34, 63]. In this topology, each router has 5 input and 5 output channels/ports: one from each of its four neighbors and one from the network interface (NI). Depending on the router architecture and the arbitration and routing policies (which impact the number of pipelined arbitration stages), each packet spends between 1 cycle (in a highly optimized best case [50]) and 4 cycles at each router before being forwarded.

Because the network size is relatively small and the topology is statically known, global coordination and coarse-grain network-wide optimizations are possible and often less expensive than distributed mechanisms. For example, our proposed congestion control mechanism demonstrates the effectiveness of such global coordination and is very cheap (§6.5). Note that fine-grained control (e.g., packet routing) must be based on local decisions, because a router processes a packet in only a few cycles. At a scale of thousands of processor clock cycles or more, however, a central controller can feasibly observe the network state and adjust the system.

**Data Unit and Provisioning:** A NoC conveys packets, which are typically either request/control messages or data messages. These packets are partitioned into *flits*: a unit of data conveyed by one link in one cycle; the smallest independently-routed unit of traffic.[2] The width of a link and flit varies, but 128 bits is typical. For NoC performance reasons, links typically have a latency of only one or two cycles, and are pipelined to accept a new flit every cycle.

[2]In many *virtual-channel buffered* routers [13], the smallest independent routing unit is the packet, and flits serve only as the unit for link and buffer allocation (i.e., flow control). We follow the design used by other bufferless NoCs [20, 22, 36, 50] in which flits carry routing information due to possible deflections.

Unlike conventional networks, NoCs cannot as easily overprovision bandwidth (either through wide links or multiple links), because they are limited by power and on-chip area constraints. The tradeoff between bandwidth and latency is different in NoCs. Low latency is critical for efficient operation (because delays in packets cause core pipeline stalls), and the allowable window of in-flight data is much smaller than in a large-scale network because buffering structures are smaller. NoCs also lack a direct correlation between network throughput and overall system throughput. As we will show (§4), for the same network throughput, choosing differently which L1 cache misses are serviced in the network can affect system throughput (instructions per cycle per node) by up to 18%.

**Routing:** Because router complexity is a critical design consideration in on-chip networks, current implementations tend to use much more simplistic routing mechanisms than traditional networks. The most common routing paradigm is x-y routing. A flit is first routed along the x-direction until the destination's y-coordinate is reached; then routed to the destination in the y-direction.

**Packet Loss:** Because links are on-chip and the entire system is considered part of one failure domain, NoCs are typically designed as *lossless* networks, with negligible bit-error rates and no provision for retransmissions. In a network without losses or explicit drops, ACKs or NACKs are not necessary, and would only waste on-chip bandwidth. However, particular NoC router designs can choose to explicitly *drop* packets when no resources are available (although the bufferless NoC architecture upon which we build does not drop packets, others do drop packets when router outputs [31] or receiver buffers [20] are contended).

**Network Flows & Traffic Patterns:** Many architectures split the shared cache across several or all nodes in the system. In these systems, a program will typically send traffic to many nodes, often in parallel (when multiple memory requests are parallelized). Multithreaded programs also exhibit complex communication patterns where the concept of a "network flow" is removed or greatly diminished. Traffic patterns are driven by several factors: private cache miss behavior of applications, the data's locality-of-reference, phase behavior with local and temporal bursts, and importantly, self-throttling [14]. Fig. 6 (where the Y-axis can be viewed as traffic intensity and the X-axis is time) shows temporal variation in injected traffic intensity due to application phase behavior.

## 2.2 Bufferless NoCs and Characteristics

The question of **buffer size** is central to networking, and there has recently been great effort in the community to determine the right amount of buffering in new types of networks, e.g. in data center networks [2, 3]. The same discussions are also ongoing in on-chip networks. Larger buffers provide additional bandwidth in the network; however, they also can significantly increase power consumption and the required chip area.

Recent work has shown that it is possible to completely eliminate buffers from NoC routers. In such *bufferless NoCs*, power consumption is reduced by 20-40%, router area on die is reduced by 40-75%, and implementation complexity also decreases [20, 50].[3] Despite these reductions in power, area and complexity, application performance degrades minimally for low-to-moderate network intensity workloads. The general system architecture does not differ from traditional buffered NoCs. However, the lack of buffers requires different injection and routing algorithms.

[3]Another evaluation [49] showed a slight energy advantage for buffered routing, because control logic in bufferless routing can be complex and because buffers can have less power and area cost if custom-designed and heavily optimized. A later work on bufferless design [20] addresses control logic complexity. Chip designers may or may not be able to use custom optimized circuitry for router buffers, and bufferless routing is appealing whenever buffers have high cost.

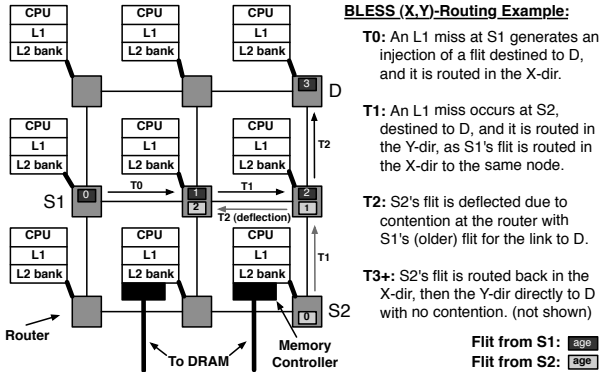



**Figure 1: A 9-core CMP with BLESS routing example.**

**Bufferless Routing & Arbitration:** Figure 1 gives an example of *injection*, *routing* and *arbitration*. As in a buffered NoC, injection and routing in a bufferless NoC (e.g., *BLESS* [50]) happen synchronously across all cores in a clock cycle. When a core must send a packet to another core, (e.g., *S1* to *D* at T0 in Figure 1), the core is able to inject each flit of the packet into the network as long as one of its output links is free. Injection requires a free output link since there is no buffer to hold the packet in the router. If no output link is free, the flit remains queued at the processor level.

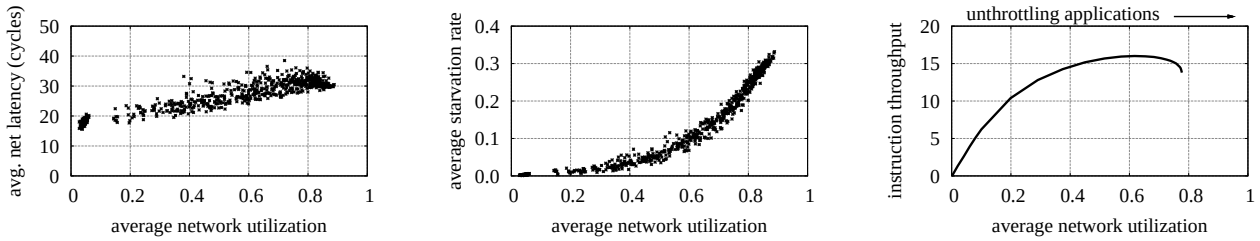
An age field is initialized to 0 in the header and incremented at each hop as the flit is routed through the network. The routing algorithm (e.g, XY-Routing) and arbitration policy determine to which neighbor the flit is routed. Because there are no buffers, flits must pass through the router pipeline without waiting. When multiple flits request the same output port, *deflection* is used to resolve contention. Deflection arbitration can be performed in many ways. In the *Oldest-First* policy, which our baseline network implements [50], if flits contend for the same output port (in our example, the two contending for the link to D at time T2), ages are compared, and the oldest flit obtains the port. The other contending flit(s) are *deflected* (*misrouted* [14]) – e.g., the flit from *S2* in our example. Ties in age are broken by other header fields to form a total order among all flits in the network. Because a node in a 2D mesh network has as many output ports as input ports, routers never block. Though some designs drop packets under contention [31], the bufferless design that we consider does not drop packets, and therefore ACKs are not needed. Despite the simplicity of the network's operation, it operates efficiently, and is livelock-free [50].

Many past systems have used this type of deflection routing (also known as *hot-potato routing* [6]) due to its simplicity and energy/area efficiency. However, it is particularly well-suited for NoCs, and presents a set of challenges distinct from traditional networks.

**Bufferless Network Latency:** Unlike traditional networks, the injection latency (time from head-of-queue to entering the network) can be significant (§3.1). In the worst case, this can lead to starvation, which is a fairness issue (addressed by our mechanism - §6). In-network latency in a bufferless NoC is relatively low, even under high congestion (§3.1). Flits are quickly routed once in the network without incurring buffer delays, but may incur more deflections.

