HAT: Heterogeneous Adaptive Throttling for On-Chip Networks

Kevin Kai-Wei Chang, Rachata Ausavarungnirun, Chris Fallin, Onur Mutlu
Carnegie Mellon University
{kevincha,rachata,cfallin,onur}@cmu.edu

***Abstract*—The network-on-chip (NoC) is a primary shared resource in a chip multiprocessor (CMP) system. As core counts continue to increase and applications become increasingly data-intensive, the network load will also increase, leading to more congestion in the network. This network congestion can degrade system performance if the network load is not appropriately controlled. Prior works have proposed *source-throttling* congestion control, which limits the rate at which new network traffic (packets) enters the NoC in order to reduce congestion and improve performance. These prior congestion control mechanisms have shortcomings that significantly limit their performance: either 1) they are not application-aware, but rather throttle all applications equally regardless of applications' sensitivity to latency, or 2) they are not network-load-aware, throttling according to application characteristics but sometimes under- or over-throttling the cores.**

In this work, we propose *Heterogeneous Adaptive Throttling*, or HAT, a new source-throttling congestion control mechanism based on two key principles: *application-aware throttling* and *network-load-aware throttling rate adjustment*. First, we observe that only network-bandwidth-intensive applications (those which use the network most heavily) should be throttled, allowing the other latency-sensitive applications to make faster progress without as much interference. Second, we observe that the throttling rate which yields the best performance varies between workloads; a single, static, throttling rate under-throttles some workloads while over-throttling others. Hence, the throttling mechanism should observe network load dynamically and adjust its throttling rate accordingly. While some past works have also used a closed-loop control approach, none have been application-aware. HAT is the first mechanism to combine application-awareness and network-load-aware throttling rate adjustment to address congestion in a NoC.

We evaluate HAT using a wide variety of multiprogrammed workloads on several NoC-based CMP systems with 16-, 64-, and 144-cores and compare its performance to two state-of-the-art congestion control mechanisms. Our evaluations show that HAT consistently provides higher system performance and fairness than prior congestion control mechanisms.

I. Introduction

In chip multiprocessors (CMP), the interconnect serves as the primary communication substrate for all cores, caches, and memory controllers. Since the interconnect implements cache coherence between fast L1 caches and lies on the datapath for all memory accesses, its latency and bandwidth are important. Hence, a high-performance interconnect is necessary to provide high system performance. At high core counts, simpler interconnects such as busses and crossbars no longer scale adequately. The most commonly proposed solution is a Network-on-Chip (NoC), which allows cores to communicate with a packet-switched substrate [6]. A two-dimensional mesh [5] is frequently implemented for large CMPs [18], [20], [39].

NoC designers also encounter on-chip hardware implementation constraints, such as total power (due to thermal limits) and chip area consumed by the network. Because these resources are often tight in modern CMPs, NoC designs with a large amount of buffers and wires are sometimes not practical. To maintain overall design efficiency, the system must be provisioned reasonably, and thus will sometimes experience *network congestion* when the network load (the fraction of occupied links or buffers) becomes too high. When this congestion occurs, packets contend for shared network resources frequently, and this can reduce overall system throughput. Congestion in on- and off-chip interconnects is a well-known problem that has been described in many prior works (e.g., [2], [9], [11], [15], [30], [31], [32], [38]). Congestion can significantly degrade system performance if it is not properly managed.

A common congestion-control approach is *source throttling* in which network nodes (sources) that are injecting significant amounts of traffic and interfering with the rest of the network are *throttled*, or temporarily prevented from injecting new traffic. A number of previous works have shown source throttling to be effective in interconnects [2], [15], [30], [31], [32], [38]. By reducing overall network load, source throttling reduces congestion and thus improves network performance.

Unfortunately, state-of-the-art source-throttling mechanisms have shortcomings that limit their performance, which we address in this work. First, many of the past works are not *application-aware* (e.g., [2], [32], [38]), i.e., do not take application characteristics into account when choosing which applications to throttle. As we show in this work, application-aware throttling decisions are essential to achieving good performance. Second, in the works that are application-aware (e.g., [30], [31]), the throttling rate itself (how aggressively the throttled nodes are prevented from injecting traffic) is adjusted in a manner that does not take into account the load or utilization of the network, which can lead to performance loss as it causes either over- or under-throttling of the sources. Adjusting the throttling rate by observing network load and using a closed-loop feedback mechanism that takes into account network load yields higher performance, as we show in this work.

To overcome these shortcomings, this paper proposes *Heterogeneous Adaptive Throttling (HAT)*, a new source-throttling mechanism. Unlike past works, HAT is the first to combine two key principles: *application-aware throttling* and *network-load-aware throttling rate adjustment*. HAT selectively throttles only some nodes (applications) in the network, chosen according to application characteristics, such that overall congestion is reduced without significant impact on any throttled node. This application-awareness allows applications that have little impact on network load to make fast forward progress. HAT then dynamically adjusts throttling rate of throttled nodes in a closed-loop fashion based on the total load (or utilization) of the network. Adapting the throttling rate to the load on the network minimizes the over- or under-throttling that occurs in a mechanism that does not adapt the throttling rate to network conditions. By combining the aforementioned two key principles together, HAT provides better performance than past source-throttling mechanisms [30], [31], [38], as we show in our quantitative evaluations.

In summary, our **contributions** are:

- We observe that two key principles are necessary to achieve high performance using source throttling in a congested on-chip network: application-aware throttling (choosing which applications to throttle based on application characteristics), and network-load-aware throttling rate adjustment (adjusting the aggressiveness of source throttling dynamically to avoid under- or over-throttling the applications).
- We introduce Heterogeneous Adaptive Throttling (HAT), the first source-throttling mechanism that uses these two key principles together to control congestion in a network. HAT 1) throttles bandwidth-intensive applications, thus allowing latency-sensitive applications to make fast progress, and 2) throttles these selected applications at a rate that is dynamically adjusted based on the network load, thus maximizing efficient utilization of the network.
- We qualitatively and quantitatively compare our new congestion control mechanism to two state-of-the-art source-throttling congestion-control mechanisms [30], [31], [38] on a variety of interconnect designs with different router designs and network sizes using a wide variety of workloads. Our results show that HAT provides the best system performance and fairness.

