Congestion Control for Scalability in Bufferless On-Chip Networks

George Nychis† Chris Fallin† Thomas Moscibroda§
gnychis@ece.cmu.edu cfallin@ece.cmu.edu moscitho@microsoft.com

Srinivasan Seshan† Onur Mutlu†
srini@cs.cmu.edu onur@cmu.edu

†Carnegie Mellon University §Microsoft Research

SAFARI Technical Report No. 2011-003

July 20, 2011

Abstract

In this paper, we present network-on-chip (NoC) design and contrast it to traditional network design, highlighting both similarities and differences between NoCs and traditional networks. As an initial case study, we examine *network congestion* in bufferless NoCs. We show that congestion manifests itself differently in a NoC than in a traditional network. This both reduces system throughput in congested workloads for smaller NoC sizes (16 and 64 nodes), and limits the scalability of the bufferless NoC in larger configurations (256 to 4096 nodes) even when data is mapped with locality in mind. We propose a source throttling-based congestion control mechanism with application-level awareness. This mechanism improves system performance by up to 28% (15% on average in congested workloads) in smaller NoCs, and achieves linear throughput scaling in NoCs up to 4096 cores. 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.

1 Introduction

One of the most important trends in computer architecture in recent years 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 [6, 39].[1] While this trend has helped address several key roadblocks in computer architecture (e.g. power dissipation and single-core complexity), it has created many possible new ones. One particular new challenge is the design of the interconnect between cores. Since this interconnect carries all inter-cache and memory traffic, it plays a critical role in both CMP performance and efficiency.

Unfortunately, the traditional bus-based and other centrally-controlled designs used on small-CMPs do not scale to the large or medium scale CMPs that are in development. As a result, the architecture research community is moving away from traditional centralized interconnect scheduling, to distributed scheduling and routing [9], both on 2D meshes and over a variety of non-uniform topologies [19, 28]. The resulting designs are far more network-like than traditional processor interconnects and are described as a network-on-chip (NoC). These designs must deal with many network-like problems such as interconnect scalability, routing [35], congestion, and prioritization [10, 11, 20] that have traditionally not been studied in the architecture community.

While not like traditional processor interconnects, these NoCs are also not like existing networks or even like the traditional multi-chip interconnects used in large-scale multiprocessors [7, 32]. 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

[1] Intel has built their Single-chip Cloud Computer CMP with 48 cores [23], research chips with 80 cores [22] exist, one company has announced a 100-core processor [49], and most recently a 1,000 core research system has been developed [50].

scale NoCs with buffers [35], use sophisticated routing and arbitration [8], and over-provision the network. Although large-scale supercomputers have used processor-memory interconnects for many years (e.g., the SGI Origin [32]), the tradeoffs in terms of latency, power, and complexity are different and network design considerations change as a function of the on-chip environment, as we will discuss in § 3.3. These and other characteristics give NoCs a unique flavor, and have important ramifications on solutions to traditional networking problems in a novel context.

In this paper, we explore the adaptation of network techniques and methodolgy for addressing two particular issues in next-generation bufferless NoC design: *congestion management* and *scalability*. Bufferless NoCs have recently gained serious consideration in the architecture community due to chip area and power constraints[2]. While the bufferless NoC has been shown to operate efficiently under moderate workloads and limited network sizes (16 and 64 cores) [35], we find that with higher-intensity workloads and larger network sizes (e.g., 256 to 4096 cores), the network operates inefficiently and does not scale effectively. As a consequence, application-level system performance can suffer heavily. One work has proposed to solve this issue by simply switching to a buffered mode when congestion occurs [25]. However, we wish to study the causes of inefficiency in bufferless NoCs and retain their advantages if possible.

Through evaluation, we find that *congestion* is a major factor in limiting the efficiency and scalability of the bufferless NoC. Unlike traditional networks, congestion is experienced in a fundamentally different way due to both unique NoC properties and bufferless properties. While traditional networks suffer from congestion collapse at high utilization, a NoC does not collapse due to the self-throttling nature of cores (i.e., the pipeline stalls). However, congestion causes the system to run sub-optimally, and increasingly inefficiently with scale, due to unique bufferless properties such as deflections and starvation (inability to inject in the network).

We develop a novel congestion-control mechanism suited to the unique properties of the NoC and bufferless routing. By monitoring the starvation of the cores, we can detect impending congestion in the network more effectively than by monitoring network latency. Because different applications respond very differently to congestion and increases/decreases in network throughput, the network must be application-aware. We define an application-level metric which can be estimated in the network, and allows for proper source throttling to alleviate congestion. By controlling congestion, we improve system performance, and allow the network to scale more effectively. We make the following contributions:

- We present the **key differences** of the NoC and bufferless NoCs from traditional networks to guide our study and future networking research in NoCs.
- 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 application-level throughput. In large networks, even with *locality* (i.e., mapping an application's data nearby in the network), the per-node performance reduces as the network becomes more congested at larger design points.
- We propose a novel low-complexity and high-performance congestion control mechanism in a bufferless NoC that is motivated by ideas from both networking and computer architecture. To our knowledge, this is the first work that examines congestion in bufferless NoCs and provides an effective solution.
- Using comprehensive evaluations with a large number of real application workloads, we begin by evaluating improved network efficiency for small (4x4 and 8x8) bufferless NoCs in which all nodes access data in shared cache slices across the chip. We show our mechanism improves system performance by up to 28% (19%) in a 16-core (64-core) system with a 4x4 (8x8) mesh NoC, and improves performance by 15% on average in congested workloads.
- Focusing on larger networks (in our case, 256 to 4096 cores), we evaluate the role of congestion control in conjunction with locality-aware data mapping to provide scalability in per-node performance. We show that even when data is mapped locally (near its owning application) in the cache slices, congestion control is necessary to achieve scalability in a bufferless NoC. At the high end (4096 cores), congestion control yields a 50% per-node throughput improvement over the locality-mapped baseline, yielding a linear system throughput trend (constant per node) as network size increaess.

[2]Existing prototypes show that NoCs can consume a substantial portion of system power (30% in the Intel 80-core Terascale chip [22], 40% in the MIT RAW chip [46]).

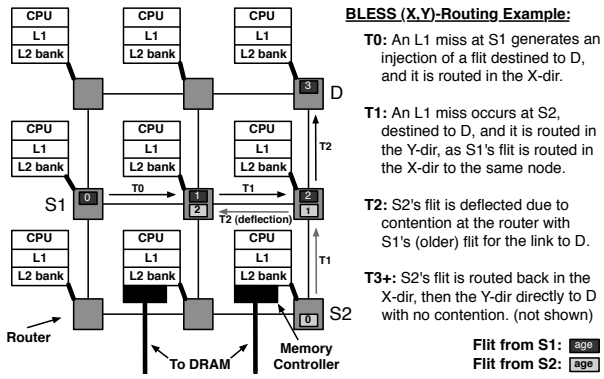


Figure 1: 9 core CMP architecture with BLESS routing example.

