Aggregate Flow-Based Performance Fairness in CMPs

ZHONGHAI LU and YUAN YAO, KTH Royal Institute of Technology, Sweden

In CMPs, multiple co-executing applications create mutual interference when sharing the underlying network-on-chip architecture. Such interference causes different performance slowdowns to different applications. To mitigate the unfairness problem, we treat traffic initiated from the same thread as an aggregate flow such that causal request/reply packet sequences can be allocated to resources consistently and fairly according to online profiled traffic injection rates. Our solution comprises three coherent mechanisms from rate profiling, rate inheritance, and rate-proportional channel scheduling to facilitate and realize unbiased workload-adaptive resource allocation. Full-system evaluations in GEM5 demonstrate that, compared to classic packet-centric and latest application-prioritization approaches, our approach significantly improves weighted speed-up for all multi-application mixtures and achieves nearly ideal performance fairness.

CCS Concepts: • **General and reference** → **Performance**; • **Hardware** → **Network on chip**; • **Networks** → **Network on chip**; **Packet scheduling**;

Additional Key Words and Phrases: Computer architecture, performance fairness, quality of service

ACM Reference Format:

Zhonghai Lu and Yuan Yao. 2016. Aggregate flow-based performance fairness in CMPs. *ACM Trans. Archit. Code Optim.* 13, 4, Article 53 (December 2016), 27 pages.
DOI: <http://dx.doi.org/10.1145/3014429>

1. INTRODUCTION

Network-on-Chip (NoC) is emerging as a critical shared architecture for Chip Multi/Many-core Processors (CMPs) running parallel and concurrent applications. As the core count grows quickly, efficient and fair sharing of communication resources becomes even more urgent in order to provide high application performance and Quality-of-Service (QoS) while exploiting application-level parallelism. Broadly speaking, QoS refers to the overall quality or performance of a system offered to clients with possibly different requirements. It is a collective attribute offered to all applications. In general-purpose CMPs, QoS often means fairness (i.e., no application should be privileged for performance when sharing resources because all applications are of equal importance). Because the NoC in CMPs is typically a best-effort network without specialized often costly interference management schemes based on exclusive time slots [Goossens et al. 2005] or separate physical/virtual channels [Marchal et al. 2005], implementing QoS in CMPs calls for fair and efficient management of aggregately shared resources.

Looking into previous QoS studies (see details in Section 7, Related Work), we can observe the following two principles which are essential to a satisfactory performance fairness scheme in CMP architectures: (i) Since traffic initiated from an application is equitable, it is desirable that packets or packet batches belonging to the same class,

---

This work was supported in part by Swedish Research Council (Vetenskapsrådet), Grant No. E0509501.

Authors' address: Z. Lu and Y. Yao, Department of Electronic and Embedded Systems, School of ICT, KTH Royal Institute of Technology, 16440 Sweden; emails: {zhonghai, yuanyao}@kth.se.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/12-ART53 \$15.00

DOI: <http://dx.doi.org/10.1145/3014429>

thread, or application should be handled similarly by the routers end-to-end, from source to destination. Such *consistent packet handling* can leverage network resource allocation from being locally fair to globally fair and avoid starvation [Das et al. 2009]. (ii) Since CMP applications produce heterogeneous traffic with variable characteristics over time, it is opportunistic for resource allocation methods to *exploit application dynamism* to be efficient and facilitate fairness. For example, application-level measures such as stall-time criticality and program slowdown are exploited in Das et al. [2009] and Ebrahimi et al. [2010], respectively; network-level metrics such as consumed bandwidth per interval and packet latency slack are exploited in Grot et al. [2009] and Das et al. [2010], respectively.

Combining the two principles, our performance fairness technique aims to perform consistent handling of packets using their runtime characteristics in order to be both fair and efficient. This triggers a few questions: (i) What and how do we effectively *profile* traffic characteristics at runtime? (ii) How do we *propagate* the runtime traffic characteristics for consistent resource allocation, particularly, when traffic request-reply causality under cache coherency should be respected? (iii) How do we *exploit* the runtime information in routers to achieve efficient and fair channel bandwidth allocation?

While addressing these three logically related questions, we seek for a systematic approach. We first introduce the concept of *aggregate flow* as sequences of request-reply messages directly or indirectly generated from the same thread. This concept leverages the notion of traffic flow with runtime-profiled traffic characteristics so that packets from the same source are handled by routers in a similar fashion. More significantly, it allows us to conveniently take *traffic causality* into consideration. In on-chip architectures, communication is bidirectional, with one way being requests and the other way being replies. Also, requests may further trigger next-level requests through the memory hierarchy. Since replies and next-level requests are consequences of initial requests, it is desirable that fair resource allocation respects the causality relationship, offering consistent service to both request and reply packets at different levels.

Based on the aggregate flow concept, we develop three coherent mechanisms that answer the three respective questions. Specifically, (i) *rate profiling* estimates a thread's message generation rate (ρ) from an L1/L2 cache, which, together with packet size (ℓ), captures the program runtime network access demand using a shifting window-based characterization and prediction. (ii) *Rate inheritance* allows a data or coherence reply flow to inherit the rate characteristics of its associated request flow for achieving consistent network resource allocation in the presence of cache coherency through memory hierarchy. And, (3) *flow arbitration* is directed toward our hardware-efficient rate-proportional scheduling policy *Self-clocked Limited Fair Queueing (SLFQ)*, adapted from SCFQ [Golestani 1994] by replacing per-flow buffering with shared buffering to fairly schedule flows in the best-effort NoC in proportion to their runtime rates.

In the remainder of this article, Section 2 exemplifies application and thread runtime dynamism. In Section 3, we present the aggregate flow concept in multicores. Section 4 first gives an overview and then describes the three mechanisms of our technique. In Section 5, we detail the SLFQ-supported flow router. Section 6 reports experiments and results. After discussing the related work in Section 7, we conclude in Section 8.

2. MOTIVATION

2.1. Target Architecture

Figure 1 shows a typical 64-node CMP. Each node has a private L1 cache and a portion of L2 cache, which is distributed and shared across all nodes. Main memory ranks are

Fig. 1. An 8×8 mesh CMP with flow-supported FNI and FR.

Fig. 2. Injection rate variations in (a) different applications and (b) different threads of the same application.

connected to the four central nodes in the first and last rows for structural symmetry. Each node consists of a Flow Network Interface (FNI) and a Flow Router (FR)¹ to support our aggregate flow rate profiling, rate inheritance, and flow arbitration. The routers are interconnected in a popular mesh topology on which the XY routing algorithm is used for simplicity and deadlock avoidance. In this article, we consider multithreaded applications with one core executing one thread. The CMP may run either applications individually or multiple applications concurrently. When running an application stand-alone, the cores are exclusively used by the application, generating as many threads as the number of cores. When running multiple applications together, the cores are equally shared.

2.2. Application Dynamism

To expose the importance of exploiting dynamic application characterization, we investigate the communication dynamism of multithreaded workloads from PARSEC [Bienia et al. 2008] under the GEM5 default full-system simulation. We select two representative programs – *bodytrack* for memory nonintensive application and *freqmine* for memory intensive application – to run stand-alone on the 64-node 8×8 mesh CMP without flow concept. See configuration details in Table I of Section 6.