# 3. LIMITATIONS OF BUFFERLESS NOCS

In this section, we will show how the distinctive traits of NoCs place traditional networking problems in new contexts, resulting in new challenges. While prior work [20,50] has shown significant reductions in power and chip-area from eliminating buffers in the network, that work has focused primarily on low-to-medium network




(a) Average network latency in cycles (Each point represents one of the 700 workloads).

(b) As the network becomes more utilized, the overall starvation rate rises significantly.

(c) We unthrottle applications in a 4x4 network to show suboptimal performance when run freely.

**Figure 2: The effect of congestion at the network and application level.**

load in conventionally sized (4x4 and 8x8) NoCs. Higher levels of network load remain a challenge in purely bufferless networks; achieving efficiency under the additional load without buffers. As the size of the CMP increases (e.g., to 64x64), these efficiency gains from bufferless NoCs will become increasingly important, but as we will show, new scalability challenges arise in these larger networks. Congestion in such a network must be managed in an intelligent way in order to ensure scalability, even in workloads that have high traffic locality (e.g., due to intelligent data mapping).

We explore limitations of bufferless NoCs in terms of network load and network size with the goal of understanding *efficiency* and *scalability*. In §3.1, we show that as network load increases, application-level throughput reduces due to congestion in the network. This congestion manifests differently than in traditional buffered networks (where in-network latency increases). In a bufferless NoC, network admission becomes the bottleneck with congestion and cores become "*starved*," unable to access the network. In §3.2, we monitor the effect of the congestion on application-level throughput as we scale the size of the network from 16 to 4096 cores. Even when traffic has locality (due to intelligent data mapping to cache slices), we find that congestion significantly reduces the scalability of the bufferless NoC to larger sizes. These two fundamental issues motivate *congestion control for bufferless NoCs*.

## 3.1 Network Congestion at High Load

First, we study the effects of high workload intensity in the bufferless NoC. We simulate 700 real-application workloads in a 4x4 NoC (methodology in §6.1). Our workloads span a range of network utilizations exhibited by real applications.

**Effect of Congestion at the Network Level:** Starting at the network layer, we evaluate the effects of workload intensity on network-level metrics in the small-scale (4x4 mesh) NoC. Figure 2(a) shows average network latency for each of the 700 workloads. Notice how per-flit network latency generally remains stable (within 2x from baseline to maximum load), even when the network is under heavy load. This is in stark contrast to traditional buffered networks, in which the per-packet network latency increases significantly as the load in the network increases. However, as we will show in §3.2, network latency increases more with load in larger NoCs as other scalability bottlenecks come into consideration.

Deflection routing shifts many effects of congestion from within the network to network admission. In a highly-congested network, it may no longer be possible to efficiently inject packets into the network, because the router encounters free slots less often. Such a situation is known as **starvation**. We define **starvation rate** ($\sigma$) as the fraction of cycles in a window of $W$, in which a node tries to inject a flit but cannot: $\sigma = \frac{1}{W}\sum_i^W starved(i) \in [0,1]$. Figure 2(b) shows that starvation rate grows superlinearly with network utilization. Starvation rates at higher network utilizations are significant. Near 80% utilization, the average core is blocked from injecting into the network 30% of the time.

These two trends – relatively stable in-network latency, and high queueing latency at network admission – lead to the conclusion that network congestion is better measured in terms of *starvation* than in terms of latency. When we introduce our congestion-control mechanism in §5, we will use this metric to drive decisions.

**Effect of Congestion on Application-level Throughput:** As a NoC is part of a complete multicore system, it is important to evaluate the effect of congestion at the application layer. In other words, network-layer effects only matter when they affect the performance of CPU cores. We define *system throughput* as the application-level instruction throughput: for $N$ cores, **System Throughput** = $\sum_i^N IPC_i$, where $IPC_i$ gives *instructions per cycle* at core $i$.

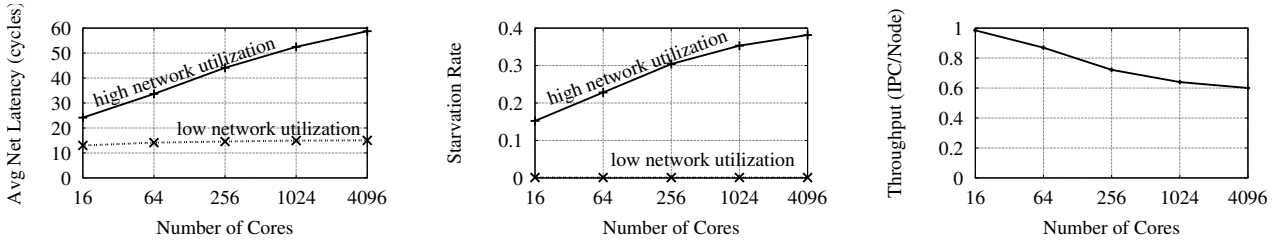
To show the effect of congestion on application-level throughput, we take a network-heavy sample workload and *throttle* all applications at a throttling rate swept from 0. This throttling rate controls how often all routers that desire to inject a flit are blocked from doing so (e.g., a throttling rate of 50% indicates that half of all injections are blocked). If an injection is blocked, the router must try again in the next cycle. By controlling the injection rate of new traffic, we are able to vary the network utilization over a continuum and observe a full range of congestion. Figure 2(c) plots the resulting system throughput as a function of average net utilization.

This static-throttling experiment yields two key insights. First, network utilization does not reach 1, i.e., the network is never fully saturated even when unthrottled. The reason is that applications are naturally *self-throttling* due to the nature of out-of-order execution: a thread running on a core can only inject a relatively small number of requests into the network before stalling to wait for replies. This limit on outstanding requests occurs because the core's instruction window (which manages in-progress instructions) cannot retire (complete) an instruction until a network reply containing its requested data arrives. Once this window is stalled, a thread cannot start to execute any more instructions, hence cannot inject further requests. This self-throttling nature of applications helps to prevent *congestion collapse*, even at the highest possible network load.

Second, this experiment shows that injection throttling (i.e., a form of congestion control) can yield increased application-level throughput, even though it explicitly blocks injection some fraction of the time, because it reduces network congestion significantly. In Figure 2(c), a gain of 14% is achieved with simple static throttling.

However, static and homogeneous throttling across all cores does not yield the best possible improvement. In fact, as we will show in §4, throttling the wrong applications can significantly reduce system performance. This will motivate the need for *application-awareness*. Dynamically throttling the proper applications based on their relative benefit from injection yields significant system throughput improvements (e.g., up to 28% as seen in §6.2).

**Key Findings:** *Congestion contributes to high starvation rates and increased network latency. Starvation rate is a more accurate indicator of the level of congestion than network latency in a*




(a) Average network latency with CMP size. (b) Starvation rate with CMP size. (c) Per-node throughput with CMP size.

**Figure 3: Scaling behavior: even with data locality, as network size increases, effects of congestion become more severe and scalability is limited.**

*bufferless network. Although collapse does not occur at high load, injection throttling can yield a more efficient operating point.*

## 3.2 Scalability to Large Network Size

As we motivated in the prior section, scalability of on-chip networks will become critical as core counts continue to rise. In this section, we evaluate the network at sizes much larger than common 4x4 and 8x8 design points [20,50] to understand the scalability bottlenecks. However, the simple assumption of uniform data striping across all nodes no longer makes sense at large scales. With simple uniform striping, we find per-node throughput degrades by 73% from a 4x4 to 64x64 network. Therefore, we model increased data locality (i.e., intelligent data mapping) in the shared cache slices.

In order to model locality reasonably, independent of particular cache or memory system implementation details, we assume an exponential distribution of data-request destinations around each node. The private-cache misses from a given CPU core access shared-cache slices to service their data requests with an exponential distribution in distance, so most cache misses are serviced by nodes within a few hops, and some small fraction of requests go further. This approximation also effectively models a small amount of global or long-distance traffic, which can be expected due to global coordination in a CMP (e.g., OS functionality, application synchronization) or access to memory controllers or other global resources (e.g., accelerators). For this initial exploration, we set the distribution's parameter $\lambda = 1.0$, i.e., the average hop distance is $1/\lambda = 1.0$. This places 95% of requests within 3 hops and 99% within 5 hops. (We also performed experiments with a power-law distribution of traffic distance, which behaved similarly. For the remainder of this paper, we assume an exponential locality model.)