2 Background

We first provide a brief background on on-chip architectures and bufferless NoCs. We refer the reader to [5, 8] for an in-depth discussion. Section 3 describes key characteristics of on-chip networks that are different from traditional networks.

2.1 NoCs in Multi-Core Architectures

In a chip multiprocessor (CMP) architecture, the NoC generally connects the processor nodes and their private caches with the shared cache banks and memory controllers (see Figure 1). A NoC might also carry other control traffic, such as interrupt requests, but it primarily exists to service cache miss requests. In this architecture, a high-speed router exists at each node, which connects the core to its neighbors by links. The width of a link varies, but 128 bits is a typical value. Nodes interchange packets, which correspond to a cache coherence protocol specific to the implementation; typical packets are small request and control messages, such as cache block read requests, and larger data packets containing cache block data. Packets are partitioned into *flits*, units that are the width of a link and thus serve as the atomic unit of traffic. Links typically have a latency of only one or two cycles, and are pipelined, so that they can accept a new flit every cycle.

A variety of on-chip topologies have been proposed in the literature (e.g., [19, 27, 28, 30]), but the most typical topology is the two-dimensional (2D) Mesh [8], which is implemented in several commercial [49, 51] and research prototype [22, 23, 46] many-core processors. In this topology, each router has 5 input and 5 output channels/ports; one from each neighbor and one from the network interface (NI). Furthermore, depending on the router architecture and the arbitration policies (i.e., the number of pipelined arbitration stages), each packet spends between 1 cycle (in a highly optimized best case [35]) and 4 cycles at each router before being forwarded to the next link.

Because router complexity is a critical design consideration in on-chip networks, current implementations tend to use simple routing algorithms. The most common routing paradigm is x-y routing, i.e., a flit is first routed along the x-direction until the destination's y-coordinate is reached; then routed to the destination in y-direction.

2.2 Bufferless NoCs and Routing

The question of buffering 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, including for example data center networks [2]. The same discussions are ongoing also in on-chip networks [13, 25, 34, 35]. Specifically, recent work has shown that it is possible to completely eliminate buffers from the routers of on-chip networks routers. In such *bufferless NoCs*, application performance degrades minimally for low-to-moderate network intensity workloads, while some work shows that power consumption decreases by 20-40%, router area on die is reduced by 75%, and implementation complexity also decreases [13, 35]. While other evaluations have shown that optimizations to traditional buffered-router designs can make buffers more area- and energy-efficient [34], bufferless design techniques such as those in [13] address inefficiencies in bufferless design. In a bufferless NoC, the general system architecture does not differ

from traditional buffered NoCs. However, the lack of buffers requires different injection and routing algorithms in the network. 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* [35]) happen synchronously across all cores on 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. A flit is then routed to a neighbor based on the routing algorithm (X,Y-Routing in our example), and the arbitration policy. With no buffers, flits must pass through the router pipeline without stalling or waiting. *Deflection* is used to resolve port-contention when two or more flits request the same output port.

Flits are arbitrated to output ports based on direction and age through the *Oldest-First* arbitration policy [35]. 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* [8]) – 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 [21] drop packets under contention, this design does not, and therefore ACKs are not needed. Despite simplicity, the policy is very efficient in terms of performance, and is livelock-free [35].

Note that many past systems have used this type of deflection routing (it is frequently known as *hot-potato routing* [3]). However, it is particularly well-suited for NoCs, and in this context, presents a set of challenges distinct from those in traditional networks.

3 Characteristics of NoCs

With an understanding of NoC and bufferless NoC design, an important question that remains is: in what sense do on-chip networks differ from other types of networks? These differences provide insight into what makes a NoC interesting from a *networking research* point of view. The unique properties of NoCs guide our study, inform our understanding of scalability, and guide the design of our congestion control mechanism. We present key properties of both general and bufferless NoCs, and also contrast them with off-chip processor-memory interconnects as used in large multiprocessors.

3.1 NoC Architecture Properties

First, we provide unique characteristics of general NoC architecture as compared to traditional networks. Such characteristics are driven by program behavior and the first-class considerations in chip design: chip area/space considerations, implementation complexity, and power.

- **Topology:** The topology is statically known, and usually very regular (e.g., a mesh). A change in topology will impact various aspects, such as routing and traffic-load.
- **Latency:** Links and (heavily-pipelined) routers have latency much lower than traditional networks: 1-2 cycles.
- **Routing:** Arbitration and routing logic are designed for minimal complexity and low latency, because these router stages typically must take no more than a few cycles.
- **Coordination:** Global coordination and network-wide optimizations, at least at a coarse grain, are possible and often less expensive than truly distributed adaptive mechanisms, due to a relatively small known topology, and low latency. Note that fine-grained control (e.g., packet routing) must remain a truly local decision. At a scale of thousands of cycles or more, however, a central controller can feasibly observe the network state and adjust the system accordingly.
- **Links:** Links are expensive, both in terms of hardware complexity and on-chip area. Therefore, links cannot easily be overprovisioned like in other types of networks.
- **Latency vs. Bandwidth:** This tradeoff is very different in NoCs. Low latency is important for efficient operation, and typically the allowable window of in-flight data is much smaller than in a large-scale network.
- **Network Flows:** Because many architectures will split the shared cache across several or all nodes in the system, a program will typically send traffic to many nodes, often in parallel. Multithreaded programs also exhibit complex communication patterns. There, the concept of a "network flow" is removed or greatly diminished.

- **Traffic Patterns:** Private cache miss behavior of applications, including locality-of-reference, phase behavior with local and temporal bursts, and importantly, self-throttling [8], drive traffic patterns in a NoC.
- **Throughput:** NoCs lack a direct correlation between network throughput, and overall system throughput. As we will show (§5), for the same network throughput, changing which L1 cache misses are serviced in the network can change system throughput (measured as instructions per cycle per node) by up to 18%.
- **Power:** Because NoCs physically reside on one chip, the entire network shares a power and thermal budget. Moreover, for CMPs, the NoC should not be the primary consumer of power: it is desirable to leave most of the power budget for cores, cache slices, and other components that more directly affect system performance. The existence of a constrained power budget distinguishes NoCs from traditional networks.

3.2 Bufferless NoC Architecture Properties

Bufferless NoCs have *all* of the unique characteristics found in general NoC architecture, as well as its own set of unique properties. Such properties are driven by the routing, arbitration, and lack of buffers:

- **Loss:** Given that a packet can only be injected if there is at least one free output port, and is otherwise guaranteed a link once in the network, the network is dropless.
- **Retransmission:** Without packet loss, there is no need for a retransmission scheme. Once a packet enters the network, it is guaranteed livelock-free delivery [35]. If packet loss or corruption does occur, it is due to catastrophic failure (e.g. a bit-flip or other transient fault) and is not ordinarily within the scope of the network.
- **(N)ACKs:** In a loss-free network, ACKs or NACKs are not needed, which would only utilize scarce link resources.
- **In-Network Latency:** In-network latency in a bufferless NoC is very stable and low, even under high congestion with deflections (§4). Flits are quickly routed in the network, without incurring delay in router buffers.
- **Injection Latency:** Unlike in traditional networks, the injection latency (time from head-of-queue to entering the network) can be significant (§4). Without a free output link, the design will prevent a core from injecting. Note that in the worst case, this can lead to starvation, which is a fairness issue. (We will show in §7 that our mechanism addresses this concern.)