Figure 2(a) shows the average request packet injection rates over all cores for *bodytrack* and *freqmine* programs in the first 80 simulation windows, with each window consisting of 1 million simulation cycles. For long-term per-core average injection rate, *freqmine* (0.61 packets/cycle) gives a 50% increase over *bodytrack* (0.41 packets/cycle).

¹If no flow concept is supported, each node comprises an ordinary NI and router.

Fig. 3. Cache actions and causal messages generated from a write miss.

This is due to the fact that, being more memory-intensive, *freqmine* encounters more frequent cache misses, thus more requests are issued to the caches and main memories via the on-chip network. However, when we track the injection rate through the time line, we do observe that during some internals, for example from cycle 55×10^6 to 75×10^6 , the injection rate of *bodytrack* exceeds that of *freqmine*. This shows that an application's communication demand, with respect to cache/memory usage and packet injection rate, varies significantly during program runtime.

2.3. Thread Dynamism

The program runtime dynamism varies not only in different applications, but also in different threads of the same application. To illustrate this phenomenon, we look into the request packet injection rates of two threads, *thread₀* on node 0 (coordinate (0, 0) on the mesh) and *thread₂* on node 2 (coordinate (2, 0) on the mesh), from *freqmine* in Figure 2(b). Since each thread in *freqmine* has largely the same long-term behavior, the overall injection rates of the two threads do not have much difference. However, when examined in finer grained time intervals, the injection rates of the two threads vary. For example, from cycle 25×10^6 to 40×10^6 , the injection rate of *thread₀* falls behind *thread₂*. However, in the succeeding cycles from 40×10^6 to 50×10^6 , the injection rate of *thread₀* exceeds that of *thread₂*. This shows that different threads in the same parallel program also have differences in network access behaviors.

Both these examples exhibit that, as an application program executes, its network packet injection rate typically varies from phase to phase, from time to time. A memory-intensive application has a higher packet rate than a compute-intensive application in the long run, but often not at each time window. The injection rate variation occurs not only for applications but also for threads of the same application. In this article, such variations will be profiled at runtime and used to guide the fair and efficient network resource allocation. Before presenting the injection rate profiling in Section 4.2, we explain the *aggregate flow* notion in cache-coherent CMPs.

3. AGGREGATE FLOW CONCEPT IN CACHE-COHERENT CMPs

3.1. Message Causality in CMPs

To motivate aggregate flow-based fair resource management in a cache-coherent context, we first look into message generation causality in response to thread behavior and cache actions. As an example, we assume write-allocate write-back caches and cache coherency maintained for L1 caches using the directory-based MOESI protocol.

In cache-based CMPs, message production of a thread is triggered by either cache write hit or read/write miss. Suppose a **L1 write miss** to a block being shared with m copies, as shown in Figure 3. The actions due to the L1 write miss have two phases: (1)

Fig. 4. The flow concept, with three example flows and an example aggregate flow.

Write allocate: Upon L1 write miss, an L2RdEx is sent to the corresponding L2 cache bank (home node). Supposing that the home node also encounters a read miss, a memory data request MemRd is then sent to the memory rank, bringing a valid data copy back via MemData and causing a write-back (MemWB) to the memory. (2) **Cache coherency phase:** After receiving the valid data copy, the home node sends Invalidation (Invs) to other data-sharer L1 caches (remote L1s) in parallel. Upon receiving the Inv requests, $L1(R_1)$, $L1(R_2)$, to $L1(R_m)$ send acknowledgments (Acks) back to the home L2 cache. Notation $L1(R_m)$ stands for L1 of a remote node R_m . After receiving all Acks, the home node forwards the data block L2Data to the requesting L1 for write, which further triggers an L2WB write-back to the home L2 and concludes the transaction.

The time line of the preceding transactions is also shown in Figure 3. We observe that the requesting thread generates one initial data request message (L2RdEx), which in turn generates another request message (MemRd) and another subsequence (MemData, MemWB, Invs and Acks, L2Data, L2WB) for data updates and coherency maintenance. During a program's execution, these $2m + 6$ messages are continuously generated and transferred as sequences of packets from their sources to destinations via the NoC. Although these messages are generated at different times to different destinations through different paths, they have one thing in common: All of them are either directly originated or indirectly rooted from the same thread, with causality between a message and its successor (as illustrated by the dashed lines in Figure 3).

While these messages are delivered in the network, conventional NoC routers allocate resources (buffers and channel bandwidth) to these messages independently of each other and obviously to the communication demand of their issuing thread. To avoid unfairness in the demand-oblivious treatment, it is necessary that messages from the same thread or serving the same cache action receive consistent on-demand service both temporally (no matter when the messages are generated) and spatially (no matter which router serves these messages). Furthermore, because of message causality, it is also necessary that succeeding messages (e.g., reply messages) enjoy consistent service with their preceding messages (e.g., request messages). This motivates us to extend previous flow concepts [Zhang 1995; Nesbit et al. 2006] in pursuing end-to-end packet scheduling that are both temporally and spatially consistent.

3.2. Notions of Flow and Aggregate Flow

Figure 4 illustrates the concept of flow as a sequence of messages directly or indirectly generated from the same thread or cache action. Figure 4(a) shows three example flows, one for an L1 read miss then an L2 read hit, one for an L1/L2 read miss to a nonshared data block, and one for an L1/L2 write miss to a shared data block. In each case, we omit the write-back messages for clarity. As shown in Figure 4, a flow associates causal messages serving the same thread or cache action together, thus sharing the

issuing thread's runtime communication characteristics among them and facilitating consistent scheduling for all messages in the same flow.

To support the flow concept and the flow communication demand-based scheduling, we need only small modifications to the traditional packet header. For simplicity, we show only the case in which one message is encapsulated into one packet, which may contain one or multiple flits. In addition to the common information fields such as packet source, destination, and type (indicating if the packet is either a *data* or a *coherence control*), we attach three *new* fields to each packet header, as shown in Figure 4(a), which carry the thread's runtime communication characteristics:

- FID: The unique flow identifier, which records the ID of a flow's issuing thread, denoting to which thread the messages in a flow belong. FID keeps the same value for all messages in a flow.
- ℓ : The packet size in flits, denoting how much service each packet in a flow needs to receive per router. Typically, short messages (without data, for coherence control) can be contained in single-flit packets and long messages (with data) in multi-flit packets. See Table 1 in Section 6 for example.
- ρ : The network request message injection rate of an issuing thread in messages per cycle, denoting how frequently a thread injects packets into the network. Because not every L1 request message will cause an L2 request message (e.g., an L2 read hit does not cause an L2 request message), for L1/L2 request messages in a flow, their ρ values are separately profiled at L1/L2 during program runtime (Section 4.2). As shown in Figure 4(a), all messages associated with L1 cache requests (L2Rd, L2RdEx, L2Data) are featured with the L1 message injection rate ρ_1 , and all messages associated with L2 cache requests (MemRd, MemData, etc.) are featured with the L2 message injection rate ρ_2 . Furthermore, for L1/L2 reply messages in a flow, for consistent service, their ρ values are inherited from corresponding L1/L2 request messages (Section 4.3).

