Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?

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

ABSTRACT

In this paper, we present network-on-chip (NoC) design and contrast it to traditional network design, highlighting core 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, and with application-level awareness in the network to make proper throttling decisions we improve system performance by up to 28%. It is our hope that the unique and interesting challenges of on-chip network design can be met by novel and effective solutions from 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

General Terms

Design, Algorithms, Performance

Keywords

On-chip networks, multi-core, congestion control

1. INTRODUCTION

One of the most important developments in computer science and computer architecture in recent years is the trend towards ever larger *multi-core processors* to overcome the diminishing performance returns of designing increasingly complex single-core processors. In a multi-core chip, efficient communication between the various components on the chip (cores, on-chip cache banks, DRAM memory controllers, accelerators, etc) is critical to performance. Many current multi-core systems are small (2-8 cores), and hence, a simple common bus suffices to connect the various components. However, chips with hundreds or thousands of cores are likely to be commonplace [5, 19].[1] Connecting so many cores with a bus is not scalable because 1) electrical loading on the bus significantly reduces its operational speed, and 2) the shared bus cannot support the bandwidth demand.

Networks on Chip: The solution to scalably and efficiently connect on-chip components is a packet-switched *on-chip network* [7, 3, 4]. A NoC consists of a high-speed router at each node, connected by links to its neighbors. Although the NoC may carry various kinds of traffic (e.g. interrupt requests), its most important purpose is to service cache miss requests. At the heart of this line of research are micro-architectural questions such as router/link design or efficient topologies. However, as we discuss in this paper, some of the key problems in on-chip networks are in fact *networking problems*, rather than architectural problems.

NoC Characteristics: NoCs feature a number of special characteristics: high performance demands, coupled with hardware implementation constraints, lead to a different trade-off space for NoCs compared to most traditional off-chip networks. NoCs run at higher utilization, and traffic patterns do not exhibit flow character (such as in the Internet), but are characterized by the self-throttling nature of *applications* on the various cores. Aspects such as chip area/space, power consumption,[2] and implementation complexity (e.g. the expense of arbitration and routing logic) are first-class considerations. These and other characteristics lend *on-chip networks* an interesting and unique flavor, and have important ramifications on the resulting networking solutions.

Bufferless NoC: The question of how much buffer space each router should have has been hotly debated in our community (Internet [1], data center networks [17], etc). In on-chip networks, a similar discussion has recently occurred. Here, a paradigm shift towards smaller buffers is based on the observation that buffers in network routers are expensive in terms of energy consumption (buffers consume signifi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Hotnets '10, October 20–21, 2010, Monterey, CA, USA.
Copyright 2010 ACM 978-1-4503-0409-2/10/10 ...$10.00.

[1]Research chips with 80 cores [12] already exist, one company recently announced a 100-core processor [23], and Intel has announced their *Single-chip Cloud Computer* [13] to service a cloud with 48 cores.

[2]Existing prototypes show that NoCs can consume a substantial portion of system power (30% in the Intel 80-core Terascale chip [12], 40% in the MIT RAW chip [22]).

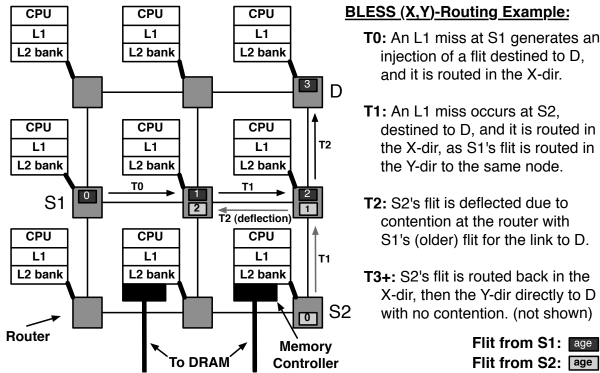


Figure 1: 9 core CMP architecture with BLESS routing example.

cant static and dynamic energy [12]), area (75% of the NoC area is consumed by buffers in the TRIPS prototype [9]), and hardware complexity (buffer allocation/deallocation). Entirely *bufferless* routing architectures have been proposed and evaluated where local and temporary traffic overload in the network is handled purely by means of deflection routing (also called hot-potato routing [2]). The resulting small degradation of network throughput results in reduction of energy consumption, router complexity, and on-chip area [18].