3.3 On-Chip vs. Off-Chip Interconnect

A large body of work studies interconnects in large supercomputer or cluster-computer arrangements. Typically, systems using these networks fall into two categories: cache-coherent shared memory systems (also known as ccNUMA, or cache-coherence non-uniform memory architecture [7, 32]), and message-passing systems [1]. The application domain is limited to highly-parallel problems, because while many-core systems are motivated by physical scaling trends in chip manufacturing, the main reason to build out a parallel system across many individual processors is to enable fine-grained parallel computation. Additionally, on-chip placement yields new constraints for NoCs that are not present in large off-chip interconnects for parallel systems. On-chip and off-chip interconnects can be compared as follows:

- **Latency:** On-chip and off-chip interconnect must provide very low latency relative to traditional networks (e.g. local or global IP networks) because they often form the backbone of a memory system or lie on the critical path of communication-intensive parallel applications. For example, Myrinet and Infiniband, two common off-chip supercomputer interconnects, achieve latencies less than 10μs, an order of magnitude smaller than TCP for small messages [15]. However, on-chip networks typically provide latencies lower still than this: even a 100-cycle cross-chip latency corresponds to only 50 ns on a 2 GHz processor.
- **Power:** Off-chip interconnect design is typically less concerned with network power than on-chip design for two reasons. The first is environment: off-chip interconnects are used mainly in supercomputing applications where power is a secondary concern to performance, and matters only insofar as cooling and packaging remains feasible. The second is more fundamental: on-chip interconnect exists on a single chip with a global power budget. Additionally, NoCs exist at a smaller scale: while link power is still significant, it does not dominate network power, and so router power is a more significant component of total power and a greater target for optimization.
- **Complexity:** On-chip interconnect topology is more limited by wire-routing and layout than off-chip interconnect. Clusters of supercomputer nodes can be arranged in three dimensions, and flexible fiber or copper cables can form

relatively complex topologies. In contrast, flattening a more complex topology onto a 2D chip is often difficult. Folded tori [8] and flattened butterfly [28] NoCs have been proposed, but hypercubes (such as the Cosmic Cube [41] off-chip interconnect) are more difficult on chip. Additionally, router complexity is a greater concern in on-chip networks due to both tighter latency constraints (e.g., only a few cycles to make each routing decision) and limited area for control logic. This can limit both router radix, with implications for more complex topologies, and the routing schemes used.

- **Traffic patterns:** Traffic patterns seen in on-chip and off-chip interconnects can sometimes differ. Supercomputer applications running on large systems with off-chip interconnects are often designed with communication cost in mind. In the ideal case, computation will have as much locality as possible, and only truly necessary coordination will use the interconnect. In addition, nodes in a system with off-chip interconnect almost always have a local memory hierarchy that is used by the local computations – e.g., local memory controllers and DRAM banks [7]. In contrast, on-chip networks typically serve as the shared-memory backbone to a general-purpose multicore or manycore CPU, and many shared-memory applications are interconnect-agnostic. Even when the operating system and/or applications are NUMA (non-uniform memory access) aware and attempt to minimize interconnect cost, on-chip designs place more burden on the interconnect by routing all accesses to the last-level cache and to memory controllers over the network.

4 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 [13, 35] 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 load in conventionally sized (4x4 and 8x8) NoCs. Higher levels of network load remain a challenge, and improving performance in these cases increases the applicability of the efficiency gains. Furthermore, as the size of the CMP increases (e.g., to 64x64), these efficiency gains from bufferless NoCs will become increasingly important. However, we will show that new scalability challenges arise with larger network sizes that must be managed in an intelligent way.

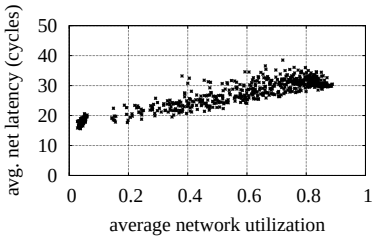
We explore the limitations of bufferless NoCs from these dimensions – network load and network size – with the goal of understanding *scalability* and *efficiency* of the bufferless NoC. First, in § 4.1, we show that as network workload increases, application-level throughput reduces due to congestion in the network. This congestion manifests itself differently than in traditional buffered networks by *starving* cores from injecting traffic in to the network: that is, admission is a greater bottleneck than in-network deflection. In § 4.2, 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 with data locality (i.e., intelligent data mapping to cache slices), we find that the congestion significantly reduces the scalability of the bufferless NoC. These two fundamental motivations lead to *congestion control for bufferless NoCs*.

4.1 Limitation of Workload Intensity

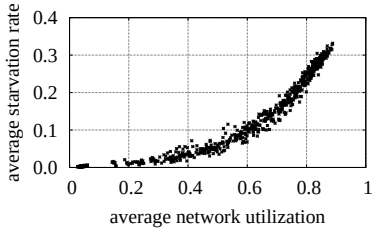
We begin by studying the effects of high workload intensity in the bufferless NoC. To do so, we simulate 700 real-application workloads in a 4x4 NoC (see methodology in §7.1). Our workloads span a range of network utilizations exhibited by real applications; by sweeping over workload intensity, we hope to understand the impact of network load on both network and application-layer performance.

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 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 §4.2, network latency increases more with load in larger NoCs as other scalability bottlenecks come into consideration.

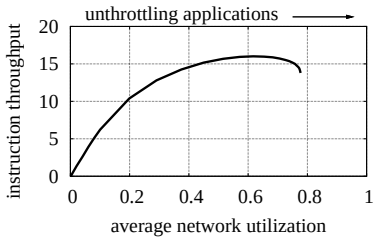
Deflection routing shifts many effects of congestion from within the network to network admission. In a highly-congested network, it may no longer be possible to efficiently inject packets into the network, because the router encounters free slots less often. Such a situation is known as **starvation**. We define **starvation rate** (σ) as the fraction of cycles (in some window of W cycles) in which a node tries to inject a flit but cannot:



(a) Avg. net. 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.

$$\sigma = \frac{1}{W}\sum_{i}^{W} starved(i) \in [0,1] \quad (1)$$

Figure 2(b) shows that starvation rate grows superlinearly with network utilization. The starvation rates at higher network utilizations are significant. Near 80% utilization, the average core in our 4x4 network is blocked from injecting a flit 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 § 6, we will use this metric to drive decisions in controlling network usage.