II. Background

A. Networks-on-Chip for Multi-Core Processors

In CMP systems (e.g., [18], [43]), NoCs (networks-on-chip) serve as communication fabrics to connect individual processing nodes. Each node has a core, a private cache, and a slice of the shared cache. One NoC router is placed at each node, and the NoC routers are interconnected by links. The routers convey packets between nodes. In the CMP system that we evaluate, NoCs primarily service private cache miss requests, carrying request and data packets between private caches, shared caches, and memory controllers. Therefore, NoCs are on the critical path of every cache miss, making NoC performance important for providing good CMP system performance. Many NoC topologies have been proposed; however, a 2D mesh is one of the most commonly used network topologies in commercial NoC-based manycore CMPs [43] and research prototypes [16], [18], [20], [37] because of the low layout complexity.

B. Network-on-Chip Operation and Router Design

NoCs carry packets between nodes (cores, caches, and memory controllers) in a CMP. Each packet consists of one or multiple *flits*, which are the unit of flow control. One flit is typically the unit of data conveyed by one network link in one cycle. When a router receives a packet, it decides which link to forward the packet to. When many packets are in the network, they can contend for shared resources, such as links. Conventional router designs often have *input buffers* at each router input [3], [5]. A packet that cannot obtain a link to travel to its next hop can wait in an input buffer. Alternatively, some other router designs [13], [28] eliminate these input buffers and instead *misroute* a contending packet to a different output link. The routing algorithm in such a *bufferless* NoC design ensures that each packet nevertheless eventually arrives at its destination. (We will evaluate our mechanisms on both such network designs.) In both input-buffered and bufferless NoC routers, performance degrades when contention for links or buffers occurs frequently. Source throttling reduces such congestion by limiting the rate at which traffic can enter the network.

III. Motivation

In this section, we first demonstrate that source-throttling can improve system performance. Then, we show that *application-aware* source-throttling is necessary when different applications share the same NoC. These two observations have motivated past work on source throttling [30], [31]. Finally, we show that the throttling rate which provides the best performance is not constant across workloads, motivating the need for network-load-aware throttling rate adjustment. This final observation leads to our *application-aware* and *network-load-aware* Heterogeneous Adaptive Throttling (HAT).

Source Throttling Congestion Control: Source throttling [2], [15], [30], [31], [32], [38] is a commonly used technique to address congestion in NoCs and general packet-switched networks. The idea is to reduce network load when the network is congested by throttling packets entering the network in order to enhance system performance. In a source-throttling system, sources (network nodes which inject packets) are notified when they are causing network congestion, either by other network nodes or by a central controller. Upon receiving such a notification, the source temporarily throttles (delays) its packets queued for network injection. By reducing the injection rate (rate of packets entering the network), throttling reduces network load and hence reduces congestion.

To show the benefits of source throttling, we evaluate the system performance of three heterogeneous application workloads on a NoC-based CMP system (details in §VI) when different static throttling rates are applied to all applications. As we will describe in detail later, a throttling rate is simply the probability that any given injection of a new packet into the network is blocked (e.g., 80% throttling allows new packets to be injected only 20% of the time). Figure 1 shows that as the network is throttled (increasing throttling rate), the system performance also increases to a point, because the reduced rate of injecting new packets reduces contention for the shared resources in the network, hence reducing congestion. However, past a point, the source-throttling is too aggressive: it not only reduces congestion but also limits forward progress unnecessarily, degrading performance. We conclude from this experiment that throttling can improve system performance when applied appropriately.

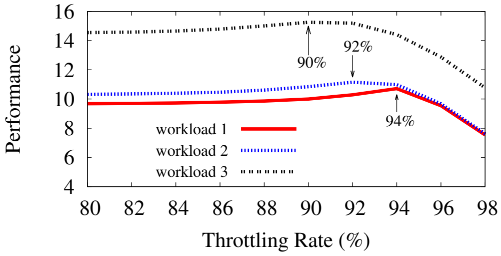

Figure 1. Performance of randomly mixed workloads with varied homogeneous throttling rate on 16-node (4x4) CMP systems.

Application-Aware Throttling: We motivate the importance of differentiating applications based on their intensity by showing that the overall and per-application performance can be significantly impacted when different applications are throttled. To illustrate this point, we reproduce an experiment shown in Nychis et al. [30], [31] to motivate their application-aware technique. Source throttling is applied in two different ways to a 16-application workload on a 4x4-mesh NoC-based CMP. This workload consists of eight instances of *mcf*, and eight of *gromacs*, which are network-intensive (use significant bandwidth) and network-non-intensive (use little bandwidth), respectively. Note that although we are showing one particular workload here, its behavior is representative of workloads with mixed application intensities in general. Table I shows the performance of this workload when (i) *only mcf* is throttled, and (ii) *only gromacs is throttled* (in both cases, by a fixed throttling rate of 95%), relative to the baseline where no applications are throttled.

Table I shows three application (instruction) throughput measurements for each throttling case normalized to no throttling: overall system throughput, average across mcf instances, and average across gromacs instances. We reiterate the following key observations demonstrated in prior work [30], [31]. Throttling different applications affects overall system performance differently. Throttling mcf (second column) increases system performance by 9%, whereas throttling gromacs (third column) decreases system performance by 2%. The main reason is that applications respond differently to network latency variations. Throttling mcf significantly reduces network interference and this allows gromacs to make faster progress (by 14%) with less network latency. Also, mcf's performance increases due to reduced network congestion and because it is not sensitive to increased network latency. In contrast, throttling gromacs negatively impacts gromacs by 4% without any benefit to mcf. The reason for this behavior is that gromacs injects fewer network packets than mcf, thus each packet represents a greater fraction of forward progress for gromacs. As a result, network-non-intensive applications (e.g., gromacs) are more sensitive to network latency and network-intensive applications (e.g., mcf) are more tolerant to network latency. Based on these observations, we conclude that throttling network-intensive applications is more beneficial to system performance than throttling network-non-intensive applications. We thus use *application-aware* throttling in HAT based on network intensity.

Metric	No Throttle	Throttle mcf	Throttle gromacs
System Throughput	1.0	1.09	0.98
mcf Throughput	1.0	1.05	1.0
gromacs Throughput	1.0	1.14	0.96

Table I. Normalized instruction throughput of selectively throttling in an application-aware manner.