**Effect of Scaling on Network Performance:** By increasing the size of the CMP and bufferless NoC, we find that the impact of congestion on network performance increases with size. In the previous section, we showed that despite increased network utilization, the network latency remained relatively stable in a 4x4 network. However, as shown in Figure 3(a), as the size of the CMP increases, average latency increases significantly. While the 16-core CMP shows an average latency delta of 10 cycles between congested and non-congested workloads, congestion in a 4096-core CMP yields nearly 60 cycles of additional latency per flit on average. This trend occurs despite a fixed data distribution ($\lambda$ parameter) – in other words, despite the same average destination distance. Likewise, shown in Figure 3(b), starvation in the network increases with CMP size due to congestion. Starvation rate increases to nearly 40% in a 4096-core system, more than twice as much as in a 16-core system, for the same per-node demand. This indicates that the network becomes increasingly inefficient under congestion, despite locality in network traffic destinations, as CMP size increases.

**Effect of Scaling on System Performance:** Figure 3(c) shows that the decreased efficiency at the network layer due to congestion degrades the entire system's performance, measured as IPC/node, as the size of the network increases. This shows that congestion is limiting the effective scaling of the bufferless NoC and system under higher intensity workloads. As shown in §3.1 and Figure 2(c), reducing congestion in the network improves system performance. As we will show in §6.3, reducing the congestion in the network significantly improves scalability with high-intensity workloads.

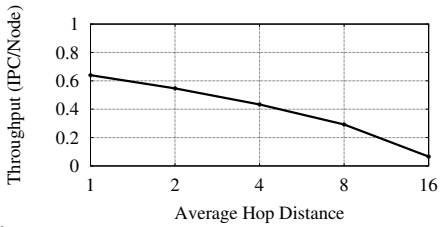



**Figure 4: Sensitivity of per-node throughput to degree of locality.**

**Sensitivity to degree of locality:** Finally, Figure 4 shows the sensitivity of system throughput, as measured by IPC per node, to the degree of locality in a 64x64 network. This evaluation varies the $\lambda$ parameter of the simple exponential distribution for each node's destinations such that $1/\lambda$, or the average hop distance, varies from 1 to 16 hops. As expected, performance is highly sensitive to the degree of locality. For the remainder of this paper, we assume that $\lambda = 1$ (i.e., average hop distance of 1) in locality-based evaluations.

**Key Findings:** *We find that even with data locality (e.g., introduced by compiler and hardware techniques), as NoCs scale into hundreds and thousands of nodes, congestion becomes an increasingly significant concern for system performance. We show that per-node throughput drops considerably as network size increases, even when per-node demand (workload intensity) is held constant, motivating the need for congestion control for efficient scaling.*

# 4. THE NEED FOR APPLICATION-LEVEL AWARENESS IN THE NOC

Application-level throughput decreases as network congestion increases. The approach taken in traditional networks – to throttle applications in order to reduce congestion – will enhance performance, as we already showed in §3.1. However, we will show in this section that *which applications* are throttled can significantly impact per-application and overall system performance. To illustrate this, we have constructed a $4 \times 4$-mesh NoC that consists of 8 instances each of `mcf` and `gromacs`, which are memory-intensive and non-intensive applications, respectively [61]. We run the applications with no throttling, and then statically throttle each application in turn by 90% (injection blocked 90% of time), examining application and system throughput.

The results provide key insights (Fig. 5). First, *which application is throttled has a significant impact on overall system throughput.* When `gromacs` is throttled, overall system throughput drops by 9%. However, when `mcf` is throttled by the same rate, the overall system throughput increases by 18%. Second, *instruction throughput is not an accurate indicator for whom to throttle.* Although `mcf` has lower instruction throughput than `gromacs`, system through-

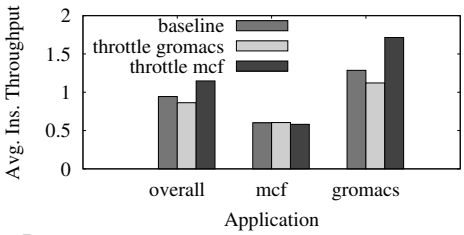



**Figure 5: Throughput after selectively throttling applications.**

put increases when `mcf` is throttled, with little effect on `mcf` (-3%). Third, *applications respond differently to network throughput variations*. When `mcf` is throttled, its instruction throughput decreases by 3%; however, when `gromacs` is throttled by the same rate, its throughput decreases by 14%. Likewise, `mcf` benefits little from increased network throughput when `gromacs` is throttled, but `gromacs` benefits greatly (25%) when `mcf` is throttled.

The reason for this behavior is that each application has a varying L1 cache miss rate, requiring a certain volume of traffic to complete a given instruction sequence; this measure depends wholly on the behavior of the program's memory accesses. Extra latency for a single flit from an application with a high L1 cache miss rate will not impede as much forward progress as the same delay of a flit in an application with a small number of L1 misses, since that flit represents a greater fraction of forward progress in the latter.

**Key Finding:** *Bufferless NoC congestion control mechanisms need application awareness to choose whom to throttle.*

**Instructions-per-Flit:** The above discussion implies that not all flits are created equal – i.e., that flits injected by some applications lead to greater forward progress when serviced. We define *Instructions-per-Flit* (IPF) as the ratio of *instructions retired* in a given period by an application $I$ to *flits of traffic* $F$ associated with the application during that period: $\textbf{IPF} = I/F$. IPF is only dependent on the L1 cache miss rate, and is thus independent of the congestion in the network and the rate of execution of the application. Thus, it is a stable measure of an application's current network intensity. Table 1 shows the *average* IPF values for a set of real applications. As shown, IPF can vary considerably: `mcf`, a memory-intensive application produces approximately 1 flit on average for every instruction retired (IPF=1.00), whereas `povray` yields an IPF four orders of magnitude greater: 20708.

| *Application* | *Mean* | *Var.* | *Application* | *Mean* | *Var.* |
|---|---|---|---|---|---|
| matlab | 0.4 | 0.4 | cactus | 14.6 | 4.0 |
| health | 0.9 | 0.1 | gromacs | 19.4 | 12.2 |
| mcf | 1.0 | 0.3 | bzip2 | 65.5 | 238.1 |
| art.ref.train | 1.3 | 1.3 | xml_trace | 108.9 | 339.1 |
| lbm | 1.6 | 0.3 | gobmk | 140.8 | 1092.8 |
| soplex | 1.7 | 0.9 | sjeng | 141.8 | 51.5 |
| libquantum | 2.1 | 0.6 | wrf | 151.6 | 357.1 |
| GemsFDTD | 2.2 | 1.4 | crafty | 157.2 | 119.0 |
| leslie3d | 3.1 | 1.3 | gcc | 285.8 | 81.5 |
| milc | 3.8 | 1.1 | h264ref | 310.0 | 1937.4 |
| mcf2 | 5.5 | 17.4 | namd | 684.3 | 942.2 |
| tpcc | 6.0 | 7.1 | omnetpp | 804.4 | 3702.0 |
| xalancbmk | 6.2 | 6.1 | dealII | 2804.8 | 4267.8 |
| vpr | 6.4 | 0.3 | calculix | 3106.5 | 4100.6 |
| astar | 8.0 | 0.8 | tonto | 3823.5 | 4863.9 |
| hmmer | 9.6 | 1.1 | perlbench | 9803.8 | 8856.1 |
| sphinx3 | 11.8 | 95.2 | povray | 20708.5 | 1501.8 |

**Table 1: Average IPF values and variance for evaluated applications.**

Fig. 5 illustrates this difference: `mcf`'s low IPF value (1.0) indicates that it can be heavily throttled with little impact on its throughput (-3% @ 90% throttling). It also gains relatively less when other applications are throttled (e.g., <+1% when `gromacs` is throttled). However, `gromacs`' higher IPF value implies that its performance will suffer if it is throttled (-10%), but can gain from additional network throughput (+25%).

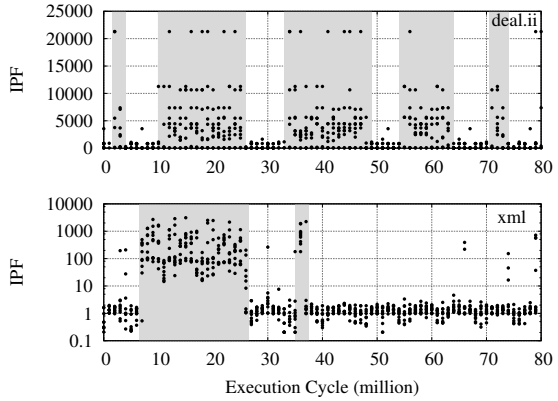



**Figure 6: IPF is dynamic and varies with application phases.**

**Need for Dynamic Throttling:** Until now, we have illustrated throttling as a static mechanism. In Figure 5, we throttled one application or the other throughout its entire execution. While applications can exhibit "average" behavior (e.g., an average IPF) by which they could be throttled, performing *only* static throttling based on average behavior can degrade performance and reduce efficiency relative to what could be possible because *application behavior is dynamic over time* (thus, IPF is also dynamic – also shown by the variance in Table 1). Throttling one application under congestion may help improve system throughput at one point in time, but hurt system performance at another point in time.