Effect of Congestion on Application-level Throughput: As a NoC is an integral component of a complete multicore system, it is important to evaluate the effect of congestion at the application layer. In other words, network-layer effects matter only insofar as 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 the application-level throughput, we take a network-heavy sample workload and *throttle* all applications at a throttling rate swept up from 0. This throttling rate controls how often a router that desires to inject a flit is blocked from doing so. Throttling a fixed workload thus allows us to vary the network utilization over a continuum and observe the full range of network congestion. Figure 2(c) plots the resulting system throughput as a function of average network 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 running on cores are naturally *self-throttling*: A thread running on a core can only inject a relatively small number of requests into the network before stalling to wait for the missing replies. Once stalled, a thread cannot inject further requests. This self-throttling nature of applications helps to prevent *congestion collapse*, even at the highest possible load in the network.

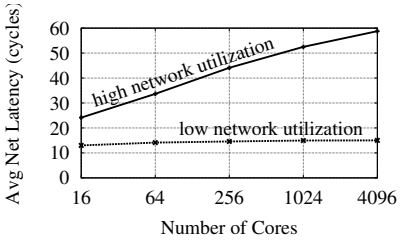
Second, and importantly for congestion control, this experiment shows that injection throttling can yield increased application-level throughput, even though it explicitly blocks injection some fraction of the time, because it reduces network congestion significantly. In Figure 2(c), a gain of 14% is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

However, static and homogeneous throttling across all cores does not yield the best possible improvement. In fact, as we will show in §5, throttling the wrong applications can significantly reduce system performance. This will

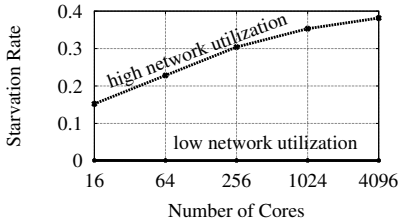
motivate the need for *application-awareness*. Dynamically throttling the proper applications yields more significant system throughput improvements (e.g., up to 28% improvement as seen in §7), based on their relative benefit they attain from injecting into the network. This is a key insight to our congestion control mechanism.

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

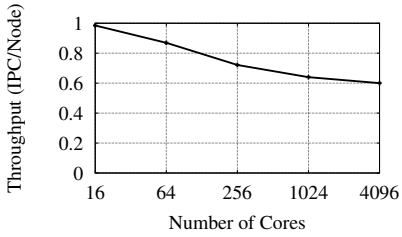
4.2 Limitation of Network Size



(a) Average network latency under heavy load increases with the CMP size.



(b) Starvation rate significantly increases under congestion as the size of the CMP increases.



(c) The increased effects of congestion as the CMP size increases, limits the scalability of the NoC.

Figure 3: Scaling behavior: as network size increases, effect of congestion becomes more severe and scalability is limited.

As we motivated in the previous 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 [13, 35] to understand the scalability bottlenecks. In doing so, we model data locality (i.e., intelligent data mapping) in the shared cache slices: the simple assumption of uniform data striping across all nodes no longer makes sense at large scales. Indeed, taking locality as a first step is critical: with simple uniform striping (no locality), we find that per-node throughput degrades by 73% from a 4x4 network to a 64x64 network with the same per-application workload intensity.

In order to model locality in a reasonable way, independent of particular cache or memory system implementation details, we assume an exponential distribution of data-request destinations around each node. In other words, the private-cache misses from a given CPU core choose shared-cache slices to service their data requests with an exponential distribution, so that most of the cache misses are serviced by nodes within a few hops, and some small fraction of requests go much 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.

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 its 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, the impact of congestion becomes increasingly severe. 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 (λ parameter) – in other words, despite the same average destination distance. Likewise, shown in Figure 3(b), the impact of 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 the size of the CMP 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 (per-node application throughput) 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 §4.1 and Figure 2(c), reducing congestion in the network improves system performance. As we will show through the introduction of a novel congestion control mechanism and evaluation in §7, reducing the congestion in the network will significantly improve the scalability of the bufferless NoC with high intensity workloads.

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 λ 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.

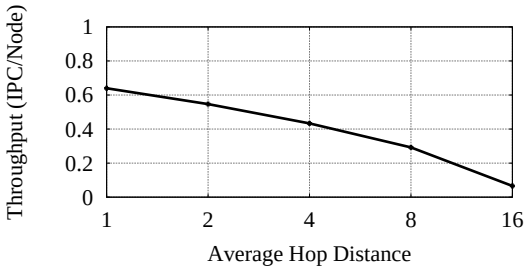


Figure 4: Sensitivity of system throughput (IPC per node) to degree of locality, as modeled by a simple exponential distribution.

Key Findings: *We find that 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 to enable efficient scaling.*

5 Need for Application Awareness

Application-level throughput decreases as network congestion increases. Therefore, as is done in traditional networks, one possible solution is that applications can be throttled to reduce the level of congestion in the network. However, *which applications* we throttle can significantly impact per-application and overall system performance. To illustrate this, we have constructed a workload in a 4×4-mesh NoC that consists of 8 instances each of `mcf` and `gromacs`, which are memory-intensive and non-intensive benchmarks, respectively [44]. We run the workload in the baseline configuration, with no throttling, and then statically throttle each application in turn by 90% (injection blocked 90% of the time) and examine per-application and overall system throughput in each case.

The results provide key insights (Fig. 5). First, *which application is throttled has a significant impact on overall system throughput*. When `gromacs` is throttled, the overall system throughput drops 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`, overall system throughput 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

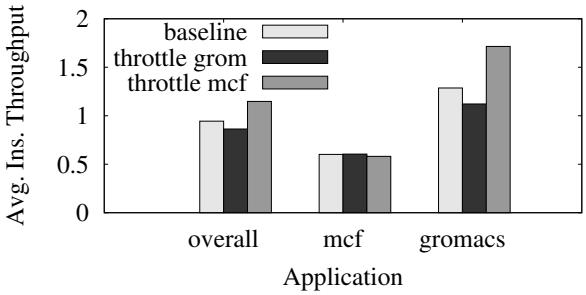


Figure 5: System throughput with selective throttling.

from the 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 different L1 cache miss rate, and thus requires a certain volume of traffic to retire 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 have as much relative impact on forward progress as the same delay of a flit in an application with few L1 misses, since that flit represents a greater fraction of forward progress in the latter application. Such awareness has been leveraged in buffered NoCs [10].

Key Finding: *Bufferless NoC congestion control needs application-layer awareness to determine whom to throttle.*

Instructions-per-Flit: The above discussion implies that not all flits are created equal. 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: $\textbf{IPF} = I/F$. For a given code sequence, set of inputs, and system parameters, IPF is a fixed value that depends only on the L1 cache miss rate. It is independent of the congestion in the network and the rate of execution of the application, and is thus a stable measure in a shared system. Table 1 shows that IPF values (for a set of SPEC CPU2006 benchmarks [44]) can vary considerably: `mcf`, a memory-intensive benchmark produces slightly less than 2 flits of traffic for every instruction retired (IPF=0.58), whereas `povray` yields an IPF of over 1000, more than 2000 times greater. The latency of a single flit in this high-IPF application thus has greater impact on performance.