Network-Load-Aware Throttling Rate Adjustment: To understand the benefits of adjusting throttling rate dynamically, let us re-examine Figure 1. As we have already argued, throttling can be applied to all the workloads evaluated in this figure to improve system performance. However, each workload requires a different throttling rate to achieve peak performance (in the example given, 94% for workload 1, 92% for workload 2, and 90% for workload 3). Prior source-throttling techniques use either a static throttling rate [38] or an open-loop rate adjustment based on measured application characteristics [30], [31]. As we will demonstrate in our results (§VII), adjusting the throttling rate dynamically with a *network-load-aware feedback mechanism* can tune throttling rate more precisely for higher performance. Thus, the second key principle of Heterogeneous Adaptive Throttling (HAT) is *network-load-aware throttling rate adjustment.*

IV. HAT: Heterogeneous Adaptive Throttling

A. Overview

Our proposed congestion-control mechanism, HAT, is designed to improve system performance through source throttling. HAT applies throttling to reduce network load when the network is congested, reducing interference and thus improving performance. In order to be most effective at achieving high system throughput, HAT applies two *key ideas*, as motivated above: *application-aware throttling* and *network-load-aware throttling rate adjustment.*

The first component is *application-aware throttling*. HAT throttles network-intensive applications that severely interfere with network-non-intensive applications, and does not throttle the network-non-intensive applications since they do not significantly impact network load. As a result, the unthrottled network-non-intensive applications can make faster progress with less interference. To determine which applications to throttle, HAT first measures applications' network intensity. Then HAT divides applications into two groups based on their intensities and performs throttling on the network-intensive group. Since it applies throttling to some applications, HAT is *heterogeneous*.

The second component, *network-load-aware throttling rate adjustment*, aims to find a throttling rate that reduces congestion sufficiently (i.e., does not throttle too lightly) while not severely degrading performance (by throttling too heavily). In order to dynamically adapt to different workloads, HAT measures *network load* (in a simple way that we define below), and controls throttling rate in a closed-loop manner to keep network load near an empirically-determined optimal value (which is constant for a given network topology and size). This ensures that the available network bandwidth is efficiently used without undue congestion.

B. Source Throttling

We define *throttling* as the blocking of new traffic injection into the network from a particular application in order to reduce load. We use a throttling technique that is *soft*: throttled nodes can still inject with some non-zero probability, explained below. We also assume that each node has separate injection queues for request packets and data response packets to other nodes, and that only a node's own request packets are throttled. For a throttling rate r, every time a node attempts to inject a request packet, it chooses to block the injection with probability r. Throttling rates range from 0% (unthrottled) to 100% (fully blocked). In this work, the maximum throttling rate is set to 95% (mostly blocked), explained further in §IV-D. When a packet is blocked from injection, it will retry its injection in subsequent cycles until it is allowed to inject.

C. Application-Aware Throttling

To perform application-aware throttling, HAT first observes each application's network intensity, classifies applications as network-intensive or network non-intensive based on their intensities, and then applies throttling only to network-intensive applications.

Measuring Application Intensity: HAT must measure applications' network intensity in order to classify them. Because the network in our baseline CMP design primarily serves L1 cache misses (due to the cache hierarchy design which places the NoC between private L1 caches and a shared, globally distributed L2 cache), HAT uses L1 MPKI (misses per thousand instructions) as a metric that closely correlates to network intensity (note that this measurement also correlates with network injection rate). L1 MPKI is easy to measure at each core with two counters (completed instruction count and cache miss count).[1] Other works [7], [8], [24], [29] also use L1 MPKI to measure application intensity.

Application Classification: Once application intensity is measured, HAT must split applications into intensive and non-intensive categories. Some past application-aware source throttling techniques have used a *threshold* to split applications: all applications with a network intensity above a certain threshold are considered network-intensive. However, we observe that the NoC fundamentally has some *total network capacity*, and that throttling an individual application that is above some threshold may over- or under-throttle it depending on the application's network intensity. If all applications have intensity just below the threshold, none would be throttled. In contrast, if total network load is low with only one application's intensity just over the threshold, that application could be unnecessarily throttled. Thus, HAT instead sets a threshold on the *total network intensity of all unthrottled applications*. This ensures that unthrottled applications do not provide sufficient load to cause network congestion. All other applications are then throttled to limit their impact on total network load while still allowing them to make some forward progress.

Algorithm 1 shows HAT's per-epoch application classification procedure which implements this key idea. Every epoch, each application's network intensity (measured as L1 MPKI) is collected. At the end of the epoch, applications are sorted by measured network intensity. Starting with the least network-intensive application, the applications are placed into the network-non-intensive unthrottled application category until the total network intensity (sum of L1 MPKIs) for this category exceeds a pre-set threshold, which we call $NonIntensiveCap$ (we will examine the impact of this parameter in §VII-G; in brief, it should be scaled with network capacity and tuned for the particular network design). All remaining applications (which are more network-intensive than these unthrottled applications) are then considered network-intensive and are throttled with the same global throttling rate for the following epoch. Note that these throttled applications can still make some forward progress, since throttling does not prevent all packet injections, but only delays some of them. However, the network-intensive applications' impact on network congestion is reduced significantly, and the network non-intensive applications' performance is therefore likely to increase.

Algorithm 1 HAT: Application Classification Algorithm

```
at the beginning of each epoch:
empty the groups
sort $N$ applications by MPKI measurements $MPKI_i$
for sorted application $i$ in $N$ do
    if total MPKI of network-non-intensive group $+MPKI_i \leq NonIntensiveCap$ then
        Place application $i$ into the network-non-intensive group
    else
        Place application $i$ into the network-intensive group
    end if
end for
```

D. Network-Load-Aware Throttling Rate Adjustment

Recall that we observed in §III that workloads achieve their peak performance at different global throttling rates. We have observed that despite the differences in throttling rate, the *network utilization*, or measure of the fraction of links and buffers in the network occupied on average, is a good indicator of where peak performance will occur. HAT thus adjusts the global throttling rate that is applied to every classified *network-intensive* application within a workload dynamically using a feedback-based mechanism that targets a *network utilization target*. The network utilization target is fixed for a given network design (a sensitivity study on its effect is presented §VII-G). This minimizes over-throttling (which could hinder forward progress unnecessarily) or under-throttling (which could fail to alleviate congestion) of the workload. **Controlling Throttling Rate:** At each epoch, HAT compares actual network utilization with a target uti-

[1] Note that we do not use L1 MPKC (misses per thousand *cycles*) because it is dependent on the throttling rate, hence dynamic throttling changes could alter application classifications and induce control instability.

lization set statically for the network design. HAT adjusts the global throttling rate upward when the network utilization is higher than the target and downward when the network utilization is lower than the target by a throttling rate step. Per-epoch rate steps are given in Table II: the rate moves with larger steps when it is lower, and smaller steps when it is higher, because most applications are more sensitive to throttling at higher throttling rates.[2] The throttling rate has a maximum value at 95%, based on empirical sensitivity sweeps that show a good performance improvement at that point without unduly impacting the throttled nodes. Note that when the baseline network utilization is below the target (indicating a lightly-used network that will likely have very little congestion), HAT will converge to a rate of 0%, effectively deactivating the throttling mechanism.