To illustrate this, we plot two applications' IPF values over time in Figure 6: **deal.II** (a finite element analysis application) and **xml** (an XML parser). As shown, application behavior changes over time, and in fact, applications can have "phases" through which they cycle (e.g., in the case of *deal.II*). Throttling either application during congestion in the periods where IPF is low (non-shaded) will improve system performance without significantly degrading the application's performance. However, throttling either in high-IPF phases will significantly degrade the application's performance (and thus the overall system's performance). Therefore, a successful throttling mechanism must continually monitor application intensity and the level of congestion, dynamically adjusting over time. IPF is a dynamic metric that enables this.

**Key Findings:** *IPF (Instructions-per-Flit) quanitifies application intensity and service requirements, enabling application-aware throttling mechanisms. Applications have phases that vary widely in network intensity. A dynamic throttling mechanism is thus needed, and this mechanism must monitor IPF dynamically.*

# 5. CONTROLLING CONGESTION

Section 4 defined a metric that determines an application's network intensity and its response to throttling. As shown in §4, *when the network is congested*, we must consider application-layer information to throttle effectively. We improve instruction throughput by throttling applications when they have low IPF. This works for three reasons: 1) applications with low IPF are relatively insensitive to throttling compared to applications with high IPF, 2) applications with high IPF benefit more at the application-level from increased network throughput than those with low IPF, and 3) throttling applications when they have low IPF is more effective at reducing congestion, because these applications are more network-intensive.

**Basic Idea**: We propose an interval-based congestion control algorithm that periodically (every 100,000 cycles, at least 10x shorter than typical application phases – see Fig. 6): 1) detects conges-

tion based on starvation rates, 2) determines IPF of applications, 3) if the network is congested, throttles *only* the nodes on which certain applications are running (chosen based on their IPF). Our algorithm, described here, is summarized in Algos. 1, 2, and 3.

**Mechanism:** A *key difference* of this mechanism to the majority of currently existing congestion control mechanisms in traditional networks [35, 38] is that it is a *centrally-coordinated* algorithm. This is possible in an on-chip network, and in fact is cheaper in our case (Central vs. Distributed Comparison: §6.6).

Since the on-chip network exists within a CMP that usually runs a single operating system (i.e., there is no hardware partitioning), the system software can be aware of all hardware in the system and communicate with each router in some hardware-specific way. As our algorithm requires some computation that would be impractical to embed in dedicated hardware in the NoC, we find that a hardware/software combination is likely the most efficient approach. Because the mechanism is periodic with a relatively long period, this does not place burden on the system's CPUs. As described in detail in §6.5, the pieces that integrate tightly with the router are implemented in hardware for practicality and speed.

There are several components of the mechanism's periodic update: first, it must determine when to throttle, maintaining appropriate responsiveness without becoming too aggressive; second, it must determine whom to throttle, by estimating the IPF of each node; and third, it must determine how much to throttle in order to optimize system throughput without hurting individual applications. We address these elements and present a complete algorithm.

**When to Throttle**: As described in §3, *starvation rate* is a superlinear function of network congestion (Fig. 2(b)). We use starvation rate ($\sigma$) as a per-node indicator of congestion in the network. Node $i$ is *congested* if:

$$\sigma_i > \min(\beta_{starve} + \alpha_{starve}/IPF_i, \gamma_{starve}) \quad (1)$$

where $\alpha$ is a scale factor, and $\beta$ and $\gamma$ are lower and upper bounds, respectively, on the threshold (we use $\alpha_{starve} = 0.4$, $\beta_{starve} = 0.0$ and $\gamma_{starve} = 0.7$ in our evaluation, determined empirically; sensitivity results and discussion can be found in §6.4). It is important to factor in IPF since network-intensive applications will naturally have higher starvation due to higher injection rates. Throttling is *active* if at least one node is congested.

**Whom to Throttle**: When throttling is active, a node is throttled if its intensity is above average (not all nodes are throttled). In most cases, the congested cores are not the ones throttled; only the heavily-injecting cores are throttled. The cores to throttle are chosen by observing IPF: lower IPF indicates greater network intensity, and so nodes with IPF below average are throttled. Since we use central coordination, computing the mean IPF is possible without distributed averaging or estimation. The *Throttling Criterion* is:

$$\text{If } \textit{throttling is active} \text{ AND } IPF_i < mean(IPF).$$

The simplicity of this rule can be justified by our observation that IPF in most workloads tend to be widely distributed: there are memory-intensive applications and CPU-bound applications. We find the separation between application classes is clean for our workloads, so a more intelligent and complex rule is not justified.

Finally, we observe that this throttling rule results in relatively stable behavior: the decision to throttle depends only on the instructions per flit (IPF), which is *independent* of the network service provided to a given node and depends only on that node's program characteristics (e.g., cache miss rate). Hence, this throttling criterion makes a throttling decision that is robust and stable.

**Determining Throttling Rate**: We throttle the chosen applications proportional to their application intensity. We compute *throttling rate*, the fraction of cycles in which a node cannot inject, as:

$$R \Leftarrow \min(\beta_{throt} + \alpha_{throt}/IPF, \gamma_{throt}) \quad (2)$$

**Algorithm 1** Main Control Algorithm (in software)

```
Every T cycles:
collect IPF[i], σ[i] from each node i
/* determine congestion state */
congested ⇐ false
for i = 0 to N_nodes − 1 do
    starve_thresh = min(β_starve + α_starve/IPF[i], γ_starve)
    if σ[i] > starve_thresh then
        congested ⇐ true
    end if
end for
/* set throttling rates */
throt_thresh = mean(IPF)
for i = 0 to N_nodes − 1 do
    if congested AND IPF[i] < throt_thresh then
        throttle_rate[i] = min(β_throt + α_throt/IPF[i], γ_throt)
    else
        throttle_rate[i] ⇐ 0
    end if
end for
```

**Algorithm 2** Computing Starvation Rate (in hardware)

```
At node i:
σ[i] ⇐ Σ_{k=0}^{W} starved(current_cycle − k)/W
```

**Algorithm 3** Simple Injection Throttling (in hardware)

```
At node i:
if trying to inject in this cycle and an output link is free then
    inj_count[i] ⇐ (inj_count[i] + 1) mod MAX_COUNT
    if inj_count[i] ≥ throttle_rate[i] ∗ MAX_COUNT then
        allow injection
        starved(current_cycle) ⇐ false
    else
        block injection
        starved(current_cycle) ⇐ true
    end if
end if
Note: this is one possible way to implement throttling with simple, deterministic hardware. Randomized algorithms can also be used.
```

where IPF is used as a measure of application intensity, and $\alpha$, $\beta$ and $\gamma$ set the scaling factor, lower bound and upper bound respectively, as in the starvation threshold formula. Empirically, we find $\alpha_{throt} = 0.90$, $\beta_{throt} = 0.20$ and $\gamma_{throt} = 0.75$ to work well. Sensitivity results and discussion of these parameters are in §6.4.

**How to Throttle:** When throttling a node, only its data requests are throttled. Responses to service requests from other nodes are *not* throttled; this could further impede a starved node's progress.

# 6. EVALUATION

In this section, we present an evaluation of the effectiveness of our congestion-control mechanism to address high load in small NoCs (4x4 and 8x8) and scalability in large NoCs (up to 64x64).

## 6.1 Methodology

We obtain results using a cycle-level simulator that models the target system. This simulator models the network routers and links, the full cache hierarchy, and the processor cores at a sufficient level of detail. For each application, we capture an instruction trace of a representative execution slice (chosen using PinPoints [56]) and replay each trace in its respective CPU core model during simulation. Importantly, the simulator is a *closed-loop* model: the backpressure of the NoC and its effect on presented load are accurately captured. Results obtained using this simulator have been published in past NoC studies [20, 22, 50]. Full parameters for the simulated system are given in Table 2). The simulation is run for 10 million cycles, meaning that our control algorithm runs 100 times per-workload.

| | |
|---|---|
| Network topology | 2D mesh, 4x4 or 8x8 size |
| Routing algorithm | FLIT-BLESS [50] (example in §2) |
| Router (Link) latency | 2 (1) cycles |
| Core model | Out-of-order |
| Issue width | 3 insns/cycle, 1 mem insn/cycle |
| Instruction window size | 128 instructions |
| Cache block | 32 bytes |
| L1 cache | private 128KB, 4-way |
| L2 cache | shared, distributed, perfect cache |
| L2 address mapping | Per-block interleaving, XOR mapping; randomized exponential for locality evaluations |

**Table 2: System parameters for evaluation.**

**Workloads and Their Characteristics:** We evaluate 875 multiprogrammed workloads (700 16-core, 175 64-core). Each consists of independent applications executing on each core. The applications do not coordinate with each other (i.e., each makes progress independently and has its own working set), and each application is fixed to one core. Such a configuration is expected to be a common use-case for large CMPs, for example in cloud computing systems which aggregate many workloads onto one substrate [34].