We note that, if each outstanding data request produces an individual flow, the total number of flows in the CMP can be very large. This is because, by allocating an entry in the Miss Status Holding Register (MSHR), each core can execute multiple outgoing data transactions, issuing multiple flows in parallel instead of waiting for one flow to finish. For example, suppose that each core has an MSHR consisting of 32 entries; thus, there are maximally $32 \times 64 = 2048$ possible flows in a 64-core CMP. To reduce the number of flows, we propose the concept of *aggregate flow*, which further associates flows issued from the same thread together. In this way, one aggregate flow consists of multiple flows, and the maximum number of aggregate flows in a CMP is restricted to the number of cores. Figure 4(b) shows an example aggregate flow in which we simplify the presentation of flow for clarity. As shown in Figure 4(b), suppose that a core encounters two L1/L2 write misses to two shared data blocks, one L1/L2 read miss to a nonshared data block, and three L1 read misses then L2 read hits. In total, the core issues 6 flows, which, however, belong to the same aggregate flow.

To clarify our flow notions, Figure 5 shows the relationships of four packet flow concepts, highlighting their evolution. Figure 5(a) is the basic packet flow concept used in open macro-networks [Parekh and Gallager 1993; Sariowan et al. 1995; Zhang 1995; Bennett and Zhang 1997]. In offering guaranteed network services to timing-sensitive applications such as audio/video and multimedia, the concept of packet is leveraged to that of flow as a stream of continuous packets, such that consistent service can be provided to all packets in the flow and the maximum packet delay/jitter can be determined. Since the macro-network is an open system, the traffic characteristics of open flows is independent from the service characteristics. For example, the bit rate of a multimedia stream is not influenced by service latency. In Figure 5(a), the flow rate ρ only depends on its source. In multicore systems, Nesbit et al. studied

Fig. 5. Previous flow concepts [(a) and (b)] and our extended flow concepts [(c) and (d)].

fair-queuing scheduling for thread-level flows to provide performance isolation to multiple applications in a shared SDRAM system in which the flow injection from a thread is influenced by the memory service [Nesbit et al. 2006]. We call this *round-trip flow*, as shown in Figure 5(b), where the request flow rate ρ is subject to the reply rate ρ' . Our flow concept in Figure 5(c) enriches the round-trip flow with propagation semantics by considering request propagation (i.e., a request may trigger next-level request(s) via intermediate node(s) down to the destination through the memory hierarchy and cache coherence). As illustrated in Figure 5(d), our aggregate flow concept further takes all propagation flows from the same source as an aggregate to share the flow ID and corresponding rate information.

4. FLOW-ORIENTED MECHANISMS

4.1. Overview

After introducing the flow and aggregate flow concepts, we give an overview of our flow-based framework comprising three coherent mechanisms: *rate profiling*, *rate inheritance*, and *rate-proportional arbitration*. Figure 1 shows its architectural support in the NI and router. Since the NI is enhanced to support rate profiling and inheritance, we rename it *Flow NI* (FNI). Because the router realizes a rate-proportional channel allocation, we rename it *Flow Router* (FR).

- (1) **Rate profiling:** Profiling a traffic source's network access rate ρ at runtime per epoch by the FNI. For the target CMP, it is profiled per L1/L2 cache, and all packets from the L1/L2 cache carry the profiled ρ_1/ρ_2 , respectively.
- (2) **Rate inheritance:** At the FNI, a reply flow inherits its corresponding request flow's rate to achieve consistent network scheduling throughout its transmission.
- (3) **Rate-proportional arbitration:** The FR uses the flow information (both ℓ and ρ) to realize our SLFQ scheduling policy, which achieves fair and consistent rate-proportional service in a best-effort network without per-flow buffering.

Next, we detail rate profiling and rate inheritance and explain the principle of SLFQ.

4.2. Rate Profiling

Rate characterization: In typical CMPs, ρ indicates how frequently a thread exchanges data with shared on-chip L2 caches and memory ranks via the NoC. Since each core has its own L1 cache, both an L1 read hit and an L1 read miss followed by a local L2 read hit incur only local actions and thus do not produce packets to the network. Network request packets are generated from other types of cache actions, such as write hit to a shared data block (cause of coherent Inv/Ack messages), read miss at local L2, and read/write miss from L1 to a remote L2 cache.

To get the ρ value, we record the total number of MSHR entries of the L1/L2 cache for each sampling time window (epoch) with length L_{sw} . This can accurately reflect a

source's communication workload because, in state-of-the-art nonblocking/lock-up-free caches, when an L1/L2 cache miss occurs, an access request to the L2/DRAM is created by first allocating an MSHR entry [Hennessy and Patterson 2011]. Similarly, in the case of write hit to a shared data block with a coherency requirement, an MSHR entry is allocated to track the status of Inv/Ack for transaction completion. The number of MSHR entries allocated to a thread directly represents the total number of outstanding requests issued by that thread to the L2 cache or the DRAM. At each sampling window L_{sw} , two counters, namely C_1 and C_2 , which are initialized to 0 at the beginning of each sampling window, are maintained to record the total number of MSHR entries for the L1/L2 cache, respectively. When a new L1/L2 MSHR entry is allocated, C_1 or C_2 is increased by 1. An existing entry is deallocated when a corresponding transaction completes. At the end of each sampling window, ρ_1 is computed for L1 cache request flows by C_1/L_{sw} and ρ_2 for L2 cache request flows by C_2/L_{sw} .

To continuously sample ρ , the window is shifted forward with an epoch T after each characterization. We set T to L_{sw}/N , where N is a natural number called the *overlapping factor*. If $N = 1$, two consecutive sampling windows have no overlap; if $N = 2$, two consecutive sampling windows overlap with a length of $L_{sw}/2$. The overlapping of consecutive sampling windows (when $N \geq 2$) ensures that the window-by-window characterization results enjoy high continuity. Due to the shifting window mechanism, the valid period for each characterized ρ value is one epoch T .

Rate prediction: Although the flow characterization can profile a thread's network access rate continuously, it can only measure *past* events. To guide *future* packet delivery, we need a flow's future characteristics. To address the dilemma, as inspired by Lu and Wang [2012], we use a simple yet effective state-aware *prediction* mechanism. Taking as input the rate output from the rate characterization, the task of rate prediction is to speculate $\rho(T_{n+1})$ based on previously calculated results $\rho'(T_n)$, $\rho'(T_{n-1})$, $\dots$, where n is the sequence number of epoch. Specifically, $\rho'(T_{n-1})$ and $\rho'(T_n)$ represent the resource access rate in the previous $(n-1)$ th and current n th epoch, and the speculated $\rho(T_{n+1})$ predicts the network access rate during the next $(n+1)$ th epoch. The prediction is done as follows:

$$\rho(T_{n+1}) = \begin{cases} \rho'(T_n) + \Delta\rho'(T_n) & : \rho'(T_n) + \Delta\rho'(T_n) > 0 \\ \rho'(T_n) & : \rho'(T_n) + \Delta\rho'(T_n) \leq 0 \end{cases} \quad (1)$$

The predicted $\rho(T_{n+1})$ is composed of a base value $\rho'(T_n)$ and an offset value $\Delta\rho'(T_n) = \rho'(T_n) - \rho'(T_{n-1})$, which captures possible rate variation and is used to avoid abrupt change in $\rho(T_{n+1})$. This approach has low implementation overhead and is thus suitable for hardware implementation. In Figure 1, the rate profiling is implemented by the *characterizer* for rate characterization and the *predictor* for rate prediction at the source FNI.