Current Throttling Rate	Throttling Rate Step
0% – 70%	10%
70% – 90%	2%
90% – 94%	1%

Table II. Throttling rate adjustment used in each epoch.

E. Implementation Cost

Our mechanism consists of two major parts in each epoch: (i) measurement of L1 MPKI and network utilization, and (ii) application classification and throttling rate computation.

First, L1 MPKI and network utilization are measured in hardware with integer counter structures resetting every epoch. Each core requires two hardware counters to measure L1 MPKI: one L1 miss counter and one instruction counter. Measuring network utilization requires one hardware counter in each router to monitor the number of flits that are routed to direct neighboring nodes. Second, throttling rates must be enforced at each node. Throttling can be implemented as a fixed duty cycle in which injection is allowed. Finally, the application classification computation must be done in some central node. The algorithm consists of (i) a sort by MPKI and (ii) one pass over the application list. Although a detailed model is outside the scope of this work, such a computation requires at most several thousand cycles out of a 100K-cycle epoch, running on one central CPU node. This computation and communication overhead is small given a 100K-cycle epoch. Assuming the computation takes 1000 cycles, then the overhead is only 1% on a single core. Similar coordination mechanisms have been proposed in previous work (e.g., [7], [8], [30], [31]) and are shown to have low overhead.

V. Related Work

To our knowledge, HAT is the first work to combine application-aware source-throttling and dynamic network load-aware throttling adjustment to reduce network congestion. In this section, we discuss prior work that addresses network congestion in other ways.

Congestion Control in NoCs: The congestion problem in NoCs has been widely explored in many previous works. Various approaches use available buffers [22], available virtual channels [4], output queue size [33] or a combination of these [17] to detect congestion, and then perform adaptive routing to avoid congested regions. However, these techniques do not actually reduce high network load because they do not throttle sources injecting intensive traffic into the network. As congestion increases with higher network load, adaptive routing can no longer find productive paths to route around congested regions, leading to system performance degradation. Furthermore, adaptive routing is less effective in bufferless networks because deflection routing provides adaptive routing anyway when network congestion is high. In contrast to adaptive-routing approaches, our source-throttling proposal (i) reduces network congestion by source throttling, effectively improving performance when network load is too high, (ii) uses application-aware throttling rather than network-level measurement to provide application-level performance gains, and (iii) works on both buffered and bufferless networks.

A few prior works use throttling to manage congestion in NoCs. In particular, Nychis et al. [30], [31] propose an application-aware throttling mechanism to mitigate the effect of congestion in *bufferless* NoCs. The proposed technique detects congestion based on each application's progress and triggers throttling when any application's starvation rate (fraction of cycles a node attempts to inject a flit but cannot) is above a threshold. Then it throttles applications that have network intensity above average with a fixed throttling rate proportional to their intensities. However, their mechanism is not adaptive to varying network conditions, hence can under- or over-throttle the applications (which reduces performance) as we will show in §VII.

Congestion Control in Off-chip Networks: Congestion control in off-chip interconnects (e.g., between compute nodes in a supercomputer) has been studied extensively. While these networks resemble NoCs in some regards, and use some of the same design techniques, many of the congestion control mechanisms for off-chip networks are not directly applicable to NoCs. Off-chip networks differ from on-chip networks in terms of both architecture and scale. For example, the popular Infiniband [15], [19] interconnect which is used in many large-scale supercomputers uses a multistage switch topology and can scale to thousands of nodes. In this topology, switch nodes are separate from compute nodes, and switches are large devices with many input and output ports. Thus, there are relatively fewer switches (since each switch serves more compute nodes). Latencies are also typically orders of magnitude longer because link distances are greater (meters rather than millimeters). For both these reasons, each switch in an off-chip interconnect can employ more complex routing and congestion control algorithms than what is possible in on-chip networks, where each router must be relatively small and very fast. Furthermore, central coordination (as we propose) is much more difficult in such an environment. More complex distributed schemes are typically used instead.

Despite these differences, source throttling is a common technique used by many works [2], [15], [32], [38] to remove

[2] We observe that non-linear steps help reduce the settling time of the throttling rate. In all of our experiments, we choose an epoch length that allows the throttling rate to settle before the end of an epoch.

Parameter	**Setting**
System topology	8x8 or 12x12 2D-mesh; core and shared cache slice at every node
Core model	Out-of-order, 128-entry instruction window, 16 miss buffers (i.e., MSHRs), stall when buffers are full
Private L1 cache	64 KB, 4-way associative, 32-byte block size
Shared L2 cache	perfect (always hits), cache-block-interleaved mapping
Cache coherence	directory-based with cache-to-cache transfers, perfect striped directory
Interconnect Links	1-cycle latency, 32-bit width (1-flit request packets, 8-flit data packets)
Bufferless router	2-cycle latency, FLIT-BLESS [28] or CHIPPER [13]
Buffered router	2-cycle latency, 4 VCs [3], 4 flits/VC, buffer bypassing [42]

Table III. Simulated system parameters.

congestion in an off-chip network. Thottethodi et al. [38] propose a mechanism to completely block the injection of every source when the globally measured virtual-channel buffer occupancy in a buffered interconnect exceeds a threshold. This threshold is dynamically tuned to maximize system throughput. This mechanism thus is triggered by *network-level* conditions rather than *application-level* performance measurements, and treats all applications the same (hence reducing fairness when applications have differing network requirements). In addition, to determine when to throttle and to adjust its threshold, this throttling mechanism measures and exchanges congestion and throughput information globally at fine-grained intervals (every 32 and 96 cycles, respectively), which requires a dedicated side-band network to avoid making congestion worse. Despite its high hardware overhead, we compare, in §VII, HAT to a variant of this mechanism adapted to the on-chip network design. We assume that global communication incurs no overhead for [38], which benefits the performance of this mechanism.