Our workloads consist of applications from SPEC CPU2006 [61], a standard benchmark suite in the architecture community, as well as various desktop, workstation, and server applications. Together, these applications are representative of a wide variety of network access intensities and patterns that are present in many realistic scenarios. We classify the applications (Table 1) into three intensity levels based on their average instructions per flit (IPF), i.e., network intensity: H (Heavy) for less than 2 IPF, M (Medium) for 2 – 100 IPF, and L (Light) for $>100$ IPF. We construct balanced workloads by selecting applications in seven different workload categories, each of which draws applications from the specified intensity levels: *{H,M,L,HML,HM,HL,ML}*. For a given workload category, the application at each node is chosen randomly from all applications in the given intensity levels. For example, an **H**-category workload is constructed by choosing the application at each node from among the high-network-intensity applications, while an **HL**-category workload is constructed by choosing the application at each node from among all high- and low-intensity applications.

**Congestion Control Parameters:** We set the following algorithm parameters based on empirical parameter optimization: the update period $T = 100K$ cycles and the starvation computation window $W = 128$. The minimum and maximum starvation rate thresholds are $\beta_{starve} = 0.0$ and $\gamma_{starve} = 0.70$ with a scaling factor of $\alpha_{starve} = 0.40$. We set the throttling minimum and maximum to $\beta_{throttle} = 0.20$ and $\gamma_{throttle} = 0.75$ with scaling factor $\alpha_{throttle} = 0.9$. Sensitivity to these parameters is evaluated in §6.4.

## 6.2 Application Throughput in Small NoCs

**System Throughput Results:** We first present the effect of our mechanism on overall system/instruction throughput (average IPC, or instructions per cycle, per node, as defined in §3.1) for both 4x4 and 8x8 systems. To present a clear view of the improvements at various levels of network load, we evaluate gains in overall system throughput plotted against the average network utilization (measured without throttling enabled). Fig. 7 presents a scatter plot that shows the percentage gain in overall system throughput with our mechanism in each of the 875 workloads on the 4x4 and 8x8 system. The maximum performance improvement under congestion (e.g., load $>0.7$) is 27.6% with an average improvement of 14.7%.

Fig. 8 shows the maximum, average, and minimum system throughput gains on each of the workload categories. The highest average and maximum improvements are seen when all applications in the workload have High or High/Medium intensity. As expected, our mechanism provides little improvement when all applications in the workload have Low or Medium/Low intensity, because the network is adequately provisioned for the demanded load.

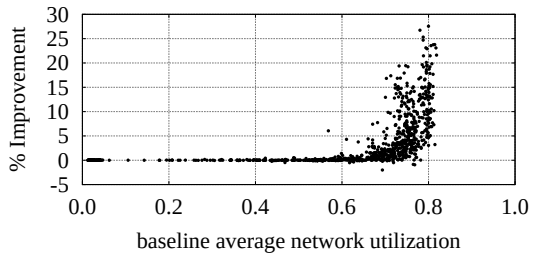



**Figure 7: Improvements in overall system throughput (4x4 and 8x8).**

**Improvement in Network-level Admission:** Fig. 9 shows the CDF of the 4x4 workloads' average starvation rate when the baseline average network utilization is greater than 60%, to provide insight into the effect of our mechanism on starvation when the network is likely to be congested. Using our mechanism, only 36% of the congested 4x4 workloads have an average starvation rate greater than 30% (0.3), whereas without our mechanism 61% have a starvation rate greater than 30%.

**Effect on Weighted Speedup (Fairness):** In addition to instruction throughput, a common metric for evaluation is *weighted speedup* [19,59], defined as $WS = \sum_i^N \frac{IPC_{i,shared}}{IPC_{i,alone}}$, where $IPC_{i,shared}$ and $IPC_{i,alone}$ are the *instructions per cycle* measurements for application $i$ when run together with other applications and when run alone, respectively. $WS$ is $N$ in an ideal $N$-node system with no interference, and drops as application performance is degraded due to network contention. This metric takes into account that different applications have different "natural" execution speeds; maximizing it requires maximizing the rate of progress – compared to this natural execution speed – across *all* applications in the entire workload. In contrast, a mechanism can maximize instruction throughput by unfairly slowing down low-IPC applications. We evaluate with weighted speedup to ensure our mechanism does not penalize in this manner.

Figure 10 shows weighted speedup improvements by up to 17.2% (18.2%) in the 4x4 and 8x8 workloads respectively.

**Fairness In Throttling:** We further illustrate that our mechanism does not unfairly throttle applications (i.e., that the mechanism is not biased toward high-IPF applications at the expense of low-IPF applications). To do so, we evaluate the performance of applications in pairs with IPF values *IPF1* and *IPF2* when put together in a 4x4 mesh (8 instances of each application) in a checkerboard layout. We then calculate the percentage change in throughput for both applications when congestion control is applied.

Figure 11 shows the resulting performance improvement for the applications, given the IPF values of both applications. Accompanying the graph is the average baseline (un-throttled) network utilization shown in Figure 12. Clearly, when both IPF values are high, there is no change in performance since both applications are CPU bound (network utilization is low). When application 2's IPF value (IPF2) is high and application 1's IPF value (IPF1) is low (right corner of both figures), throttling shows performance improvements to application 2 since the network is congested. Importantly, however, application 1 is *not* unfairly throttled (left corner), and in fact shows some improvements using our mechanism. For example, when $IPF1 = 1000$ and $IPF2 < 1$, application 1 still shows benefits (e.g., 5-10%) by reducing overall congestion.

**Key Findings:** *When evaluated in 4x4 and 8x8 networks, our mechanism improves performance up to 27.6%, reduces starvation, improves weighted speedup, and does not unfairly throttle.*

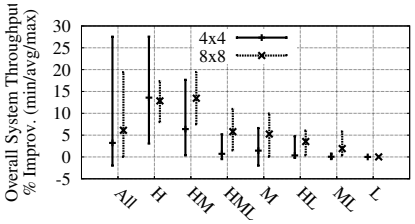



**Figure 8: Improvement breakdown by category.**

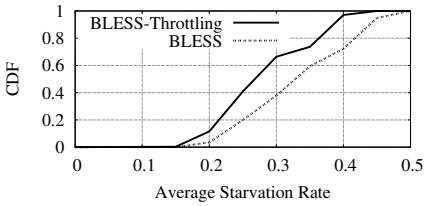



**Figure 9: CDF of average starvation rates.**

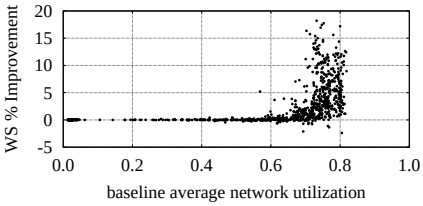



**Figure 10: Improvements in weighted speedup.**

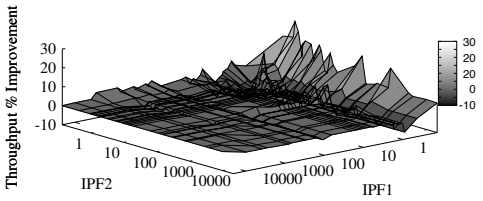



**Figure 11: Percentage improvements in throughput when shared.**

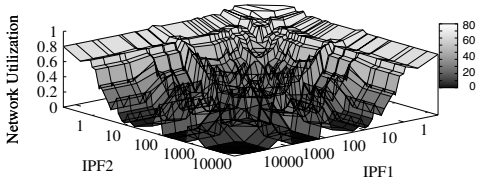



**Figure 12: Average baseline network utilization when shared.**

## 6.3 Scalability in Large NoCs

In §3.2, we showed that even with fixed data locality, increases in network sizes lead to increased congestion and decreased per-node throughput. We evaluate congestion control's ability to restore scalability. Ideally, per-node throughput remains fixed with scale.