Contributions: We first outline characteristic features of on-chip networks, and discuss their impact on the resulting networking problems. We then study congestion control in bufferless NoCs and show that such networks experience congestion in fundamentally different ways than traditional networks which requires different application-aware metrics must be applied. We develop and evaluate a new congestion control algorithm, yielding up to 28% (19%) improvement in a 16-core (64-core) mesh NoC for a rich set of workloads.

2. BACKGROUND

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 the 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. 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.

A variety of on-chip topologies have been proposed in the literature (e.g., [16, 15, 10]), but the most typical topology is the 2D mesh [6], which is implemented in several commercial [24, 23] and research prototype [22, 12, 13] 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 [18]) and 4 cycles at each router before being forwarded to the next link.

Bufferless NoCs and Routing: The question of buffering is central to networking [1]. In NoCs, it was shown to be feasible to go completely bufferless and eliminate buffers from NoC routers: application performance degrades minimally, while power consumption reduces by 20-40%, on-chip area can be reduced by 75%, and implementation complexity also reduces [18]. The general system architecture does not differ from buffered NoCs. However there are two key differences in the NoC design: a lack of buffers at the routers, and consequently, injection and routing algorithms in the network. Fig. 1 gives an example of *injection*, *routing* and *arbitration*.

Like in a buffered NoC, injection and routing in a bufferless NoC (*BLESS*, for short) 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 [18]. 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* [6]) – 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 [11] 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 [18].

Bufferless routing by itself is not novel: It is frequently known as *hot-potato routing* [2]. However, it is particularly 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, and helps to guide the design of our congestion control mechanism. We present key properties of both general and bufferless NoCs.

NoC Architecture Properties: Several characteristics are driven by chip area/space considerations, implementation complexity, and program behavior:

- **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 are possible and often less expensive due to a relatively small known topology, and low latency.
- **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 all nodes a program will typically send traffic to all nodes. Multithreaded programs also exhibit complex communication patterns. There, the concept of a "network flow" is removed.
- **Traffic Patterns:** Private cache miss behavior of applications, including locality-of-reference, phase behavior with local and temporal bursts, and importantly, self-throttling, drive traffic patterns in a NoC.
- **Throughput:** NoCs lack a direct correlation between network throughput, and overall system throughput. As we will show (§4.2), for the same network throughput, changing which L1 cache misses are serviced in the network can change system throughput by up to 18%.

Bufferless NoC Architecture Properties: in addition to the above, the bufferless NoC we study has unique properties driven by the routing, arbitration, and the 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 drop-less.
- **Retransmission:** Without packet loss, there is no need for a retransmission scheme. Once a packet enters the network, it is guaranteed live-lock free delivery [18].
- **(N)ACKs:** In a dropless 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.

4. CONGESTION IN BUFFERLESS NOCS

The different NoC properties result in many challenges with regard to classical networking problems in NoCs. One particularly important and interesting such networking problem is *congestion control for bufferless NoCs*. Intuitively, bufferless routing seems a promising design for moderate-to-low network utilizations; however, a bufferless network saturates more quickly than a buffered NoC. Here, we show that congestion has fundamentally different consequences in NoCs and that application-aware solutions are required.

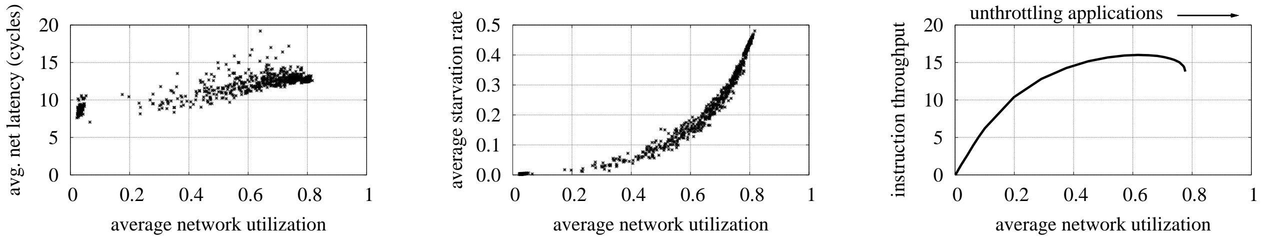
4.1 Effect of Congestion

In order to understand the effect of congestion in a bufferless NoC, we explore how congestion affects system performance at both the network and application levels. We simulate 700 real-application workloads in a 4x4 NoC (see methodology in §6) while examining application behavior. Our workloads span a range of network utilizations exhibited by real applications to demonstrate the network and system performance under various levels of congestion.