4.3. Rate Inheritance

Rate inheritance is a process by which a reply message inherits the rate characteristics of its corresponding request message. After network access rate ρ_1/ρ_2 of L1/L2 cache² is obtained, the data and coherency request messages will carry this information in their packet headers. For example, upon write hit to a coherent L1/L2 cache, generated Inv(s) packets will carry the corresponding ρ_1/ρ_2 . When the request packets are received, their ρ_1/ρ_2 will be extracted and inherited by their associated Ack(s) packets.

Corresponding to data and coherency messages, there are two kinds of inheritance (" $\rightarrow$ ") relationship: *data request* $\rightarrow$ *data reply*, and *coherence request* $\rightarrow$ *coherence reply*. In a two-level cache architecture, depending on whether a request message is produced

²Strictly speaking, it is L1/L2-associated MSHR.

Fig. 6. Request-Reply rate inheritance at multiple levels.

as a result of L1 or L2 action, the *data request* $\rightarrow$ *data reply* inheritance can be further divided into *L1 to L2 data request* $\rightarrow$ *L2 to L1 data reply*, and *L2 to Memory data request* $\rightarrow$ *Memory to L2 data reply* inheritance. Similarly, the coherence request-reply inheritance (Inv $\rightarrow$ Ack) can be expanded to different levels of caches with coherence requirements. In the case of write-back, the write-back data packet inherits the rate of its respective triggering packet; that is, L2Data/MemData $\rightarrow$ L2WB/MemWB.

Using the write miss example in Figure 3, Figure 6 shows the inheritance relationship among different messages. Since the L1 to L2 data request is due to the L1 action, the request message follows the network access rate of L1, ρ_1 . Similarly, the L2 to memory request message uses the network access rate of L2, ρ_2 . The figure also illustrates how the rate information of the request messages, including both data request and coherence request, is propagated and inherited by their associated reply messages. Such inheritance maintains a consistent relation between a request and its reply, ensuring that the end-to-end (not only source-destination-wise, but also round-trip protocol-wise at multiple levels) resource allocation is consistent.

4.4. Flow Arbitration with SLFQ

Our flow-based rate-proportional resource allocation aims to fairly allocate channel bandwidth to contending flows according to their dynamically profiled message injection rates. Among many service disciplines in the literature [Zhang 1995; Grot et al. 2009; Lee et al. 2008; Ouyang and Xie 2010], *Weighted Fair Queueing* (WFQ) [Demers et al. 1989] excels in providing fairness, limiting interference, and being adaptive to dynamic traffic scenarios. When multiplexing flows onto a shared channel, WFQ is work-conserving and allocates a weighted portion of channel bandwidth according to, for example, each flow's rate. Specifically, the principle is to divide the channel bandwidth B among a set of *backlogged* flows³ $G(t)$, with each flow $f_i \in G(t)$ getting a share in proportion to its rate ρ_i . Hence, a backlogged flow f_i receives service rate r_i :

$$r_i = \frac{\rho_i}{\sum_{j \in G(t)} \rho_j} B. \quad (2)$$

Since $G(t)$ is varying, r_i also varies over time and could be larger or smaller than ρ_i , showing WFQ's adaptivity to different traffic contention scenarios.

Although offering adaptive rate-proportional fairness, WFQ incurs high complexity in implementation since it relies on virtual time to keep track of service progress [Zhang 1995]. To reduce implementation complexity, *Self-Clocked Fair Queueing* (SCFQ) approximates WFQ by recording only the finish time of the packet currently being serviced without maintaining the service progress [Golestani 1994]. Nevertheless implementing SCFQ in hardware is still too expensive because it requires per-flow buffering.

³A flow is *backlogged* at time t if it has flits queued at time t .

Fig. 7. Comparison of FCFS, RR, and SLFQ scheduling policies.

To make SCFQ amenable for cost-effective on-chip implementation, we adapt SCFQ from exclusive per-flow buffering (N flows N buffers) to shared limited buffering (N flows M buffers, where $M \leq N$), which we call SLFQ. This adaptation can greatly reduce router complexity while bringing about two consequences: (i) Buffer allocation: A flow needs to be dynamically allocated to a downstream buffer before advancement. (ii) Buffer sharing: Packets from multiple flows may interleave in a buffer, causing head-of-line blocking. Although losing guaranteed service to each flow in SCFQ, SLFQ is intended for best-effort fairness in an NoC. Because both consequences are typical features of on-chip routers offering best-effort services, SLFQ is suitable for CMP NoCs. Limited buffering also means limited fair queuing because the maximum number of active flows ready for bandwidth allocation is limited to that of buffers.

Figure 7 illustrates how SLFQ achieves fair bandwidth allocation and interference containment. For comparison purposes, we also examine two other widely used work-conserving scheduling policies: First-Come-First-Served (FCFS) and Round Robin (RR). Figure 7(a) qualitatively illustrates the difference between the three scheduling policies, and Figure 7(b) quantitatively compares the corresponding average/maximum packet delay and flow transmission finish time.

As shown in Figure 7(a), four flows share an output link. For simplicity, we assume single-packet messages, single-flit packets, and transmitting one packet takes one cycle. Packets in the three higher injection rate flows f_1^H , f_2^H , and f_3^H have the same (ℓ, ρ) values of $(1, 0.2)$, indicating packet size of 1 flit and average injection rate of 0.2 packet/cycle. Packets in flow f_4^L have (ℓ, ρ) values of $(1, 0.1)$, representing a lower injection rate flow. Flow f_1^H , f_2^H , and f_3^H injects 8 packets via the east, west, and north input link, respectively, and flow f_4^L 4 packets via the south input link. The packet arrival and departure of the four flows are shown in Figure 7(a). Realizing FIFO fairness, FCFS allocates bandwidth only on packet arrival order, leading the lower rate flow f_4^L to complete at cycle 23 due to intensive arrivals of higher injection flows. Moreover, although RR achieves certain interference isolation among flows, it ignores per-flow packet rate and service history, attaining an equal but unfair share of the total bandwidth to each competing flow. For example, in Figure 7(a), although the three higher injection rate flows f_1^H , f_2^H , and f_3^H receive service earlier than f_4^L , when the lower injection rate flow f_4^L arrives at cycle 12, both the higher and lower rate flows are equally served with rate r of 0.25 packet/cycle, thus delaying f_4^L 's finish time to cycle 26.

Being different from FCFS and RR, SLFQ proportionally allocates bandwidth to each flow according to (i) the per-flow characterized packet injection rate and (ii) the service history of each flow. Counting from its arrival by a self-clock (as illustrated by ⌚ in Figure 7(a)), every competing flow is served with a bandwidth share that is proportional to its characterized injection rate. In SLFQ, an early arrival higher injection rate flow does not affect the service of a later but lower injection rate flow. This is because, by intensively injecting packets, the early arrival higher rate flow receives a service rate that exceeds its rate-proportional bandwidth share. In this case, when the lower rate flow arrives, the higher rate flow postpones its next service to ensure that the lower rate flow also enjoys its rate-proportional bandwidth share. For example, at cycle 12 in Figure 7(a), because the router has transmitted 3 of f_1^H 's, 3 of f_2^H 's, and 5 of f_3^H 's packets, the first two packets (denoted α and β) of flow f^L are scheduled next. As such, SLFQ benefits the lower rate flow f^L to finish at cycle 18, greatly accelerating its transmission without significantly delaying the other higher rate flows.