Benchmark	*IPF*	*Benchmark*	*IPF*	*Benchmark*	*IPF*
mcf	0.583	omnetpp	3.150	wrf	69.75
leslie3d	0.814	cactusADM	4.905	sjeng	134.15
soplex	1.186	bzip2	6.281	gcc	155.18
libquantum	1.252	astar	6.376	namd	168.08
lbm	1.429	hmmer	9.362	calculix	253.23
milc	1.751	gromacs	12.41	tonto	256.53
GemsFDTD	2.267	h264ref	14.64	perlbench	425.19
sphinx3	2.253	dealII	37.99	povray	1189.8
xalancbmk	2.396	gobmk	60.73		

Table 1: IPF (Instructions-per-Flit) values for our set of workloads.

Fig. 5 illustrates this: `mcf`'s low IPF value (0.583) indicates that it can be heavily throttled with little impact on its throughput (-3% @ 90% throttling). It also gains little from additional network throughput (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%).

Key Finding: *The IPF metric enables application-awareness and can inform per-application throttling decisions.*

6 Congestion Control Mechanism

Section 5 defined a metric that determines an application's network intensity and its response to throttling. As shown in 5, *when the network is congested*, we must consider application-layer information to throttle effectively. We improve instruction throughput by throttling applications with low IPF (high network intensity). This works for three reasons: 1) applications with low IPF are relatively insensitive to throttling compared to applications with high IPF, 2) conversely, applications with high IPF benefit more at the application-level from increased network throughput than those with low IPF, and 3) throttling applications with low IPF is more effective at reducing overall congestion, because these applications are more network-intensive. In order to determine whom to throttle, we need only a *ranking* of intensity.

Basic Idea: We propose an interval-based congestion control algorithm that periodically (every $100,000$ cycles): 1) detects congestion based on starvation rates in the network, 2) determines IPF of applications, 3) if the network is congested, throttles the appropriate applications based on the IPF metric. Our algorithm, described in this section, is summarized in Algorithms 1, 2, and 3.

Controller Mechanism: A *key difference* of this mechanism to the majority of currently existing congestion control mechanisms in traditional networks [24, 26] is that it is a *centrally-coordinated* algorithm. This is possible in an on-chip network, and in fact is cheaper in our case (Section 7.4). Since the on-chip network exists within a CMP that usually runs a single operating system (i.e., 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 the most efficient approach. Because the mechanism is periodic with a relatively long period, this does not place an undue burden on the system's CPUs. As described in detail in Section 7.4, 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 ranking the nodes accordingly based on their estimated IPF values; and third, it must determine how much to throttle in order to optimize system throughput without destroying the performance of any individual application. We will address each of these elements in turn, and then present a complete algorithm.

6.1 When to Throttle

As described in §4, *starvation rate* is a superlinear function of network congestion (Fig. 2(b)). We use starvation rate (σ) as a per-node indicator of congestion in the network. Node i is *congested* if:

$$\sigma_i > \min(\beta_{starve} + \alpha_{starve}/IPFe_i, \gamma_{starve}) \tag{2}$$

where α is a scale factor, and β and γ are lower and upper bounds, respectively, on the threshold (we use $\alpha_{starve} = 0.2$, $\beta_{starve} = 0.35$ and $\gamma_{starve} = 0.8$ in our evaluation, determined empirically). It is important to factor in IPF since network-intensive applications will naturally have higher starvation due to higher injection rates. Note that we use an IPF *estimate*, $IPFe$, based on *injection queue length*, since queue length increases as starvation (due to congestion) increases. Finally, throttling is *active* if at least one node is congested. Active throttling mode picks only certain nodes to throttle, and scales throttling rate according to intensity.

6.2 Whom to Throttle

When throttling is active, a node is throttled if its intensity is above average. This is determined by observing IPF: lower IPF indicates greater network intensity, and so nodes with IPF below average are throttled. Since we run a central coordination algorithm, knowing the mean of all queue lengths is possible without any sort of distributed averaging or estimation. The *Throttling Criterion* is:

If *throttling is active* AND $IPF_i < mean(IPF)$.

The simplicity of this rule can be justified by our observation that IPF in most workloads tend to be fairly widely distributed: there are memory-intensive applications and CPU-bound applications. We find that in most cases, the

separation between application classes is clean, and so the additional complexity of a more intelligent rule is not justified.

Algorithm 1 Main Control Algorithm (in software)

Every T cycles:
collect $queue\_len[i]$, $\sigma[i]$ from each node i

/* *determine congestion state* */
$congested \Leftarrow false$
for $i = 0$ to $N_{nodes} - 1$ **do**
  $thresh = \min(\alpha_{starve} * queue\_len[i] + \beta_{starve}, \gamma_{starve})$
  if $\sigma[i] > thresh$ **then**
    $congested \Leftarrow true$
  end if
end for

/* *set throttling rates* */
$Qthresh = mean(queue\_len)$
for $i = 0$ to $N_{nodes} - 1$ **do**
  if $congested$ AND $queue\_len[i] > Qthresh$ **then**
    $throttle\_rate[i] = \min(\alpha_{thr} * queue\_len[i] + \beta_{thr}, \gamma_{thr})$
  else
    $throttle\_rate[i] \Leftarrow 0$
  end if
end for

Algorithm 2 Starvation Rate Computation (in hardware)

At node i:
$\sigma[i] \Leftarrow \sum_{k=0}^{W} starved(current\_cycle - k)/W$

Algorithm 3 Injection Throttling (in hardware)

At node i:
if trying to inject in this cycle and an output link is free **then**
  $inj\_count[i] = (inj\_count[i] + 1) \bmod MAX\_COUNT$
  if $inj\_count[i] > throttle\_rate[i] * MAX\_COUNT$ **then**
    allow injection
    $starved(current\_cycle) \Leftarrow false$
  else
    block injection
    $starved(current\_cycle) \Leftarrow true$
  end if
end if

6.3 Determining Throttling Rate

We throttle the chosen set of applications proportional to their application intensity. The *throttling rate*, the fraction of cycles in which a node cannot inject, is computed as follows:

$$R = \min(\beta_{rate} + \alpha_{rate}/IPF, \gamma_{rate}) \tag{3}$$

where IPF is used as a measure of application intensity, and α, β and γ set the scaling factor, lower bound and upper bound respectively, as in the starvation threshold formula above. Empirically, we determine $\alpha_{rate} = 0.30$, $\beta_{rate} = 0.45$ and $\gamma_{rate} = 0.75$ work well, and are used in our evaluation.