We model network scalability *with data locality* using fixed exponential distributions for each node's request destinations, as in §3.2.[4] This introduces a degree of locality achieved by compiler and hardware optimizations for data mapping. Real application traces are still executing in the processor/cache model to generate the request timing; the destinations for each data request are simply mapped according to the distribution. This allows us to study scalability independent of the effects and interactions of more complex data distributions. Mechanisms to distribute data among multiple private caches in a multicore chip have been proposed [46, 57], including one which is aware of interconnect distance/cost [46].

Note that we also include a NoC based on *virtual-channel buffered routers* [14] in our scalability comparison.[5] A buffered router can attain higher performance, but as we motivated in §1, buffers carry an area and power penalty as well. We run the same workloads on the buffered network for direct comparison with a baseline bufferless network (BLESS), and with our mechanism (BLESS-Throttling).

Figures 13, 14, 15, and 16 show the trends in network latency, network utilization, system throughput, and NoC power as network size increases with all three architectures for comparison. The baseline case mirrors what is shown in §3.2: congestion becomes a scalability bottleneck as size increases. However, congestion control successfully throttles the network back to a more efficient operating point, achieving essentially flat lines. We observed the same scalability trends in a torus topology (however, note that the torus topology yields a 10% throughput improvement for all networks).

[4] We also performed experiments with powerlaw distributions, not shown here, which resulted in similar conclusions.

[5] The network has the same topology as the baseline bufferless network, and routers have 4 VCs/input and 4 flits of buffering per VC.

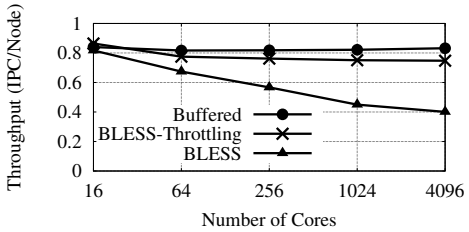



**Figure 13: Per-node system throughput with scale.**

We particularly note the NoC power results in Figure 16. This data comes from the BLESS router power model [20], and includes router and link power. As described in §2, a unique property of on-chip networks is a global power budget. Reducing power consumption as much as possible is therefore desirable. As our results show, through congestion control, we reduce power consumption in the bufferless network by up to 15%, and improve upon the power consumption of a buffered network by up to 19%.

## 6.4 Sensitivity to Algorithm Parameters

**Starvation Parameters:** The $\alpha_{starve} \in (0, \infty)$ parameter scales the congestion detection threshold with application network intensity, so that network-intensive applications are allowed to experience more starvation before they are considered congested. In our evaluations, $\alpha_{starve} = 0.4$; when $\alpha_{starve} > 0.6$ (which increases the threshold and hence under-throttles the network), performance decreases 25% relative to $\alpha_{starve} = 0.4$. When $\alpha_{starve} < 0.3$ (which decreases the threshold and hence over-throttles the network), performance decreases by 12% on average.

$\beta_{starve} \in (0, 1)$ controls the minimum starvation rate required for a node to be considered as starved. We find that $\beta_{starve} = 0.0$ performs best. Values ranging from 0.05 to 0.2 degrade performance by 10% to 15% on average (24% maximum) with respect to $\beta_{starve} = 0.0$ because throttling is not activated as frequently.

The upper bound on the detection threshold, $\gamma_{starve} \in (0, 1)$, ensures that even network-intensive applications can still trigger throttling when congested. We found that performance was not sensitive to $\gamma_{starve}$ because throttling will be triggered anyway by the less network-intensive applications in a workload. We use $\gamma_{starve} = 0.7$.

**Throttling Rate Parameters:** $\alpha_{throt} \in (0, \infty)$ scales throttling rate with network intensity. Performance is sensitive to this parameter, with an optimal in our workloads at $\alpha_{throt} = 0.9$. When $\alpha_{throt} > 1.0$, lower-intensity applications are over throttled: more than three times as many workloads experience performance loss with our mechanism (relative to not throttling) than with $\alpha_{throt} = 0.9$. Values below 0.7 under-throttles congested workloads.

The $\beta_{throt} \in (0, 1)$ parameter ensures that throttling has some effect when it is active for a given node by providing a minimum throttling rate value. We find, however, that performance is not sensitive to this value when it is small because network-intensive applications will already have high throttling rates because of their network intensity. However, a large $\beta_{throt}$, e.g. 0.25, over-throttles sensitive applications. We use $\beta_{throt} = 0.20$.

The $\gamma_{throt} \in (0, 1)$ parameter provides an upper bound on the

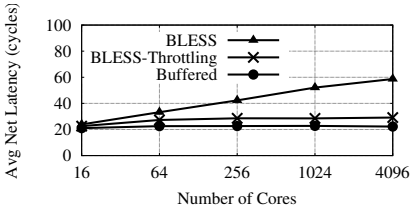



**Figure 14: Network latency with scale.**

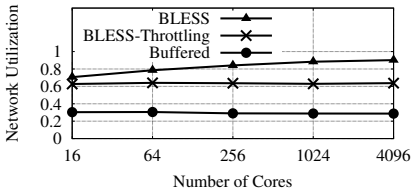



**Figure 15: Network utilization with scale.**

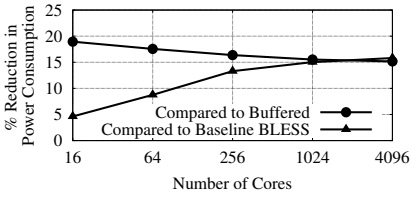



**Figure 16: Reduction in power with scale.**

throttling rate, ensuring that network-intensive applications will not be completely starved. Performance suffers for this reason if $\gamma_{throt}$ is too large. We find $\gamma_{throt} = 0.75$ provides the best performance; if increased to 0.85, performance suffers by 30%. Reducing $\gamma_{throt}$ below 0.65 hinders throttling from effectively reducing congestion.

**Throttling Epoch:** Throttling is re-evaluated once every 100k cycles. A shorter epoch (e.g., 1k cycles) yields a small gain (3–5%) but has significantly higher overhead. A longer epoch (e.g., 1M cycles) reduces performance dramatically because throttling is no longer sufficiently responsive to application phase changes.

## 6.5 Hardware Cost

Hardware is required to measure the starvation rate $\sigma$ at each node, and to throttle injection. Our windowed-average starvation rate over $W$ cycles requires a $W$-bit shift register and an up-down counter: in our configuration, $W = 128$. To throttle a node with a rate of $r$, we disallow injection for $N$ cycles every $M$, such that $N/M = r$. This requires a free-running 7-bit counter and a comparator. In total, only 149 bits of storage, two counters, and one comparator are required. This is a minimal cost compared to (for example) the 128KB L1 cache.

## 6.6 Centralized vs. Distributed Coordination

Congestion control has historically been distributed (e.g., TCP), because centralized coordination in large networks (e.g., the Internet) is not feasible. As described in §2, NoC global coordination is often less expensive because the NoC topology and size are statically known. Here we will compare and contrast these approaches.

**Centralized Coordination:** To implement centrally-coordinated throttling, each node measures its own IPF and starvation rate, reports these rates to a central controller by sending small control packets, and receives a throttling rate setting for the following epoch. The central coordinator recomputes throttling rates only once every 100k cycles, and the algorithm consists only of determining average IPF and evaluating the starvation threshold and throttling-rate formulae for each node, hence has negligible overhead (can be run on a single core). Only $2n$ packets are required for $n$ nodes every 100k cycles.

**Distributed Coordination:** To implement a distributed algorithm, a node must send a notification when congested (as before), but must decide on its own when and by how much to throttle. In a separate evaluation, we designed a simple distributed algorithm which (i) sets a "congested" bit on every packet that passes through a node when that node's starvation rate exceeds a threshold; and (ii) self-throttles at any node when that node sees a packet with a "congested" bit. This is a "TCP-like" congestion response mechanism: a congestion notification (e.g., a dropped packet in TCP) can be caused by congestion at any node along the path, and forces the receiver to back off. We found that because this mechanism is not selective in its throttling (i.e., it does not include application-awareness), it is far less effective at reducing NoC congestion. Alternatively, nodes could approximate the central approach by periodically broadcasting their IPF and starvation rate to other nodes. However, every node would then require storage to hold other nodes' statistics, and broadcasts would waste bandwidth.

**Summary:** Centralized coordination allows better throttling because the throttling algorithm has explicit knowledge of every node's state. Distributed coordination can serve as the basis for throttling in a NoC, but was found to be less effective.