Another common off-chip congestion control approach is based on avoiding head-of-line (HoL) blocking, in which packets at the head of a queue cannot allocate downstream buffers and make forward progress. These blocked packets in turn block other packets further upstream. Many works [1], [9], [11], [36] have proposed allocating separate queues for different packet flows within a switch to avoid HoL blocking. For example, separate queues could be allocated by output port [1] or dynamically for congested packet flows [9]. However, such techniques rely on large hardware buffers, and require more complex packet processing, both of which are poorly adapted to on-chip routers that have limited power and on-die area, and only a short time to process each packet.

VI. Experimental Methodology

Simulator Model: We use an in-house cycle-level CMP simulator that consumes instruction traces of x86 applications for our evaluation of application-level performance. We model 64- and 144-node CMP systems with 8x8 and 12x12 2D-mesh cycle-level networks, respectively. Each node stalls when its request buffers, instruction window, etc are full. We faithfully model private L1 data caches and a shared L2 cache with a directory-based cache coherence protocol [26]. All private L1 caches use Miss Status Holding Registers (MSHRs) [25] to track requests to the shared L2 cache until they are serviced. Unless stated otherwise, our detailed system configuration is as shown in Table III.

Note that we model a *perfect shared L2 cache* to stress the network, as also evaluated for CHIPPER [13], BLESS [28] and previous work on throttling [30], [31]. In this model, all L1 misses hit in their destination L2 cache slice. This setup potentially increases the network load compared to a system with realistic off-chip memory that often becomes the system bottleneck. Note that we still use realistic configurations (e.g., access latency and MSHRs) for the cache hierarchy. As a result, using a perfect shared L2 cache allows us to conduct studies on the fundamental congestion problem due to capacity of the evaluated networks.

Because only the L1 caches require warming with a perfect shared cache, 5M cycles of warmup is found to be sufficient. We find that 25M cycles of execution gives stable results for the applications simulated.

Evaluated Router Designs: We evaluate HAT and prior throttling mechanisms on NoCs with the following router designs: i) BLESS [28], ii) CHIPPER [13], and iii) VC-buffered router with buffer bypassing [42].

Workloads: We focus on network-intensive workloads for our main evaluations since the problem we are solving occurs only when the network is under high load. In a sensitivity study in §VII-G, we will separately show behavior in network non-intensive workloads. To form suitable workloads for our main evaluations, we split the SPEC CPU2006 [35] benchmarks based on L1 MPKI into three categories: High, Medium and Low. In particular, High-intensity benchmarks have MPKI greater than 50, Medium-intensity benchmarks fall between 5 and 50 MPKI, and Low-intensity benchmarks form the remainder. Based on these three categories, we randomly pick a number of applications from each category and form seven intensity mixes with each containing 15 workloads: L (All Low), ML (Medium/Low), M (All Medium), HL (High/Low), HML (High/Medium/Low), HM (High/Medium), and H (All High).

We also evaluate multithreaded workloads in §VII-F using the SPLASH-2 [44] applications. For these workloads, an 8x8 network runs 64 threads of a single application. Simulations are run for a fixed number of barriers (i.e., main loop iterations) and total execution times are compared.

Performance Metrics: We measure *application-level system performance* with the commonly-used *Weighted Speedup* metric: $WS = \sum_{i=1}^{N} \frac{IPC_i^{shared}}{IPC_i^{alone}}$ [12], [34]. All IPC_i^{alone} values are measured on the baseline network without throttling. We also briefly report our results in *System IPC* [12] and *Harmonic Speedup* [12], [27], but we do not present full results for brevity.

Additionally, to show that throttling does not unfairly penalize any single application to benefit the others, we report *unfairness* using maximum application slowdown:

$Unfairness = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}$ [7], [23], [24], [29], [40]. For this metric, lower is better (more fair). Because slowdown is the inverse metric of speedup, we report the harmonic mean rather than arithmetic mean of unfairness.

HAT Parameters: Unless stated otherwise, we set the the *epoch length* to 100K cycles. For the *NonIntensiveCap* (*network utilization target*), we use 150 (60%), 150 (55%), and 350 (5%) for BLESS, CHIPPER, and VC-buffered, respectively (all determined empirically).

Comparisons to Other Mechanisms: We compare our mechanism to two state-of-the-art throttling techniques in our evaluation. The first comparison is to an implementation of prior work on throttling in bufferless networks proposed by Nychis et al. [30], [31], which we discussed in §V (called *Heterogeneous Throttling* in §VII). We adapt this work to buffered NoCs for comparisons as well. Second, we compare to the mechanism proposed by Thottethodi et al. [38] that is also described in §V (called *Self-Tuned* in §VII). Note that *Self-Tuned* was proposed explicitly for buffered networks, and its adaptation to bufferless networks is non-trivial because the algorithm measures buffer utilization; hence we compare against *Self-Tuned* only in VC-buffered networks.

VII. Experimental Evaluation

In this section, we evaluate the performance of HAT on NoCs with different router designs: BLESS [28], CHIPPER [13], and VC-buffered [3]. We demonstrate that HAT provides better system performance than prior throttling mechanisms on all designs. In addition, we report results on fairness, and show that HAT does not impair fairness when throttling to improve overall performance.

A. Results on Buffered NoCs

Figure 2 shows the application-level system performance (left panel) and fairness (right panel) of the VC-buffered network in a 64-node (8x8) CMP system. In fairness plots, a higher number indicates higher maximum slowdown and thus worse fairness. For each bar group, results are presented in the order of baseline, *Heterogeneous Throttling* [30], [31], *Self-Tuned* [38], and HAT. Results are split by workload categories as defined in §VI, with network intensities increasing from left to right. Based on the results, we find that HAT consistently provides better system performance and fairness than the two previously proposed mechanisms. We make the following observations:

1. HAT improves system performance by 3.9% and fairness by 4.9% on average across all workloads on the VC-buffered network. The main reason for this performance gain over the baseline is that HAT performs throttling on applications according to their network intensities. By throttling network-intensive applications that contribute the majority of congestion at all times, HAT reduces network load and allows better network access for the other applications.

2. Compared to *Heterogeneous Throttling*, which is the best previous throttling mechanism that we evaluate, HAT attains 3.5% (4.4%) better system performance (fairness) on average across all workloads. This is because *Heterogeneous Throttling* does not adjust its throttling rate according to network load when it throttles network-intensive applications. Instead, it triggers throttling when at least one node is starved and uses a fixed throttling rate that is independent of actual network load. This can result in network under- or over-utilization.