Network topology	2D mesh, 4x4 or 8x8 size
Routing algorithm	FLIT-BLESS [35] (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 interleave, XOR mapping; randomized exponential for locality/scalability evaluations

Table 2: System Parameters for Evaluation

6.4 Implementation: Estimating IPF

In order to use IPF as a driving metric in our throttling algorithm, we need to have a way to compute it easily in hardware. Fortunately, the design of the CPU core and bufferless NoC router allows for another measurement that corresponds to the inverse of IPF, or FPI (Flits per Instruction): the **length of the request queue (QueueLength)** of the core. The request queue is the injection queue into the network that buffers outstanding flits that are waiting to be injected. Its length correlates with FPI because 1) the outstanding requests at any given time correspond to cache misses from the current instruction window (a fixed-size buffer that bounds the number of pending instructions), and 2) in a congested network, most time is spent waiting in the queue to inject, and so the queue length is representative of the number of outstanding requests. The correlation is not exact, but we find in practice that using the average queue length to rank applications in IPF order works very well. Figure 6 shows the correlation between QueueLength and IPF for our set of workloads. (The outlier at $FPI = 0.3$ is `omnetpp`, which exhibits high average queue length due to bursty behavior.)

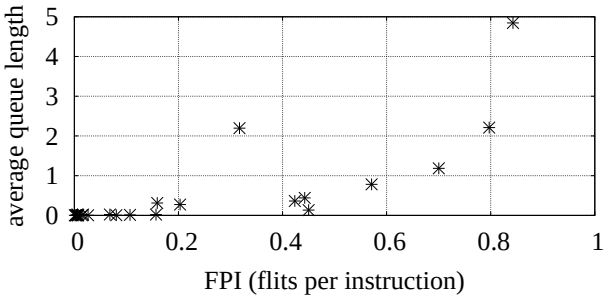


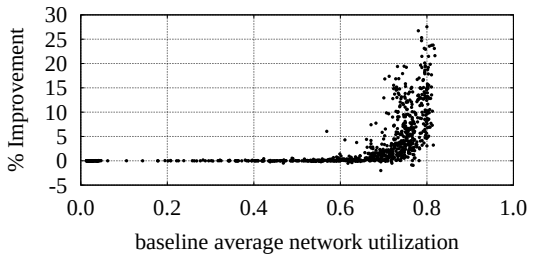
Figure 6: Correlation between FPI (Flits-per-Instruction) and average queue length for our set of benchmarks.

7 Evaluation

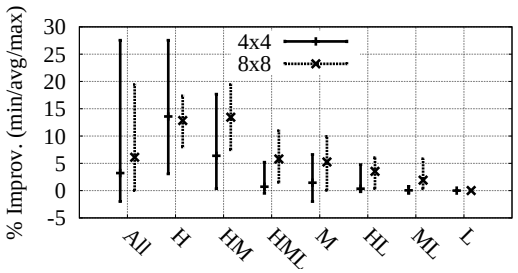
We evaluate the effectiveness of our congestion-control mechanism to address both the high-load problem in small NoCs (4x4 and 8x8) and the scalability problem in large NoCs (up to 64x64). We show results for the former in § 7.2 and the latter in § 7.3.

7.1 Methodology

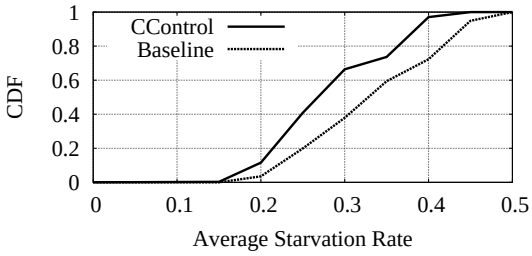
We use a *closed-loop* model of a complete network-processor-cache system, so that the system is *self-throttling* as in a real multi-core system (parameters in Table 2). In other words, by modeling the instruction windows and tracking outstanding cache-miss requests as in a real system, we capture system behavior more accurately. For each application,



(a) 4x4 and 8x8 All Workloads.



(b) Workload Category Breakdown.



(c) CDF of Starvation Rates.

Figure 7: Percentage improvement in overall system throughput and starvation provided by our mechanism for *all* workloads (4x4 & 8x8).

we capture an instruction trace of a representative execution slice (chosen using PinPoints [40]) and replay each trace in its respective CPU core model during simulation. All workloads are run for 500K cycles; although this run length is very short by architecture-community standards, we find that because the only state is in small private caches and in the interconnect, the system warms to a steady state quickly. The short run-length makes evaluations at our very large design points (4096 nodes) possible.

Workloads and Their Characteristics: We evaluate 875 multiprogrammed workloads (700 16-core, 175 64-core), where each workload is a multiprogrammed combination of applications, which is a realistic expectation for large CMPs (e.g., cloud computing [23] might place many applications on one substrate). These applications include SPEC CPU2006 [44], a standard benchmark suite in the architecture community, as well as various desktop, workstation, and server applications. We classify the applications (Table 1) into three categories based on their IPF values: H=Heavy, M=Medium, L=Light and systematically ensure a balanced set of multiprogrammed workloads, which is important for evaluation of many-core systems (e.g., for cloud computing [23]). To do this, seven categories are created based on randomly mixing applications of specific intensities: $\{H,M,L,HML,HM,HL,ML\}$.

Congestion Control Parameters: We empirically determined that the following control algorithm parameters work well: We set the update period to $T = 10K$ cycles and the starvation computation window to $W = 128$. The minimum and maximum starvation rate thresholds are $\beta_{starve} = 0.35$ and $\gamma_{starve} = 0.8$ with a scaling factor of $\alpha_{starve} = 50$. We set the throttling minimum and maximum to $\beta_{throttle} = 0.45$ and $\gamma_{throttle} = 0.75$ with scaling factor $\alpha_{throttle} = 0.2$.

7.2 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 §4.1) 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%, and average improvement over these workloads is 14.7%.

Fig. 7(b) 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

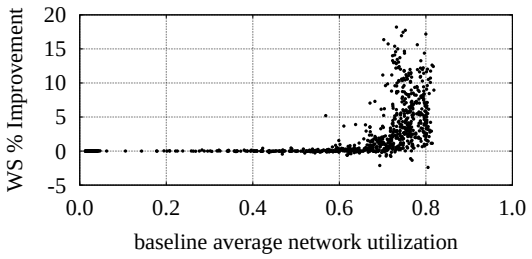


Figure 8: Percentage improvements in weighted speedup.

High/ Medium intensity. As expected, our mechanism provides little to no improvement when all applications in the workload have Low or Medium/Low intensity, because in such cases, the network is adequately provisioned for the demanded load.

Improvement in Network-level Admission: Fig. 7(c) shows the cumulative distribution function 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: In addition to instruction throughput, a common metric for evaluation is *weighted speedup* [42], 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 multiprogrammed workload. In contrast, a mechanism can maximize instruction throughput by unfairly slowing down low-IPC applications. To ensure that our congestion control mechanism does not significantly penalize applications in this manner, we evaluate it using the weighted speedup metric.

Figure 8 shows weighted speedup improvements by up to 17.2% (18.2%) in the 4x4 and 8x8 workloads respectively. Our proposed mechanism improves weighted speedup by up to 17.2% (18.2%) in the 4x4 (8x8) systems respectively.