# 7. DISCUSSION

We have provided an initial case study showing that core networking problems with novel solutions appear when designing NoCs. However, congestion control is just one avenue of networking research in this area, with many other synergies.

**Traffic Engineering:** Although we did not study multi-threaded applications in our work, they have been shown to have heavily local/regional communication patterns, which can create "hot-spots" of high utilization in the network. In fact, we observed similar behavior after introducing locality (§3.2) in low/medium congested workloads. Although our mechanism can provide small gains by throttling applications in the congested area, traffic engineering around the hot-spot is likely to provide even greater gains.

Due to application-phase behavior (Fig. 6), hot-spots are likely to be dynamic over run-time execution, requiring a dynamic scheme such as TeXCP [37]. The challenge will be efficiently collecting information about the network, adapting in a non-complex way, and keeping routing simple in the constrained environment. Our hope is that prior work can be leveraged, showing robust traffic engineering can be performed with fairly limited knowledge [4]. A focus could be put on a certain subset of traffic that exhibits "long-lived behavior" to make NoC traffic engineering feasible [58].

**Fairness:** While we have shown in §6.2 that our congestion control mechanism does not unfairly throttle applications to benefit system performance, our controller has no explicit fairness target. As described in §4, however, different applications have different rates of progress for a given network bandwidth; thus, explicitly managing fairness is a challenge. Achieving network-level fairness may not provide application level fairness. We believe the bufferless NoC provides an interesting opportunity to develop a novel application-aware fairness controller (e.g., as targeted by [10]).

**Metrics:** While there have been many metrics adopted by the architecture community for evaluating system performance (e.g., weighted speedup and IPC), more comprehensive metrics are needed for evaluating NoC performance. The challenge, similar to what has been discussed above, is that network performance may not accurately reflect system performance due to application-layer effects. This makes it challenging to know where and how to optimize the network. Developing a set of metrics which can reflect the coupling of network and system performance will be beneficial.

**Topology:** Although our study focused on the 2D mesh, a variety of on-chip topologies exist [11, 29, 40, 41, 43] and have been shown to greatly impact traffic behavior, routing, and network efficiency. Designing novel and efficient topologies is an on-going challenge, and resulting topologies found to be efficient in on-chip networks can impact off-chip topologies e.g., Data Center Networks (DCNs). NoCs and DCNs have static and known topologies, and are attempt-

ing to route large amounts of information multiple hops. Showing benefits of topology in one network may imply benefits in the other. Additionally, augmentations to network topology have gained attention, such as *express channels* [5] between separated routers.

**Buffers:** Both the networking and architecture communities continue to explore bufferless architectures. As optical networks [69, 71] become more widely deployed, bufferless architectures are going to become more important to the network community due to challenges of buffering in optical networks. While some constraints will likely be different (e.g., bandwidth), there will likely be strong parallels in topology, routing techniques, and congestion control. Research in this area is likely to benefit both communities.

**Distributed Solutions:** Like our congestion control mechanism, many NoC solutions use centralized controllers [16,18,27,65]. The benefit is a reduction in complexity and lower hardware cost. However, designing distributed network controllers with low-overhead and low-hardware cost is becoming increasingly important with scale. This can enable new techniques, utilizing distributed information to make fine-grained decisions and network adaptations.

# 8. RELATED WORK

**Internet Congestion Control:** Traditional mechanisms look to prevent *congestion collapse* and provide fairness, first addressed by TCP [35] (subsequently in other work). Given that delay increases significantly under congestion, it has been a core metric for detecting congestion in the Internet [35, 51]. In contrast, we have shown that in NoCs, network latencies remain relatively stable in the congested state. Furthermore, there is no packet loss in on-chip networks, and hence no explicit ACK/NACK feedback. More explicit congestion notification techniques have been proposed that use coordination or feedback from the network core [23, 38, 62]. In doing so, the network as a whole can quickly converge to optimal efficiency and avoid constant fluctuation [38]. However, our work uses application rather than network information.

**NoC Congestion Control:** The majority of congestion control work in NoCs has focused on buffered NoCs, and work with packets that have already entered the network, rather than control traffic at the injection point. The problems they solve are thus different in nature. Regional Congestion Awareness [27] implements a mechanism to detect congested regions in buffered NoCs and inform the routing algorithm to avoid them if possible. Some mechanisms are designed for particular types of networks or problems that arise with certain NoC designs: e.g., Baydal et al. propose techniques to optimize wormhole routing in [7]. Duato et al. give a mechanism in [18] to avoid head-of-line (HOL) blocking in buffered NoC queues by using separate queues. Throttle and Preempt [60], solves priority inversion in buffer space allocation by allowing preemption by higher-priority packets and using throttling.

Several techniques avoid congestion by deflecting traffic selectively (BLAM [64]), re-routing traffic to random intermediate locations (the Chaos router [44]), or creating path diversity to maintain more uniform latencies (Duato et al. in [24]). Proximity Congestion Awareness [52] extends a bufferless network to avoid routing toward congested regions. However, we cannot make a detailed comparison to [52] as the paper does not provide enough algorithmic detail for this purpose.

**Throttling-based NoC Congestion Control:** Prediction-based Flow Control [54] builds a state-space model for a buffered router in order to predict its free buffer space, and then uses this model to refrain from sending traffic when there would be no downstream space. Self-Tuned Congestion Control [65] includes a feedback-based mechanism that attempts to find the optimum throughput point dynamically. The solution is not directly applicable to our bufferless NoC problem, however, since the congestion behavior is different in a bufferless network. Furthermore, both of these prior works are application-unaware, in contrast to ours.

Adaptive Cluster Throttling [10], a recent source-throttling mechanism developed concurrently to our mechanism, is also targeted for bufferless NoCs. Unlike our mechanism, ACT operates by measuring application cache miss rates (MPKI) and performing a clustering algorithm to group applications into "clusters" which are alternately throttled in short timeslices. ACT is shown to perform well on small (4x4 and 8x8) mesh networks; we evaluate our mechanism on small networks as well as large (up to 4096-node) networks in order to address the scalability problem.

**Application Awareness:** Some work handles packets in an application aware manner in order to provide certain QoS guarantees or perform other traffic shaping. Several proposals, e.g., Globally Synchronized Frames [47] and Preemptive Virtual Clocks [30], explicitly address quality-of-service with in-network prioritization. Das et al. [16] propose ranking applications by their intensities and prioritizing packets in the network accordingly, defining the notion of "stall time criticality" to understand the sensitivity of each application to network behavior. Our use of the IPF metric is similar to L1 miss rate ranking. However, this work does not attempt to solve the congestion problem, instead simply scheduling packets to improve performance. In a later work, Aérgia [17] defines packet "slack" and prioritizes requests differently based on criticality.

**Scalability Studies:** We are aware of relatively few existing studies of large-scale 2D mesh NoCs: most NoC work in the architecture community focuses on smaller design points, e.g., 16 to 100 nodes, and the BLESS architecture in particular has been evaluated up to 64 nodes [50]. Kim et al. [42] examine scalability of ring and 2D mesh networks up to 128 nodes. Grot et al. [28] evaluate 1000-core meshes, but in a buffered network. That proposal, Kilo-NoC, addresses scalability of QoS mechanisms in particular. In contrast, our study examines congestion in a deflection network, and finds that reducing this congestion is a key enabler to scaling.

**Off-Chip Similarities:** The Manhattan Street Network (MSN) [48], an off-chip mesh network designed for packet communication in local and metropolitan areas, resembles the bufferless NoC in some of its properties and challenges. MSN uses drop-less deflection routing in a small-buffer design. Due to the routing and injection similarities, the MSN also suffers from starvation. Although similar in these ways, routing in the NoC is still designed for minimal complexity whereas the authors in [48] suggested more complex routing techniques which are undesirable for the NoC. Global coordination in MSNs were not feasible, yet often less complex and more efficient in the NoC (§2). Finally, link failure in the MSN was a major concern whereas in the NoC links are considered reliable.

**Bufferless NoCs:** In this study, we focus on bufferless NoCs, which have been the subject of significant recent work [10, 20–22, 26, 31, 49, 50, 67]. We describe some of these works in detail in §2.

# 9. CONCLUSIONS & FUTURE WORK

This paper studies congestion control in on-chip bufferless networks and has shown such congestion to be fundamentally different from that of other networks, for several reasons (e.g., lack of congestion collapse). We examine both network performance in moderately-sized networks and scalability in very large (4K-node) networks, and we find congestion to be a fundamental bottleneck. We develop an application-aware congestion control algorithm and show significant improvement in application-level system throughput on a wide variety of real workloads for NoCs.