Effect of Congestion at the Network-Level: Figure 2(a) shows average network latency in each of the workloads. Notice how per-flit network latency generally remains stable, 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. The reason is that in bufferless NoCs, deflection routing shifts the effect of congestion from within the network to network admission: Due to high congestion, it may no longer be possible to efficiently *inject* packets into the network, but once injected, every packet will reach its destination without too much delay. Thus, *network latency is not the proper metric* for detecting congestion and should not be the target of improvement under congestion. Instead, *injection starvation* is the critical measure. Starvation occurs when injection is blocked by high network utilization, because in-flight traffic takes precedence in the router. We define the *starvation rate* at a given node as the fraction of cycles in which injection is blocked in this way: $\sigma = \frac{1}{C}\sum_i^C starved(i) \in [0,1]$. Figure 2(b) shows that starvation rate grows superlinearly with network utilization.

Effect of Congestion on Application-level Throughput: Given that the NoC is an integral component of a complete multicore system, it is important to evaluate the effect of congestion across the layers, specifically at the application layer. 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/unthrottle all applications, to observe the full range of network congestion. Fig. 2(c) plots the resulting system throughput as a function of average network utilization. The key observation is that 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



(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.**

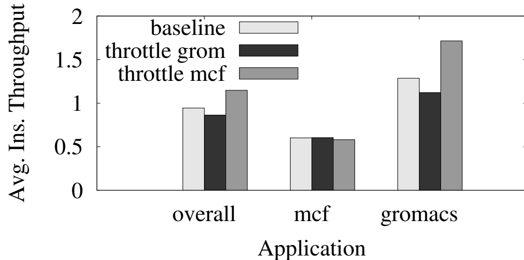


Figure 3: **System throughput with selective throttling.**

thread cannot inject further requests. This self-throttling nature of applications, in combination with stable in-network packet latencies, implies that bufferless NoCs do not exhibit a *congestion collapse* even at the highest possible load.

While the consequences are not catastrophic (e.g., collapse), static and homogeneous throttling across all cores does not yield the best possible improvement. As we will show, more significant system throughput improvements can be achieved by dynamically throttling specific applications based on the relative benefit they get from injecting into the network. This is the key to our mechanism.

Key Findings: *Starvation rate, not network latency, indicates congestion. Letting applications run freely is suboptimal, albeit will not cause congestion collapse.*

4.2 Need for Application Awareness

Application-level throughput decreases as network congestion increases. Therefore, like in traditional networks, applications should 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 [21]. We run the workload with no throttling, and then throttle each application in turn by 90% (injection blocked 90% of the time), and examine per-application and overall system throughput.

The results provide key insights (Fig. 3). 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 determinant 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 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 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 [8].

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: **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 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 [21]) 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.

Fig. 3 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 gains from additional network throughput (+25%).

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

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.

Network topology	2D mesh, 4x4 or 8x8 size
Routing algorithm	FLIT-BLESS [18] (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

Table 2: System Parameters for Evaluation

5. CONGESTION CONTROL MECHANISM

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. A *key difference* to existing congestion control mechanisms (say, TCP or XCP [14]) is that ours is *centrally coordinated*. This is in fact is cheaper in NoCs: a distributed algorithm would require more complex computations and indirect information gathering, whereas a central controller reduces redundancy and uses directly-tracked statistics to compute IPF.

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:

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

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.

Whom to Throttle: When throttling is active, a node is throttled if its intensity (as by IPF) is above average. 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.

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:

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

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.