3. As the workload's network intensity increases, HAT's performance improvement becomes more significant because congestion increases. This trend changes in the highest-load category ("H") because these workloads consist only of network-intensive applications, which are nearly always network-bound, frequently stalling on outstanding requests. In such a uniformly high-network-intensity workload, applications are not sensitive to network latency, thus throttling cannot help as much. Throttling is most effective when some applications in the "L" and "M" categories are in the network, because these applications are more sensitive to network latency and can benefit from reduced interference. To confirm our qualitative explanation, we observe that HAT reduces packet latency by 28.3% and 18.2% on average across applications in the "L" and "M" categories, respectively. On the other hand, HAT increases packet latency by 3.0% on average across applications in the "H" category with little system performance degradation. This is because these applications have better network latency tolerance compared to latency-sensitive applications. As a result, we conclude that HAT provides the highest improvement when the workload mix is heterogeneous.

4. In our evaluations, *Self-Tuned* provides little performance gain over the baseline (and HAT attains 5.0% better system performance and 5.3% better fairness than *Self-Tuned*). This is because the throttling algorithm in *Self-Tuned* performs *hard throttling*: when the network is throttled, no new packet injections are allowed until the congestion (measured as the fraction of full packet buffers) reduces below a threshold. Thus, *Self-Tuned* will temporarily reduce congestion by halting all network injections, but the congestion will return as soon as injections are allowed again. *Self-Tuned* [38] was previously evaluated only with synthetic network traffic, rather than with a realistic, closed-loop CMP model as we do. This previous evaluation also found that network throughput improvements only occur near network saturation. In contrast to *Self-Tuned*, HAT is able to obtain performance and fairness improvements over a wide range of network load by (i) adaptively adjusting its throttling rate according to network load and (ii) selectively throttling certain applications based to their network intensity.

We conclude that, by using application-aware throttling with adaptive rate adjustment, HAT reduces inter-application interference to achieve better system performance and fairness than previously proposed mechanisms.

B. Results on Bufferless NoCs

In this section, we present results on 64-node (8x8) systems with two different *bufferless* network designs. Figure 3 shows the application-level system performance and fairness for CHIPPER and BLESS with the same set of workloads shown in the previous section. For each network's bar group,

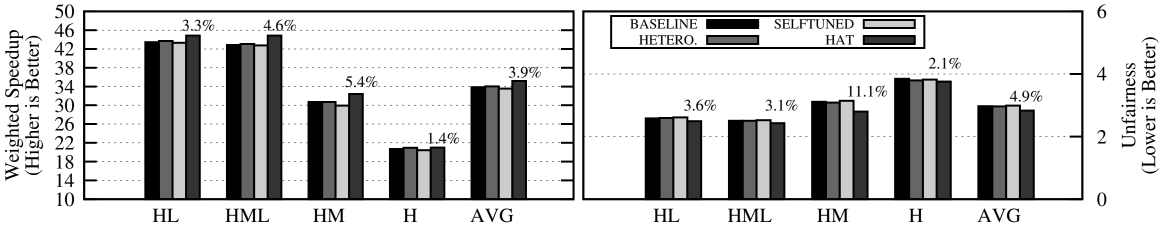

Figure 2. System performance and fairness of 64-node (8x8) CMP systems with VC-buffered NoCs. % values are gains of HAT over VC-buffered.

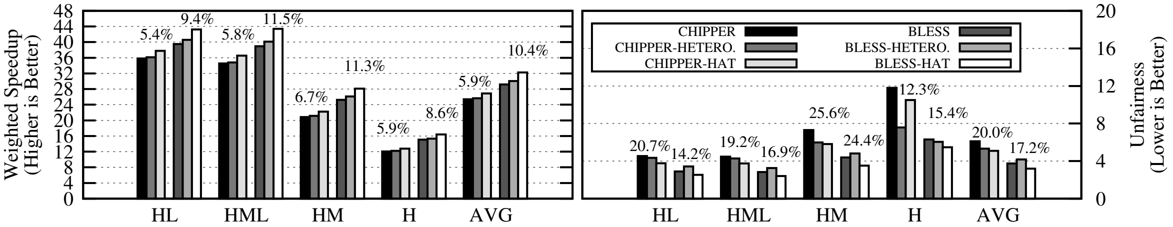

Figure 3. System performance and fairness of 64-node CMP systems with CHIPPER and BLESS. % values are gains of HAT over CHIPPER or BLESS.

we show results in the order of baseline, *Heterogeneous Throttling* [30], [31], and HAT. *Self-Tuned* [38] is not evaluated because the mechanism detects congestion based on the fraction of full VC-buffers, which are not available in these bufferless router designs. Several observations are made based on the results:

1. Similar to our results on VC-buffered networks presented above, HAT provides the best performance and fairness among all evaluated mechanisms. Compared to *Heterogeneous Throttling* (the best previous mechanism) in a CHIPPER (BLESS) network, HAT improves system performance by 5.1% (7.4%) and fairness by 4.5% (23.7%) on average. Relative to a baseline network with no congestion control, HAT improves system performance by 5.9% (CHIPPER) and 10.4% (BLESS) on average across all workloads. Furthermore, HAT attains 20.0% (CHIPPER) and 17.2% (BLESS) better fairness than the baseline network. The reason for the performance improvement is that in a CHIPPER (BLESS) network, HAT reduces average packet latency by 22.4% (37.4%) and 9.1% (18.6%) for latency-sensitive applications in the “L” and “M” categories, respectively. Although the average packet latency increases by 14.5% (16.1%) for “H” applications, HAT still improves both system performance and fairness since these applications are tolerant to network latency. Note that congestion is more severe in bufferless networks than in VC-buffered networks because the lack of buffers reduces total network throughput. Because of this increased congestion, HAT attains larger improvements over baseline in both performance and fairness in these bufferless networks than in VC-buffered networks.

2. In the “H” workload category that consists of only high-intensity applications, in the CHIPPER network, *Heterogeneous Throttling* provides better fairness than HAT (contrary to the trend in other workload categories). This behavior is due to the way *Heterogeneous Throttling* picks nodes to throttle: the throttling mechanism happens to favor the single most-slowed down application. This decision causes improvement in the maximum-slowdown definition of unfairness that we use. However, if we examine Harmonic Speedup (another measure of fairness [12], [27]), HAT actually provides 4.3% higher fairness than *Heterogeneous Throttling* for this workload category.