More generally, NoCs are bound to become a critical system resource in many-core processors, shared by diverse applications.

Techniques from the networking research community can play a critical role to address research issues in NoCs. While we focus on congestion, we are already seeing other ties between these two fields. For example, data-center networks in which machines route packets, aggregate data, and can perform computation while forwarding (e.g., CamCube [1]) can be seen as similar to CMP NoCs. XORs as a packet coding technique, used in wireless meshes [39], are also being applied to the NoC for performance improvements [32]. We believe the proposed techniques in this paper are a starting point that can catalyze more research cross-over from the networking community to solve important NoC problems.

**Acknowledgements** We gratefully acknowledge the support of our industrial sponsors, AMD, Intel, Oracle, Samsung, and the Gigascale Systems Research Center. We also thank our shepherd David Wetherall, our anonymous reviewers, and Michael Papamichael for their extremely valuable feedback and insight. This research was partially supported by an NSF CAREER Award, CCF-0953246, and an NSF EAGER Grant, CCF-1147397. Chris Fallin is supported by an NSF Graduate Research Fellowship.

# 10. REFERENCES

[1] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. SIGCOMM, 2010.
[2] M. Alizadeh et al. Data center TCP (DCTCP). SIGCOMM 2010, pages 63–74, New York, NY, USA, 2010. ACM.
[3] Appenzeller et al. Sizing router buffers. *SIGCOMM*, 2004.
[4] D. Applegate and E. Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs. SIGCOMM 2003.
[5] J. Balfour and W. J. Dally. Design tradeoffs for tiled cmp on-chip networks. Proceedings of the international conference on Supercomputing (ICS) 2006.
[6] P. Baran. On distributed communications networks. *IEEE Trans. Comm.*, 1964.
[7] E. Baydal et al. A family of mechanisms for congestion control in wormhole networks. *IEEE Trans. on Dist. Systems*, 16, 2005.
[8] L. Benini and G. D. Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35:70–78, Jan 2002.
[9] S. Borkar.Thousand core chips: a technology perspective.*DAC-44*, 2007.
[10] K. Chang et al. Adaptive cluster throttling: Improving high-load performance in bufferless on-chip networks. SAFARI TR-2011-005.
[11] M. Coppola et al. Spidergon: a novel on-chip communication network. *Proc. Int'l Symposium on System on Chip*, Nov 2004.
[12] D. E. Culler et al. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.
[13] W. Dally. Virtual-channel flow control. *IEEE Par. and Dist. Sys.*, '92.
[14] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.
[15] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *DAC-38*, 2001.
[16] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. *MICRO-42*, 2009.
[17] R. Das et al. Aérgia: exploiting packet latency slack in on-chip networks. *International Symposium on Computer Architecture (ISCA)*, 2010.
[18] J. Duato et al. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. *HPCA-11*, 2005.
[19] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28:42–53, May 2008.
[20] C. Fallin, C. Craik, and O. Mutlu. CHIPPER: A low-complexity bufferless deflection router. *HPCA-17*, 2011.
[21] C. Fallin et al. A high-performance hierarchical ring on-chip interconnect with low-cost routers. SAFARI TR-2011-006.
[22] C. Fallin et al. MinBD: Minimally-buffered deflection routing for energy-efficient interconnect. *NOCS*, 2012.
[23] S. Floyd. Tcp and explicit congestion notification. *ACM Comm. Comm. Review, V. 24 N. 5, October 1994, p. 10-23.*
[24] D. Franco et al. A new method to make communication latency uniform: distributed routing balancing. *ICS-13*, 1999.
[25] C. Gómez, M. Gómez, P. López, and J. Duato. Reducing packet dropping in a bufferless noc. *EuroPar-14*, 2008.
[26] C. Gómez, M. E. Gómez, P. López, and J. Duato. Reducing packet dropping in a bufferless noc. *Euro-Par-14*, 2008.
[27] P. Gratz et al. Regional congestion awareness for load balance in networks-on-chip. *HPCA-14*, 2008.
[28] B. Grot et al. Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees. *ISCA-38*, 2011.
[29] B. Grot, J. Hestness, S. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. *HPCA-15*, 2009.
[30] B. Grot, S. Keckler, and O. Mutlu. Preemptive virtual clock: A flexible, efficient, and cost-effective qos scheme for networks-on-chip. *MICRO-42*, 2009.
[31] M. Hayenga et al. Scarab: A single cycle adaptive routing and bufferless network. *MICRO-42*, 2009.
[32] M. Hayenga and M. Lipasti. The NoX router. MICRO-44, 2011.
[33] Y. Hoskote et al. A 5-ghz mesh interconnect for a teraflops processor. *In the proceedings of IEEE Micro*, 2007.
[34] Intel. Single-chip cloud computer. `http://goo.gl/SfgfN`.
[35] V. Jacobson. Congestion avoidance and control. *SIGCOMM*, 1988.
[36] S. A. R. Jafri et al. Adaptive flow control for robust performance and energy. *MICRO-43*, 2010.
[37] S. Kandula et al. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. SIGCOMM 2005.
[38] D. Katabi et al. Internet congestion control for future high bandwidth-delay product environments. *SIGCOMM*, 02.
[39] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. Xors in the air: practical wireless network coding. SIGCOMM, 2006.
[40] J. Kim, W. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ISCA-35*, 2008.
[41] J. Kim et al. Flattened butterfly topology for on-chip networks. *IEEE Computer Architecture Letters*, 2007.
[42] J. Kim and H. Kim. Router microarchitecture and scalability of ring topology in on-chip networks. *NoCArc*, 2009.
[43] M. Kim, J. Davis, M. Oskin, and T. Austin. Polymorphic on-chip networks. *In the proceedings of ISCA-35*, 2008.
[44] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. *In the proceedings of ISCA-18*, 1991.
[45] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA Highly Scalable Server. *In the proceedings of ISCA-24*, 1997.
[46] H. Lee et al. CloudCache: Expanding and shrinking private caches. *In the proceedings of HPCA-17*, 2011.
[47] J. Lee et al. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. *ISCA-35*, 2008.
[48] N. Maxemchuk. Routing in the manhattan street network. *Communications, IEEE Transactions on*, 35(5):503 – 512, may 1987.
[49] G. Michelogiannakis et al. Evaluating bufferless flow control for on-chip networks. *NOCS-4*, 2010.
[50] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. *In the proceedings of ISCA-36*, 2009.
[51] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks.
[52] E. Nilsson et al. Load distribution with the proximity congestion awareness in a network on chip. *DATE*, 2003.
[53] G. Nychis et al. Next generation on-chip networks: What kind of congestion control do we need? *Hotnets-IX*, 2010.
[54] U. Y. Ogras and R. Marculescu. Prediction-based flow control for network-on-chip traffic. *DAC-43*, 2006.
[55] J. Owens et al. Research challenges for on-chip interconnection networks. *In the proceedings of IEEE Micro*, 2007.
[56] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. *MICRO-37*, 2004.
[57] M. Qureshi. Adaptive spill-receive for robust high-performance caching in CMPs. *HPCA-15*, 2009.
[58] A. Shaikh, J. Rexford, and K. G. Shin. Load-sensitive routing of long-lived ip flows. SIGCOMM 2009.
[59] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-9*, 2000.
[60] H. Song et al. Throttle and preempt: A new flow control for real-time communications in wormhole networks. *ICPP*, 1997.
[61] Standard Performance Evaluation Corporation. SPEC CPU2006. `http://www.spec.org/cpu2006`.
[62] I. Stoica et al. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. *In the proceedings of SIGCOMM*, 1998.
[63] M. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 2002.
[64] M. Thottethodi et al. Blam: a high-performance routing algorithm for virtual cut-through networks. *ISPDP-17*, 2003.
[65] M. Thottethodi, A. Lebeck, and S. Mukherjee. Self-tuned congestion control for multiprocessor networks. *HPCA-7*, 2001.
[66] Tilera. Announces the world's first 100-core processor with the new tile-gx family. `http://goo.gl/K9c85`.
[67] S. Tota, M. Casu, and L. Macchiarulo. Implementation analysis of noc: a mpsoc trace-driven approach. *GLSVLSI-16*, 2006.
[68] University of Glasgow. Scientists squeeze more than 1,000 cores on to computer chip. `http://goo.gl/KdBbW`.
[69] A. Vishwanath et al. Enabling a Bufferless Core Network Using Edge-to-Edge Packet-Level FEC. INFOCOM 2010.
[70] D. Wentzlaff et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.
[71] E. Wong et al. Towards a bufferless optical internet. *Journal of Lightwave Technology*, 2009.