Key Findings: *Overall, when evaluated in 4x4 and 8x8 networks over a range of workload intensities, our mechanism improves performance up to 27.6%, reduces starvation, and improves weighted speedup.*

7.3 Scalability in Large NoCs

In § 4.2, we showed that even with fixed data locality, increases in network sizes lead to increased congestion and decreased system throughput. Here, we evaluate the ability of congestion control to alleviate this congestion and restore scalability. Ideally, per-node throughput remains fixed as the network size increases. We show that this is the case.

We model network scalability simply with fixed exponential distributions for each node's request destinations, as in § 4.2. Note that 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 model allows us to study scalability independently of the effects and interactions that more complex data distributions might have.

Figures 9, 10, 11, and 12 show the trends in network latency, network utilization, instruction throughput (IPC) and NoC power per node as network size increases, both with and without our congestion-control mechanism active. The baseline case mirrors what is shown in § 4.2: 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 curves for all relevant metrics per node.

We particularly note the per-node NoC power results in Figure 12. This data comes from the BLESS router power model used in [13], and includes both router and link power. As described briefly in § 3, a unique property of on-chip networks is that a global power budget exists. Reducing power consumption as much as possible, or at least not

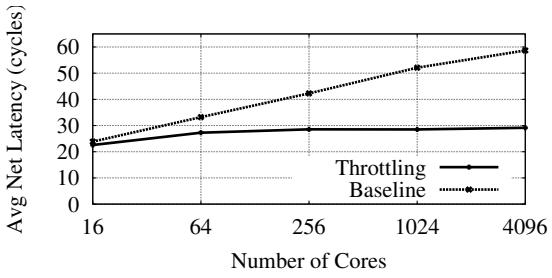


Figure 9: Average network latency with scale.

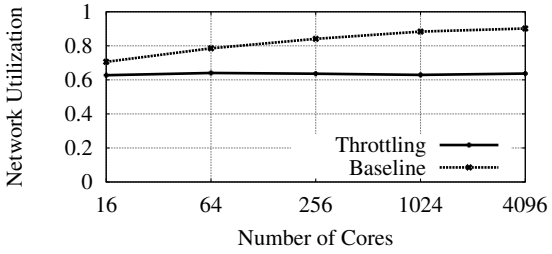


Figure 10: Average network utilization with scale.

increasing the per-node power cost, is therefore desirable. As our results show, power per node remains constant.[3] Note that router design for low power is outside the scope of this work; we simply maintain constant per-node power with scalability by removing the congestion bottleneck.

Finally, we note that the scalability of the NoC is fundamentally dependent on the nature of the workload's traffic distribution. In this simple scalability analysis, we assume a fairly tight exponential distribution ($\lambda = 1.0$). If a node's data distribution spreads (i.e., loses locality) to a sufficient degree, scalability in per-node throughput is lost, despite congestion control, because the bisection bandwidth of the network becomes the limiting factor. In other words, locality is a necessary condition for scalability. Congestion control simply enables the network to efficiently handle workloads at large scales, given that the workload has sufficient traffic locality.

7.4 Hardware Cost

Hardware Implementation: Hardware is required to measure the starvation rate σ 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 rate r, we disallow injection for N cycles every M, such that $N/M = r$. This requires a free-running counter and a comparator; 7 bits provides sufficient granularity. In total, this implementation requires only 149 bits of storage, two counters, and one comparator: a minimal cost per-router compared to (for example) the 128KB L1 cache.

8 Related Work

8.1 Internet Congestion Control

Congestion control in traditional buffered networks is extremely well studied. Traditional mechanisms look to prevent the *congestion collapse* problem first addressed by TCP [24] (and subsequently in many other works), which can cause the throughput of all network flows to sharply drop. One difference between these protocols and our proposed on-chip mechanism is the way congestion is detected. Given that delay increases significantly under congestion, it has been a core metric for detecting congestion in the Internet [24, 36], and latency has been successfully used to

[3] We note that for a manycore system with 1K or 4K nodes to be feasible, several technology generations will likely pass, reducing system power by a constant factor. Thus, the large total power implied by this model will scale down and should be dwarfed by core and cache power.

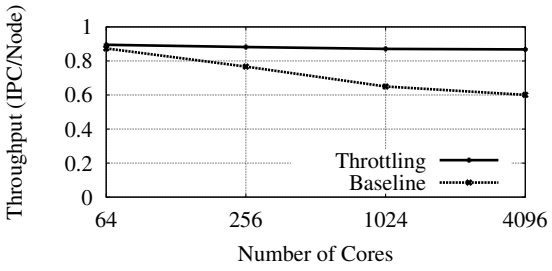


Figure 11: Average system throughput with scale.

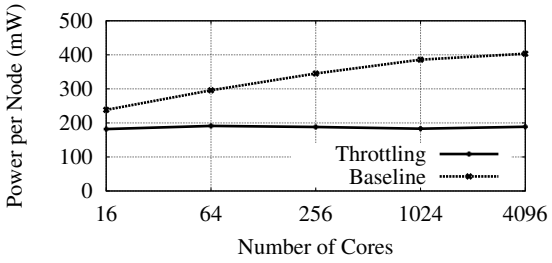


Figure 12: Average NoC power per node with scale.

distributedly detect congestion. 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 core of the network [14, 26, 45], so that the network as a whole can more quickly converge to optimal efficiency and avoid constant fluctuation created by dropped packets [26]. Our work is different in that it uses application-level information, rather than information at the network-level (see Section 5), to improve system throughput and reduce starvation in the on-chip environment.

8.2 NoC Congestion Control

There has been an expanding body of recent work in providing congestion control or prioritization in buffered NoCs. The majority of these proposals focus 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. However, there is potential to combine prioritization with admission-based congestion control. Several proposals, including Globally Synchronized Frames [33] and Preemptive Virtual Clocks [20], implement global frameworks for packet prioritization based on batching and network-level quality of service metrics. In these cases, prioritization serves to provide hard or soft guarantees to applications rather than to optimize system performance under network-intensive workloads. However, the two goals of congestion control and quality-of-service could be combined under a single framework. Regional Congestion Awareness [18] 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 particular problems that arise with certain NoC designs: for example, Baydal et al. propose several techniques to optimize wormhole routing in [4]. Duato et al. give a mechanism in [12] to avoid HOL blocking in buffered NoC queues by using separate queues. Regional Explicit Congestion Notification [17] also aims to alleviate HOL blocking in queues. Another mechanism, Throttle and Preempt [43], solves priority inversion in buffer space allocation by allowing preemption by higher-priority packets and using throttling to create buffer space for such packets.

Several techniques avoid congestion by deflecting traffic selectively (BLAM [48]), or re-routing traffic to random intermediate locations (the Chaos router [31]), or creating path diversity to maintain more uniform latencies (Duato et al. in [16]). We are aware of one congestion control mechanism for bufferless NoCs: Proximity Congestion Awareness [37] extends a bufferless network to avoid routing toward congested regions. However, PCA is very light on algorithmic details and so we cannot make a detailed comparison.