We conclude that HAT is effective for a variety of NoC designs, and in fact it provides higher performance gains when the network capacity is limited, as is the case in bufferless deflection networks.

C. *Scalability with Network Size*

Figure 4 presents system performance (left panel) and fairness (right panel) of HAT on VC-buffered, CHIPPER, and BLESS NoCs with the network size varied from 4x4 (16 nodes) to 12x12 (144 nodes). The percentage on top of each bar indicates the performance improvement of HAT over the baseline network. HAT always improves system performance and fairness over the baselines. The best performance gain is observed on 8x8 networks because we reasonably optimized HAT's parameters for these network configurations. For other networks (4x4 and 12x12), we did not optimize the parameters. Even with non-optimized parameters, HAT still improves both system performance and fairness. We expect that further performance gain can be achieved when parameters are tuned for these other network sizes.

D. *Sensitivity to Performance Metrics*

To ensure no bias in selecting metrics to report our system performance, we briefly present our results in system IPC and harmonic speedup (described in §VI). HAT improves average system IPC by 4.8%/6.1%/11.0% and average harmonic speedup by 2.2%/7.1%/10.5% over the best previous mechanism on VC-buffered/CHIPPER/BLESS 8x8 NoCs. We conclude that HAT provides consistent performance improvement over all evaluated mechanisms.

E. *Energy Efficiency*

In addition to system performance, we report energy efficiency as performance-per-Watt, which is weighted speedup divided by average network power. We use a power model developed by Fallin et al. [14], which is based on ORION 2.0 [41], to evaluate static and dynamic power

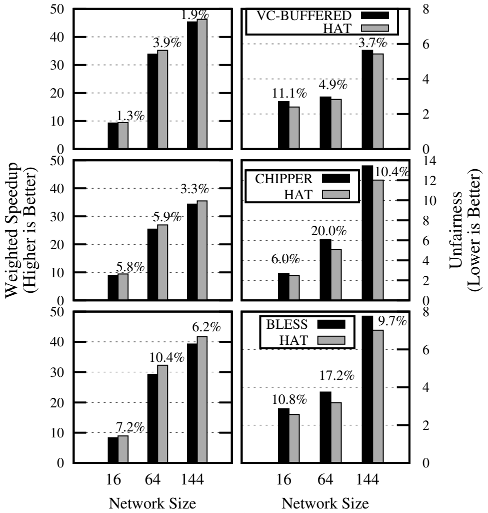

Figure 4. System performance and fairness of CMP systems with varied network sizes and VC-buffered, CHIPPER, and BLESS routers.

consumed by the on-chip network. Table IV shows the energy efficiency improvement of HAT compared to different networks without source throttling. HAT improves energy efficiency by up to 14.7% because it allows packets to traverse through the network more quickly with less congestion, thus reducing dynamic energy consumption. HAT shows the greatest improvement on CHIPPER because dynamic energy dominates the total energy due to higher deflection rate in CHIPPER.

Router design	VC-Buffered	BLESS	CHIPPER
Δ Perf/Watt	5.0%	8.5%	14.7%

Table IV. Energy efficiency improvement of HAT.

F. Effect on Multithreaded and Low-Intensity Workloads

Multithreaded Workloads: Table V shows execution time reduction of HAT relative to baselines without throttling on 64-node systems with different network designs running four 64-threaded workloads from SPLASH-2 [44]. Although HAT is not explicitly designed for multithreaded workloads, it can still provide benefit by reducing network contention. HAT improves performance of `lun`, which has roughly 20 MPKI, by 7.5%/4.2%/1.0% on VC-buffered/BLESS/CHIPPER NoCs. This is because HAT reduces the average network latency by 15.2% (VC-buffered), 9.0% (BLESS), and 12.8% (CHIPPER). On the other hand, *Heterogeneous Throttling* improves performance of `lun` by 1.6%/0.8%/2.0% on VC-buffered/BLESS/CHIPPER NoCs. The reason that *Heterogeneous Throttling* shows less performance gain than HAT is that starvation rate, the metric monitored by *Heterogeneous Throttling* to make the throttling decision, remains low through the execution.

In addition, neither HAT or *Heterogeneous Throttling* shows performance impact on `fft`, `luc`, and `cholesky` because these workloads have very low network intensity with 4.3 MPKI, 2.0 MPKI, and 4.9 MPKI, respectively. We do not present results on these applications for *Heterogeneous Throttling* because they show similar trend (no performance improvement) as HAT. New throttling mechanisms that selectively throttle non-bottleneck or non-limiter threads (e.g., as determined in [21], [10]), can provide better performance in multithreaded workloads compared to HAT, and the development of such mechanisms is a promising avenue of future work.

Benchmark	fft	luc	lun	cholesky
VC-Buffered	0.1%	0.0%	7.5%	-0.1%
BLESS	0.1%	0.0%	4.2%	0.0%
CHIPPER	0.1%	-0.1%	1.0%	-0.1%

Table V. Execution time reduction of HAT on multithreaded workloads.

Low-Intensity Workloads: We observe that HAT does not impact system performance of low-intensity workloads (i.e., L, ML, and M mixes). Since these workloads have minimal contention, there is no need for throttling. All techniques that we evaluate perform essentially the same.

G. Sensitivity Analysis

For brevity, we only present sensitivity results for 4x4 BLESS network and we do not present extensive results for some studies.

Capacity of Network-Non-Intensive Applications: The total network intensity of applications which are not throttled ($NonIntensiveCap$) determines what portion of the network-non-intensive nodes are unthrottled. It is important that this parameter is set appropriately for a given network design: if it is too large, then the network will be overloaded because too many network-intensive applications will be allowed unthrottled access to the network. On the other hand, if it is too small, then too few applications will be allowed unthrottled network access, and some network non-intensive applications may be throttled despite their low impact on network load.

Table VI shows the performance of HAT compared to the baseline with no throttling as $NonIntensiveCap$ varies. When every node is throttled (NonIntensiveCap= 0), HAT provides the best fairness because the most slowed down application can make faster progress with reduced congestion. As $NonIntensiveCap$ increases, system performance increases since the low-intensity applications are given more opportunities to freely inject, but fairness slightly degrades because some nodes are throttled more heavily to reduce the network load. When the parameter is increased past a certain point, HAT will be unable to bring the network load down to the target even if the throttling rate is near 100% because too few nodes are throttled. Such a situation results in unfair treatment of the few applications that are throttled and small performance gains overall because network load is lowered only slightly.