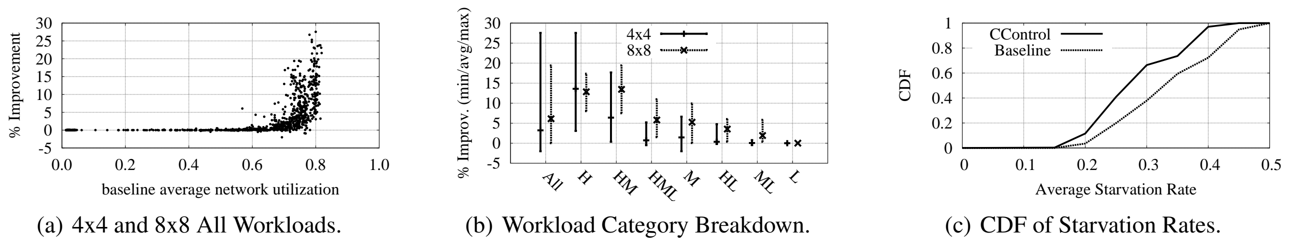
6. EVALUATION

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). We evaluate 875 multiprogrammed workloads (700 16-core, 175 64-core), used in desktop, workstation, and scientific computing and commonly used in the architecture community for evaluation [21]. 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 [13]). To do this, seven categories are created based on randomly mixing applications of specific intensities: $\{H,M,L,HML,HM,HL,ML\}$.

System Throughput Results: We first present the effect of our mechanism on overall system/instruction throughput (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. 4 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 (average) performance improvement under congestion (e.g., load >0.7) is 27.6% (14.7%).

Fig. 4(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 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. 4(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



(a) 4x4 and 8x8 All Workloads. (b) Workload Category Breakdown. (c) CDF of Starvation Rates.

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

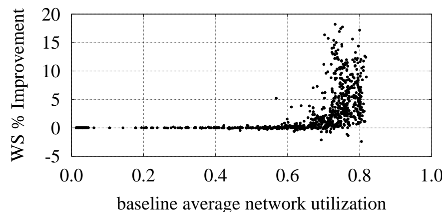


Figure 5: **Percentage improvements in weighted speedup.**

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* [20], 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* rates 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. Figure 5 shows weighted speedup improvements by up to 17.2% (18.2%) in the 4x4 and 8x8 workloads respectively.

Key Finding: *Our mechanism improves performance up to 27.6%, reduces starvation, and improves weighted speedup.*

7. 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., lack of congestion collapse). We develop an application-aware congestion control algorithm and show significant improvement in application-level system throughput on a wide variety of real workloads. 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.

8. ACKNOWLEDGMENTS

We acknowledge the support of Gigascale Systems Research Center, Intel, and CyLab. This research was partially supported by an NSF CAREER Award, CCF-0953246.

9. REFERENCES

[1] Appenzeller et al. Sizing router buffers. *SIGCOMM*, 2004.
[2] P. Baran. On distributed communications networks. *IEEE Trans. on Comm.*, 1964.
[3] L. Benini and G. D. Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35:70–78, Jan 2002.
[4] T. Bjerregaard et al. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 2006.
[5] S. Borkar. Thousand core chips: a technology perspective. *DAC-44*, 2007.
[6] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.
[7] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *DAC-38*, 2001.
[8] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. *MICRO-42*, 2009.
[9] P. Gratz, C. Kim, R. McDonald, and S. Keckler. Implementation and evaluation of on-chip network architectures. *ICCD*, 2006.
[10] B. Grot, J. Hestness, S. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. *HPCA-15*, 2009.
[11] M. Hayenga et al. Scarab: A single cycle adaptive routing and bufferless network. *MICRO-42*, 2009.
[12] Y. Hoskote et al. A 5-ghz mesh interconnect for a teraflops processor. *IEEE MICRO*, 2007.
[13] Intel Corporation. Single-chip cloud computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.
[14] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for future high bandwidth-delay product environments. *SIGCOMM*, 02.
[15] J. Kim, W. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ISCA-35*, 2008.
[16] J. Kim et al. Flattened butterfly topology for on-chip networks. *IEEE Computer Architecture Letters*, 2007.
[17] A. Mohammad et al. DCTCP: Efficient packet transport for the commoditized data center. *SIGCOMM*, 2010.
[18] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. *ISCA-36*, 2009.
[19] J. Owens et al. Research challenges for on-chip interconnection networks. *IEEE MICRO*, 2007.
[20] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-9*, 2000.
[21] Standard Performance Evaluation Corporation. SPEC CPU2006. `http://www.spec.org/cpu2006`.
[22] 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.
[23] 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`.
[24] D. Wentzlaff et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.