The effectiveness of SLFQ is further confirmed by the quantification results in terms of average/maximum packet delay and flow transmission finish time shown in Figure 7(b). For clarity, we use $f_{1,2,3}^H$ to denote the averaged results over the three higher injection rate flows f_1^H , f_2^H , and f_3^H . Across the figures, we consistently observe that SLFQ effectively expedites packets from the lower injection rate flow f^L , largely reducing its average/maximum packet delay and flow transmission time compared to FCFS and RR, without significantly impacting the higher rate flows.

5. FLOW ROUTER

5.1. Flow Router Architecture

We develop a flow router to serve flows with the proposed SLFQ scheduling policy. Our flow router is based on the classic input-buffering Virtual Channel (VC) router. Figure 8 sketches its architecture partitioned into data path and control path. On the data path, there are mainly multiple VCs per input channel and a crossbar. On the control path, there are Route Compute (RC), VC Allocation (VA), virtual Finish time (f_v) Computation (FC, details in Section 5.2), and SLFQ-based Switch Allocation (SA). As highlighted in Figure 8, a flow router uses the flow state table, the f_v computation logic, and the SLFQ-based switch allocator to work cooperatively to realize SLFQ.

When a packet header reaches a VC, the flow router records the flow's characteristics (FID, ℓ , ρ) into a flow state table entry. Both parameters (ℓ , ρ) are used to compute f_v for each packet. After completing the VA, SA compares the f_v of contending packets and determines their delivery order by switching a packet with lower f_v first.

5.2. Router Pipeline and Pipeline Stages

Like the speculative router [Peh and Dally 2001], our flow router has a two-stage pipeline in which the first stage performs RC, VA, and SA in parallel and the second stage ST (Switch Traversal), as shown in Figure 10. To realize SLFQ without increasing the pipeline, we include FC in the first pipeline stage in parallel with RC, VA, and SA.

Virtual finish time computation (FC): Like WFQ/SCFQ, SLFQ can be realized by a *sorted priority queue* mechanism [Zhang 1995; Rexford et al. 1996]. In such a mechanism, each flow computes a *virtual finish time*, f_v , representing a dynamic priority to determine its service order. Upon arrival of each packet, a flow's f_v is updated according to (i) the flow's own traffic characteristics and (ii) the *system virtual time*, F_v , estimated from the f_v value of the packet currently in service. Competing flows are then served in the order of increasing f_v values. Specifically, f_v is calculated by:

$$f_v^{i,k} = \max\{f_v^{i,k-1}, F_v(i_a^{i,k})\} + \ell^{i,k}/r_i, \quad (3)$$

Fig. 8. The flow router architecture.

Fig. 9. SLFQ-based switch allocator.

Fig. 10. 2-stage pipeline.

where $f_v^{i,k}$ is the f_v value of flow i when the k th packet arrives, $t_a^{i,k}$ the arrival time of the k th packet of flow i , r_i the bandwidth (flit/cycle) allocated to flow i in the rate-proportional fashion (Equation (2)) and $\ell^{i,k}$ is the packet length in flits. As shown in Equation (3), the f_v of a flow is the service start time ($\max\{f_v^{i,k-1}, F_v(t_a^{i,k})\}$) plus the service time ($\ell^{i,k}/r_i$). The service start time is usually the f_v of the last served packet in the same flow ($f_v^{i,k-1}$); if no packets in the flow were served in the last busy period, the service start time is then the system virtual time $F_v(t_a^{i,k})$ at the k th packet arrival moment $t_a^{i,k}$, estimated from the virtual finish time of the packet currently being serviced. Initially, $F_v(t_a^{i,k})$ equals 0.

Lookahead RC: FC may result in a new pipeline stage if no proper measure is taken since computing f_v depends on the results of RC to know competing flows and get their characteristics. To remove f_v computation from the critical path, we perform look-ahead RC as detailed in Peh and Dally [2001]. To do this, while a flow header is traversing through a router, RC is not performed for this router but for the *next*, and the result is passed along with the header flit. Thus, when a flow reaches the next router, it has already selected the output port so that RC and FC can begin in parallel.

Flow-aware VA: Similar to the base packet router, the flow router conducts VC flow control at the network level. VCs are allocated in round-robin. At the link level, it uses credit-based flow control to avoid buffer overflow in the downstream router and propagate back-pressure to the upstream router. Being different from packet routers, the flow router allocates VC in awareness of flows. It maintains the states of buffered

flows using additional flow state tables. There is one such table associated with each input port, and, in each table, one entry corresponds to one flow, recording the flow's (ℓ, ρ, f_v) states. The number of entries equals the VC number.

SLFQ-based SA: In the SA stage, any VC that contains buffered flits and has allocated the downstream VC available bids for the switch traversal. As shown in Figure 9, the SLFQ-based switch allocator maintains a sorted priority queue using a circular buffer for each output port to record the service order of flows. Each priority queue consists of an f_v comparison logic, a circular buffer, and a selection signal decoder. When a packet of a flow enters the SA stage, the allocator first consults the flow state table to find the associated f_v value of the corresponding flow. It then determines the service order of the packets inside the priority queue by comparing the f_v value with that of all other contending flows. As the VA and SA are conducted at the packet level, our flow router performs virtual cut-through.

To reduce complexity, we adopt the low-latency priority comparator in Grot et al. [2009], which uses a binary comparison tree based on fast adder circuits. The comparison results are then recorded into the circular buffer. To reduce storage expenses, the circular buffer stores only pointers to the flits of packets, but not the flits themselves. When a flit reaches the head of the priority queue, selection signals are generated by the selection signal decoder, driving the switch to transmit the flit out.

5.3. Avoid Per-Flow Buffering and Deadlock

Per-flow VC avoidance: Because buffers consume a significant portion of area and power in on-chip routers [Park et al. 2012], it is important to minimize the buffering cost. In the best-effort CMP environment, our flow router utilizes a small number of VCs instead of per-flow VC (i.e., no exclusive VCs for all active flows passing through a router). The question is: What happens when there is no available VC for a flow? Similar to packet routers which do not need per-packet VC, when all VCs in a router are busy, back-pressure is propagated backward to the upstream routers, where flows keep waiting until the downstream router has free buffers. If the back-pressure reaches the source of a flow, the packet injection temporarily stops, to help the network recover from resource insufficiency.

At this point, we make notes on the flow rate ρ , which is used as a *reference* to indicate estimated flow communication intensity. In the best-effort network, it might reflect neither real flow arrival rate nor flow service rate because flow injection is affected by back-pressure and flow service depends on contending flows. It does, however, represent a *weight* for dynamic fair allocation of channel bandwidth. As such, the actual rate value carried in packet headers may even be amplified in accordance with the criticality of memory access, thread, and application in order to provide differentiated services to expedite particular flows. This aspect may be investigated in future work.

Deadlock avoidance: Network deadlock occurs when two or more flows wait on one another to release their occupied resources. In the 2D mesh NoC, the XY dimension-order routing ensures the network is free from deadlock. To avoid protocol deadlock in the multicore architecture, it is necessary not to mix read/write request messages, read/write reply messages, and protocol messages (invalidate, acknowledgment) in one VC to avoid potential mutual blocking. For this purpose, our network is organized as three logical networks, with each logical network owning an exclusive subset of VCs but sharing physical channels. One logical network is only used by read/write requests, another one by read/write replies, while the third is used only by coherence messages.