	0	50	100	150	200	250
Δ WS	8.5%	10.1%	10.6%	10.8%	10.3%	9.6%
Δ Unfairness	-11%	-9.5%	-7.1%	-5.0%	-2.5%	-2.1%

Table VI. Sensitivity of HAT improvements to $NonIntensiveCap$.

Epoch Length: We use an epoch length of 100K cycles which shows a good trade-off between responsiveness and overhead. The performance delta is less than 1% if we vary the epoch length from 20K to 1M cycles. A shorter epoch captures more fine-grained changes in application behavior, but with higher overhead. A longer epoch amortizes HAT's computation over more time, but does not promptly capture fine-grained changes in applications' behavior.

Target Network Utilization: This parameter controls the throttling aggressiveness of HAT and thus its effectiveness. We find that weighted speedup peaks at between 50% and 65% network utilization with less than 1% variation at points within this range. In addition, we observe that fairness is best in this range. Beyond 65%, performance drops significantly (by as much as 10%) because throttling is not aggressive enough, and the network remains congested.

VIII. Conclusion

In this paper, we present Heterogeneous Adaptive Throttling (HAT), a new application-aware throttling mechanism that reduces congestion to improve performance in NoC-based multi-core systems. HAT achieves this improvement by using two key principles. First, to improve system performance, HAT observes applications' network intensity and selectively throttles network-intensive applications, allowing latency-sensitive applications to make fast progress with reduced network congestion. Second, to minimize the over- and under-throttling of applications, which limits system performance, HAT observes the network load and dynamically adjusts the throttling rate in a closed-loop fashion. To our knowledge, HAT is the first source-throttling mechanism that combines application aware throttling and network-load-aware throttling rate adjustment. We show that HAT outperforms two different state-of-the-art congestion-control mechanisms, providing the best system performance and fairness. We conclude that HAT is an effective high-performance congestion-control substrate for network-on-chip-based multi-core systems.

Acknowledgments

We thank the anonymous reviewers for their feedback. We gratefully acknowledge members of the SAFARI group at CMU for feedback. Kevin Chang is partially supported by Prabhu and Poonam Goel Fellowship. Rachata Ausavarungnirun is partially supported by Royal Thai Government Scholarship. Chris Fallin is partially supported by an NSF Graduate Research Fellowship. We acknowledge the generous support of AMD, Intel, Oracle, and Samsung. This research was partially supported by an NSF CAREER Award CCF-0953246.

References

[1] T. E. Anderson et al. High-speed switch scheduling for local-area networks. 1993.

[2] E. Baydal et al. A congestion control mechanism for wormhole networks. *PDP-9*, 2001.

[3] W. Dally. Virtual-channel flow control. *IEEE TPDS*, 1992.

[4] W. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE TPDS*, 1993.

[5] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[6] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *DAC-38*, 2001.

[7] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. *MICRO-42*, 2009.

[8] R. Das et al. Aérgia: exploiting packet latency slack in on-chip networks. *ISCA-37*, 2010.

[9] J. Duato et al. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. *HPCA-11*, 2005.

[10] E. Ebrahimi et al. Parallel application memory scheduling. In *MICRO-44*, 2011.

[11] Escudero-Sahuquillo et al. FBICM: efficient congestion management for high-performance networks using distributed deterministic routing. *HiPC-15*, 2008.

[12] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28:42–53, May 2008.

[13] C. Fallin et al. CHIPPER: A low-complexity bufferless deflection router. *HPCA-17*, 2011.

[14] C. Fallin et al. MinBD: Minimally-buffered deflection routing for energy-efficient interconnect. *NOCS*, 2012.

[15] E. Gran et al. First experiences with congestion control in InfiniBand hardware. *IPDPS*, 2010.

[16] P. Gratz et al. Implementation and evaluation of on-chip network architectures. *ICCD*, 2006.

[17] P. Gratz et al. Regional congestion awareness for load balance in networks-on-chip. *HPCA-14*, 2008.

[18] Y. Hoskote et al. A 5-GHz mesh interconnect for a Teraflops processor. *IEEE Micro*, 27, 2007.

[19] InfiniBand Trade Association. InfiniBand architecture specification, release 1.2.1, 2007.

[20] Intel Corporation. Single-chip cloud computer. http://techresearch.intel.com/ProjectDetails.aspx?Id=1.

[21] J. A. Joao et al. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS-21*, 2012.

[22] J. Kim et al. A low latency router supporting adaptivity for on-chip interconnects. *DAC-42*, 2005.

[23] Y. Kim et al. ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers. *HPCA-16*, 2010.

[24] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO-43*, 2010.

[25] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. *ISCA-8*, 1981.

[26] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. *ISCA-24*, 1997.

[27] K. Luo et al. Balancing thoughput and fairness in SMT processors. *ISPASS*, 2001.

[28] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. *ISCA-36*, 2009.

[29] S. Muralidhara et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO-44*, 2011.

[30] G. Nychis et al. Next generation on-chip networks: What kind of congestion control do we need? *Hotnets-IX*, 2010.

[31] G. Nychis et al. On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects. *SIGCOMM*, 2012.

[32] S. Scott and G. Sohi. The use of feedback in multiprocessors and its application to tree saturation control. *IEEE TPDS*, 1990.

[33] A. Singh et al. GOAL: a load-balanced adaptive routing algorithm for torus networks. *ISCA-30*, 2003.

[34] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-9*, 2000.

[35] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006.

[36] Y. Tamir and G. Frazier. Dynamically-allocated multi-queue buffers for VLSI communication switches. *IEEE TC*, 1992.

[37] M. Taylor et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *MICRO-35*, 2002.

[38] M. Thottethodi et al. Self-tuned congestion control for multiprocessor networks. *HPCA-7*, 2001.

[39] 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.

[40] H. Vandierendonck and A. Seznec. Fairness metrics for multi-threaded processors. *IEEE Computer Architecture Letters*, 2011.

[41] H. Wang et al. Orion: a power-performance simulator for interconnection networks. *MICRO-35*, 2002.

[42] H. Wang et al. Power-driven design of router microarchitectures in on-chip networks. *MICRO-36*, 2003.

[43] D. Wentzlaff et al. On-chip interconnection architecture of the Tile processor. *MICRO-40*, 2007.

[44] S. Woo et al. The SPLASH-2 programs: characterization and methodological considerations. *ISCA-22*, 1995.