8.3 Throttling-based Approaches

A few congestion control approaches in buffered NoCs work by throttling sources to control network load. Prediction-based Flow Control [38] 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 [47] performs throttling to optimize network throughput. The novelty in that proposal is a feedback-based mechanism to find the optimum throughput point dynamically. The solution is not applicable to our bufferless NoC problem, however, since the congestion behavior is different. Also, our approach incorporates application-awareness into throttling, unlike all these previous works.

8.4 Application Awareness

We are aware of two explicitly application-aware proposals for performance optimization in a NoC. Das et al. [10] 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 the use of L1 miss rate ranking in [10]. However, Das et al.'s scheme is a packet scheduling algorithm for buffered networks that does not (attempt to) solve the problem of network congestion. In a later work, Aérgia [11], Das et al. define packet "slack" and prioritize each request differently based on its criticality.

8.5 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, on the order of 16 to 100 nodes, and the BLESS architecture in particular has been evaluated up to 64 nodes [35]. Kim et al. [29] examine scalability of ring and 2D mesh networks up to 128 nodes. No work of which we are aware has evaluated scalability of bufferless 2D meshes up to 4096 nodes.

9 Summary & Conclusions

This paper studies congestion control in on-chip bufferless networks and has shown such congestion to be fundamentally different from those in other networks (e.g., due to lack of congestion collapse). We examine network performance under both high application load and as the network scales, and find congestion to be the fundamental bottleneck in both cases. 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 on-chip networks from 16 to 4096 nodes. More generally, NoCs are bound to become a critical system resource in many-core processors, shared by diverse applications. Finding solutions to networking problems in NoCs is paramount to effective many-core computing, and we believe the networking research community can and should weigh in on these challenges.

References

[1] Open MPI high-performance message passing library. `http://www.open-mpi.org/`.
[2] Appenzeller et al. Sizing router buffers. *SIGCOMM*, 2004.
[3] P. Baran. On distributed communications networks. *IEEE Trans. on Comm.*, 1964.
[4] E. Baydal, P. Lopez, and J. Duato. A family of mechanisms for congestion control in wormhole networks. *IEEE Trans. on Par. and Dist. Sys.*, 16, 2005.
[5] L. Benini and G. D. Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35:70–78, Jan 2002.
[6] S. Borkar. Thousand core chips: a technology perspective. *DAC-44*, 2007.
[7] D. E. Culler et al. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.
[8] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.
[9] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *DAC-38*, 2001.
[10] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. *MICRO-42*, 2009.
[11] R. Das et al. Argia: exploiting packet latency slack in on-chip networks. *ISCA*, 2010.
[12] J. Duato, I. Johnson, J. Flich, F. Naven, P. Garcia, and T. Nachiondo. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. *HPCA-11*, 2005.
[13] C. Fallin, C. Craik, and O. Mutlu. CHIPPER: A low-complexity bufferless deflection router. *HPCA-17*, 2011.

[14] S. Floyd. Tcp and explicit congestion notification. *ACM Comm. Comm. Review, V. 24 N. 5, October 1994, p. 10-23.*
[15] N. C. for Supercomputing Applications. Latency results from Pallas MPI benchmarks. `http://vmi.ncsa.uiuc.edu/performance/pmb_lt.php`.
[16] D. Franco et al. A new method to make communication latency uniform: distributed routing balancing. *ICS-13*, 1999.
[17] P. Garcia et al. Efficient, scalable congestion management for interconnection networks. *IEEE MICRO*, 26, 2006.
[18] P. Gratz, B. Grot, and S. W. Keckler. Regional congestion awareness for load balance in networks-on-chip. *HPCA-14*, 2008.
[19] B. Grot, J. Hestness, S. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. *HPCA-15*, 2009.
[20] 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.
[21] M. Hayenga et al. Scarab: A single cycle adaptive routing and bufferless network. *MICRO-42*, 2009.
[22] Y. Hoskote et al. A 5-ghz mesh interconnect for a teraflops processor. *IEEE MICRO*, 2007.
[23] Intel Corporation. Single-chip cloud computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.
[24] V. Jacobson. Congestion avoidance and control. *SIGCOMM*, 1988.
[25] S. A. R. Jafri et al. Adaptive flow control for robust performance and energy. *MICRO-43*, 2010.
[26] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for future high bandwidth-delay product environments. *SIGCOMM*, 02.
[27] J. Kim, W. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ISCA-35*, 2008.
[28] J. Kim et al. Flattened butterfly topology for on-chip networks. *IEEE Computer Architecture Letters*, 2007.
[29] J. Kim and H. Kim. Router microarchitecture and scalability of ring topology in on-chip networks. *NoCArc*, 2009.
[30] M. Kim, J. Davis, M. Oskin, and T. Austin. Polymorphic on-chip networks. *ISCA-35*, 2008.
[31] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. *ISCA-18*, 1991.
[32] J. Laudon and D. Lenoski. The sgi origin: a ccnuma highly scalable server. *ISCA-24*, 1997.
[33] J. Lee, M. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. *ISCA-35*, 2008.
[34] G. Michelogiannakis et al. Evaluating bufferless flow-control for on-chip networks. *NOCS*, 2010.
[35] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. *ISCA-36*, 2009.
[36] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks.
[37] E. Nilsson et al. Load distribution with the proximity congestion awareness in a network on chip. *DATE*, 2003.
[38] U. Y. Ogras and R. Marculescu. Prediction-based flow control for network-on-chip traffic. *DAC-43*, 2006.
[39] J. Owens et al. Research challenges for on-chip interconnection networks. *IEEE MICRO*, 2007.
[40] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. *MICRO-37*, 2004.
[41] C. Seitz. The cosmic cube. *CACM*, 28, Jan 1985.
[42] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-9*, 2000.
[43] H. Song et al. Throttle and preempt: A new flow control for real-time communications in wormhole networks. *ICPP*, 1997.
[44] Standard Performance Evaluation Corporation. SPEC CPU2006. `http://www.spec.org/cpu2006`.
[45] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. *SIGCOMM*, 1998.
[46] M. Taylor, J. Kim, J. Miller, and D. Wentzlaff. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE MICRO*, Mar 2002.
[47] M. Thottethodi, A. Lebeck, and S. Mukherjee. Self-tuned congestion control for multiprocessor networks. *HPCA-7*, 2001.
[48] M. Thottethodi, A. Lebeck, and S. Mukherjee. Blam: a high-performance routing algorithm for virtual cut-through networks. *ISPDP-17*, 2003.
[49] Tilera Corporation. Tilera announces the world's first 100-core processor with the new tile-gx family. `http://www.tilera.com/news_&_events/press_release_091026.php`.

[50] University of Glasgow. Scientists squeeze more than 1,000 cores on to computer chip. http://www.gla.ac.uk/news/headline_183814_en.html.

[51] D. Wentzlaff et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.