5.4. RTL Implementation of SLFQ Scheduler

To evaluate the hardware complexity of the SLFQ router, we focus on SLFQ-related functions. Since the router structure follows the classic input-buffering router, the

remaining functions are pretty normalized. To give an accurate estimation, we made an RTL implementation of an SLFQ scheduler including the FC, the circular buffers, and the SLFQ arbiter. We employed Synopsis Design Compiler for logic synthesis and CACTI 6.0 [Muralimanohar et al. 2007] for evaluating circular buffers as on-chip SRAM. In both Design Compiler and CACTI, the technology was set to 45nm with voltage 1V and operation temperature 350K.

We optimize FC, since it is critical for the router speed due to its running in parallel with RC, VA, and SA stages. FC computes virtual finish time, f_v , of packets using Equation (3), which is relatively computation intensive. Although the equation contains division, it can be implemented efficiently using shift operation. Bringing Equation (2) into Equation (3), we have $\frac{1}{r_i} = \frac{\sum_{j \in G(i)} \rho_j}{\rho_i B}$. Let $B = 1$, we set valid ρ values as a set of 2^{-n} , where n is a natural, so as to realize division with shift. To prevent overflow when storing f_v in width-limited registers but still follow the SLFQ principle, the FC logic keeps track of relative rather than absolute virtual finish times of the packets in service.

Considering an SLFQ-scheduler with five circular buffers of depth 32 units each storing f_v in 32 bits, synthesis and compilation results show that the scheduler can operate up to 2.2GHz, occupying 22,646 μm^2 area in which 12,486 μm^2 is dedicated to circular buffers and 10,160 μm^2 to combinational logic including the FC and SCFQ arbiter.

Both SCFQ and SLFQ schedulers have the same algorithm complexity. They differ only in the number of circular buffers and associated control logic, which determine the exact area/cost difference. Suppose that 16 flows pass a router. An SCFQ scheduler requires 16 circular buffers to implement per-flow buffering, whereas an SLFQ scheduler keeps the number of circular buffers constant (e.g., 5). Our evaluation results show that the SCFQ scheduler (16 circular buffers) consumes 81,202 μm^2 area, in which the area for circular buffers increases to 40,597 μm^2 and for combinational logic 40,605 μm^2 . Thus, the SLFQ scheduler (5 circular buffers) achieves a $3.58\times$ greater area reduction than the SCFQ scheduler.

6. EXPERIMENTS AND RESULTS

6.1. Evaluation Methodology

Experimental platform and benchmark setup: We implemented and integrated our framework within full-system simulator GEM5 [Binkert et al. 2011], in which the built-in network GARNET [Agarwal et al. 2009] was revised with the flow network interface and the flow router. More simulation platform configurations are listed in Table I. For benchmarking, we use the PARSEC suite [Bienia et al. 2008], which includes emerging applications in recognition and data mining, as well as system applications mimicking large-scale multithreaded commercial programs. To scale the PARSEC benchmarks well to 64 cores, we chose large input sets (simlarge) for all programs. For data validity, we only report results obtained from the parallel execution phase called Region-of-Interest (ROI) in the experiments. Based on the data in Bienia et al. [2008], we divide all 12 PARSEC programs into two halves, with the first half consisting six applications (*blackscholes*, *bodytrack*, *facesim*, *ferret*, *swaptions*, *vips*) exhibiting low network packet injection rates and the other half (*canneal*, *dedup*, *fluidanimate*, *freqmine*, *streamcluster*, *x264*) high network packet injection rates.

Comparison studies: During evaluation, we made *incremental* comparisons from a baseline to a representative ideal case in order to better understand the benefits of the valid combinations of our three mechanisms in rate profiling, rate inheritance, and SLFQ scheduling. We also compared them against a state-of-the-art technique, **STC**, which utilizes application-sensitive dynamic *Stall Time Criticality* for network

Table I. Simulation Platform Configuration

Item	Amount	Description
Processor	64 cores	Alpha based 2.0GHz processors with out-of-order execution. 32-entry instruction queue, 64-entry load queue, 64-entry store queue, 128-entry reorder buffer.
L1-Cache	64 banks	Private, 32KB per-core, 4-way set associative, 128B block size, 2-cycle latency, split I/D caches, 32 MSHRs.
L2-Cache	64 banks	Chip-wide shared, 1MB per-bank, 16-way set associative, 128B block size, 6-cycle latency, 32 MSHRs.
Memory	8 ranks	4GB DRAM, 512MB per-rank, up to 16 outstanding requests for each processor, 8 memory controllers.
NoC	64 node	8×8 mesh network. X-Y dimension routing. 128-bit datapath. Each node consists of 1 core, 1 private L1 cache, and 1 shared L2 cache, 1 flow router and 1 flow network interface. 2-stage pipeline flow router, 6 VCs per port, 16 flits per VC.
Coherency		Directory-based MOESI cache coherency protocol. One cache block consists of 1 packet with 8 flits. One coherence control message consists of 1 single-flit packet.
Characterizer	1 per node	10,000 cycles of sampling window, overlapping factor $N = 4$.
STC	–	8 ranking levels. Refresh all applications' ranks every 10,000 cycles (ranking interval: 10,000 cycles). Packets issued in each successive 1,250 cycles form a batch (batching interval: 1,250 cycles).

resource allocation to achieve high performance and performance fairness [Das et al. 2009]. In total, we realized and experimented with the following five comparative cases⁴ in GEM5:

(A) p-RR (packet-based Round Robin scheduling): As the *baseline*, the CMP uses the classic packet-based NoC with RR channel scheduling. RR is a popular locally fair policy widely used in industrial practices, for example, in the NoC of the latest MPPA many-core processors from Kalray [Dupont de Dinechin et al. 2014].

(B) STC (STC-based application ranking with packet batching): STC combines two mechanisms, application ranking plus packet batching, to realize priority-based resource allocation in routers. At first, packets are prioritized by application ranking according to runtime profiled private L1 Misses Per Instruction (MPI). Packets from an application with lower MPI implying lower network access rate are considered more stall-time critical and thus enjoy higher priority. To avoid starvation for low-priority packets, packet batching, in which packets are grouped into periodical batches, is employed to prioritize older batches over younger batches. Packets in the same batch and with the same rank are scheduled with an RR policy. The parameters used for STC in our experiments are listed in Table I.

(C) f-SLFQ (rate profiling and SLFQ scheduling): The CMP realizes the rate profiling and SLFQ channel scheduling features but *without* rate inheritance. When no rate inheritance is present, request and reply packet injection rates are separately profiled.

(D) f-SLFQ+IHT (rate profiling, SLFQ scheduling, rate inheritance): The CMP is fully featured with rate profiling, SLFQ scheduling, and rate inheritance. This is the design proposal of this article.

(E) f-WFQ+IHT (rate profiling, WFQ scheduling, rate inheritance). The CMP mimics WFQ channel scheduling with rate profiling and inheritance. Although WFQ might not suit on-chip networks due to high complexity, it somewhat represents an ideal fair resource allocation scheme and thus serves as a yardstick for our evaluations.

⁴Note that we did not make **p-SLFQ** and **f-RR** cases. Because SLFQ requires rate information in flows to operate, **p-SLFQ** is invalid. Since **RR** ignores the rate information, **f-RR** is meaningless.

Fig. 11. System IPCs of PARSEC.

Fig. 12. Packet delay histograms.

6.2. Single-Application Performance

Before presenting multiapplication performance, we report single-application performance (i.e., one application is executed stand-alone with 64 threads executing on the 64 cores, one thread per core). The purpose is to show that our technique will not slow down single-application performance but improve it.

Application IPC: Figure 11 reports system IPCs (sum of IPCs of all threads) of all benchmark programs. Our scheme increases system IPCs for all programs. Specifically, f-SLFQ raises the system IPCs of p-RR and STC by 9.8% and 7.5% on average and 12.1% and 10.7% at maximum, respectively. As a pure gain from rate inheritance, f-SLFQ+IHT further leverages the system IPCs of f-SLFQ by 4.8% on average and 7.0% at maximum. The ideal case of f-WFQ+IHT shows the possibility of further increasing the system IPCs of f-SLFQ+IHT only by 3.9% on average and 5.1% at maximum. For a single application, STC exploits packet batching only because its application ranking becomes ineffective. Since all threads conduct packet batching uniformly, the performance of STC is dominated by the underlying RR scheduling, which is less effective than SLFQ (Section 4.4) at exploiting thread dynamism (Section 2.3).

Packet delay distribution: To give insights into the IPC improvements, we look into network packet delay histograms of both low and high packet injection rate programs. Figure 12(a) shows the delay distribution of benchmark *bodytrack*, which has a low packet injection rate and thus low NoC utilization. We can see that the average delay of p-RR, STC, f-SLFQ, and f-SLFQ+IHT is 24, 24, 21, and 19, respectively. Also, p-RR has the longest tail while f-SLFQ+IHT has the shortest tail. Figure 12(b) shows the delay distribution of benchmark *canneal*, which has a high packet injection rate and is thus high NoC utilization. Since the network becomes more loaded, the average delay of p-RR, STC, f-SLFQ, and f-SLFQ+IHT becomes 51, 51, 47, and 41, respectively. Likewise, p-RR results in longest tail, while f-SLFQ+IHT has the shortest. Both cases exhibit that both the improved latency fairness of SLFQ over RR and consistent service allocation contribute to sharpen the latency distribution envelopes. The delay histograms can indeed explain the improved IPCs due to reducing the CPU stall time. For both programs, the delay histogram shape of STC is slightly different from but resembles that of p-RR; thus, STC and p-RR have the same average packet delay values.

6.3. Multiple-Application Performance

Figure of Merit: To quantify multiapplication performance, we consider both *throughput* and *fairness*. We adopt a commonly used metric, *weighted speedup* [Eyerhan and Eeckhout 2008; Snively and Tullsen 2000] denoted Θ , to measure system throughput. Θ computes the sum of IPC slowdowns of all co-running applications. Let IPC^{alone} be the IPC of an application when it is run alone and IPC^{shared} its IPC when it is run together with others. The relative IPC, $X_i = \frac{IPC_i^{shared}}{IPC_i^{alone}}$, is defined as *IPC slowdown* of application i , $i \in [1, M]$, where M is the number of co-running applications. Θ is then

Fig. 13. Average packet delay comparison for 6 two-program mixtures.

Fig. 14. Per-node IPC for jointly executing a mixture of 2 programs: *bodytrack* and *canneal* (Mixture-1).

calculated as follows:

$$\Theta = \sum_i^M X_i. \quad (4)$$

Although Θ can reflect the influence of resource sharing on IPCs, it cannot quantify the *variation of IPC slowdowns* of individual applications, which indicates the evenness of IPC slowdowns. This is important because a system could achieve high throughput (Θ) by unfairly starving one or two applications while benefiting all others.

To complement Θ , we introduce the *standard deviation of IPC slowdowns*, denoted Ω , as a *metric of fairness*. Let $\bar{X}$ be the average of X_i , $\bar{X} = \sum_i X_i / M$. Ω is defined as:

$$\Omega = \sqrt{\frac{\sum_i (X_i - \bar{X})^2}{M}}. \quad (5)$$

Apparently, the smaller the Ω , the better the fairness. $\Omega = 0$ means ideal fairness.

Two co-running applications: In the two-application experiments, we set up six comparative mixtures, with each mixture containing 1 high-injection and 1 low-injection application. For each application mixture, we compare the differences among p-RR (denoted A), STC (B), f-SLFQ (C), f-SLFQ+IHT (D), and f-WFQ+IHT (E).

Figure 13 shows the average packet delay achieved by the five schemes, which consistently illustrates that, across different application mixtures, f-SLFQ slightly sacrifices the average delay of high-injection applications but attains remarkable average packet delay reduction among low-injection applications, thus confirming the benefits of low-injection rate flow acceleration exhibited in Figure 7(b). The flow-inheritance mechanism further strengthens the effectiveness of f-SLFQ, making f-SLFQ+IHT closely approach the results of the ideal case f-WFQ+IHT.

Figure 14 shows the IPC results for an example benchmark mixture, *bodytrack* (low injection rate program) and *canneal* (high injection rate program). Each program consists of 32 threads that are evenly distributed on half of the 64-core CMP. In Figure 14(a), by unfairly providing the same service rate to both high- and low-injection applications, p-RR benefits *canneal* with higher IPCs, leaving *bodytrack* suffering lower results. With STC in Figure 14(b), *canneal*'s average IPC is reduced from 1.03 to 0.70

Fig. 15. Performance and fairness of 6 two-program mixtures.

(33% degradation), and *bodytrack*'s average IPC rises from 0.14 to 0.35 ($2.5\times$ improvement). With f-SLFQ in Figure 14(c), *cannal*'s average IPC is reduced from 1.03 to 0.81 (18.4% degradation), but *bodytrack*'s average IPC rises from 0.14 to 0.46 ($3.3\times$ improvement). In Figure 14(d), f-SLFQ+IHT consistently improves the average IPCs for both *cannal* and *bodytrack* to 0.87 and 0.49, respectively. Finally, Figure 14(e) shows the average IPC of f-WFQ+IHT to be 0.92 and 0.51 for *cannal* and *bodytrack*, correspondingly.

Figure 15(a) summarizes the weighted speedup (denoted Θ , see Equation (4)) results. For the 6 two-program mixtures, weighted speedup is averagely improved from $1\times$ to $1.6\times$. More significantly, we observe that low injection rate programs are much improved by STC and f-SLFQ over p-RR and further by f-SLFQ+IHT over STC and f-SLFQ. We also quantify the fairness (denoted Ω , see Equation (5)) results in Figure 15(b). Clearly, we can see that STC and f-SLFQ greatly reduce the average standard deviation for all combinations of p-RR from 0.34 to 0.19 and 0.17, respectively, and f-SLFQ+IHT further reduces the deviation to 0.02, which is very close to 0.01 of f-WFQ+IHT.

For multiple co-executing applications, both STC and SLFQ exploit application dynamism. Because application ranking prioritizes packets from low-injection applications (more critical) over high-injection applications (less critical) while packet batching is needed to avoid starvation for packets from high-injection applications, STC achieves similar effects as f-SLFQ in average delay increment/decrement among the benchmark applications, as shown in Figure 13. However, since its underlying scheduling policy for the same batch with the same rank is still RR, STC is less fair and less efficient than f-SLFQ, as observable in Figures 14 and 15. Furthermore, STC is much less effective than f-SLFQ+IHT because STC's targeting of multiprogrammed applications does not take causality of cache coherence-related messages into account. In this regard, consider a scenario when a low-injection thread modifies a shared data block in the last level L2 cache and sends invalidations to multiple high-injection threads. Since STC separately determines the priorities for invalidates and corresponding replies, acknowledgments from the high-injection threads are given lower priorities, which may in turn delay the service of the original low-injection thread. The benefits of low-injection applications are thus hampered, leading to STC's lower efficiency than f-SLFQ+IHT in handling multithreaded applications.

Four co-running applications: We executed nine mixtures of four programs, with each program occupying 16 cores. We form three kinds of mixtures: 3-low 1-high, 2-low 2-high, and 1-low 3-high injection rate programs. Each kind has three mixtures.

Figure 16 shows IPCs of one mixture running two high injection rate programs, *cannal* and *freqmine*, and two low injection rate programs, *bodytrack* and *vips*. Compared to p-RR in Figure 16(a), STC in Figure 16(b) and f-SLFQ in Figure 16(c) increase the IPCs of both low injection rate programs at the expense of both high injection rate

Fig. 16. Per-node IPC for jointly executing a mixture of 4 programs: *bodytrack*, *canneal*, *vips* and *freqmine* (Mixture-4).

Fig. 17. Performance and fairness of 9 four-program mixtures.

programs. With p-RR, the average IPCs of *canneal*, *freqmine* and *bodytrack*, *vips* are 0.94, 0.87 and 0.15, 0.14, respectively. With STC and f-SLFQ, the corresponding values become 0.64, 0.62, 0.32, 0.33 and 0.77, 0.76, 0.39, 0.38, respectively. f-SLFQ+IHT in Figure 16(d) further improves the respective values to 0.82, 0.78 and 0.44, 0.43, approaching the best values of 0.87, 0.83, 0.47, and 0.47 with f-WFQ+IHT in Figure 16(e).

Figure 17(a) gives weighted speedup of the nine mixtures of four applications each. The average speedup of p-RR, STC, f-SLFQ, f-SLFQ+IHT, and f-WFQ+IHT is 1.84, 2.31, 2.46, 2.67, and 2.88, respectively. The average improvements of STC and f-SLFQ over p-RR are 26% and 34%, respectively, and of f-SLFQ+IHT over p-RR, 45%. The difference between f-SLFQ+IHT and f-WFQ+IHT is only 0.21. Consistently, we see that low injection rate programs benefit most from our approach. Their relative IPCs become comparable to those of high injection rate programs. Figure 17(b) quantifies performance fairness. Clearly, we can see that STC and f-SLFQ greatly reduce the average fairness for all combinations of p-RR from 0.30 to 0.14 and 0.13, respectively, and, in turn, f-SLFQ+IHT reduces the fairness to 0.02, which is very close to 0.01 of f-WFQ+IHT.

Eight co-running applications: We further executed eight mixtures of eight programs, with each program symmetrically running on eight cores. Each mixture contains 4-high and 4-low injection rate programs. The weighted speedup results for the eight mixtures are shown in Figure 18(a). The average speedup of p-RR, STC, f-SLFQ, f-SLFQ+IHT, and f-WFQ+IHT is 1.92, 2.35 (22% increase), 2.48 (29% increase), 2.60 (35% of increase), and 2.74 (43% of increase), respectively. As expected, we see again that (i) all mixtures gain improvement in weighted speedup and (ii) low injection rate programs contribute most to the improvements. As shown in Figure 18(b), the performance fairness of p-RR, STC, f-SLFQ, f-SLFQ+IHT, and f-WFQ+IHT equals 0.12, 0.065, 0.05, 0.01, and 0.007, respectively.

Fig. 18. Performance and fairness of 8 eight-program mixtures.

Fig. 19. Accuracy of the rate characterization and prediction.

6.4. Evaluation of Rate Profiling

We evaluated the shifting window-based rate characterization and the state-based rate prediction mechanism. Figure 19(a) illustrates the difference between the actual resource access rate and the prediction value (both in terms of L1 cache miss rate) for the *canneal* and *bodytrack* programs. The figure manifests that, for both programs, the rate characterization mechanism achieves high fidelity in prediction within each program phase. Figure 19(b) further shows the prediction error and the standard deviation (using *canneal* as an example) to illustrate the dispersion of the predicted values. With standard deviation equaling only 2.5% of the mean, the rate profiling mechanism precisely characterizes application dynamism. Finally, Figure 19(c) summarizes the standard deviations of all the 12 PARSEC programs. Because these deviations are insensitive to programs' injection rates, we confirm that our rate profiling scheme is valid and very effective.

7. RELATED WORK

QoS-oriented resource management: According to *targeted degree of strength*, QoS techniques for managing shared resources in multi-core systems may be roughly classified as *firm*, *strong*, and *fair* ones.

Firm QoS techniques aim to provide guaranteed performance isolation by reserving or partitioning resources in spatial and/or time domains either statically or dynamically for exclusive use. In Nesbit et al. [2008], the multicore system is partitioned as Virtual Private Machines (VPMs), each of which includes partitioned cores, a slice of cache space, and a portion of main memory. Each application has a minimum-maximum number of VPMs and its own time slots for execution. In Lee et al. [2008], the guaranteed-service NoC hosts Globally Synchronized Time Frames (GSF) to accept a limited quota of traffic at each frame. In Xie et al. [2014], a dynamic bank partitioning

mechanism is proposed to dynamically partition memory banks according to applications' requirements. To reduce interapplication interference in the memory system, the data of applications that are likely to severely interfere with each other are mapped to different memory channels [Muralidhara et al. 2011]. In Jimenez et al. [2015], an online bandwidth shifting mechanism is developed to dynamically assign bandwidth to applications according to their prefetch efficiency. To jointly manage cache, NoC, and off-chip memory resources simultaneously, a class-of-service-based unified framework advocates the necessity of coordinated management, and a QoS policy in favor of high-priority applications is used to statically allocate resources at the beginning of application runtime to guarantee their performance [Li et al. 2011].

Strong QoS schemes intend to provide a certain level of guaranteed performance isolation by aggregately sharing resources, with interference typically controlled by admission or prioritization or analytically bounded. To enhance resource utilization, resource reallocation is often needed to exploit an application's resource demand fluctuation. In Guo et al. [2007], the excessive resources can be reallocated by other jobs to optimize system throughput, while an admission control is incorporated to accept jobs only when their QoS targets can be met. Based on the architecture of Li et al. [2011], a dynamic QoS co-management technique is developed in Li et al. [2012] that monitors the resource usage of applications at runtime and then conditionally steals resources from high-priority applications for lower-priority ones. The goal is to maintain the targeted level of performance for high-priority applications while improving the performance of lower-priority applications. In METE [Sharifi et al. 2011], multi-level resources (core, cache, off-chip memory) are holistically managed in a coordinated fashion through a multilevel feedback control system according to a potential performance target set by each application. The necessity of coordinated management of multiple interacting resources in CMPs was first studied in Bitirgen et al. [2008], where a machine learning-based global resource allocation framework is established to enforce system-level performance objectives. In Ebrahimi et al. [2010], a hardware-based source throttling solution enables system software-specified fairness goals to be achieved in the entire memory system. If runtime unfairness, which is measured as the ratio of the largest slowdown to the smallest slowdown of co-running programs, is above a threshold set by the system software, the source cores causing unfairness are throttled, limiting the number and the frequency of requests being injected into the system. In Grot et al. [2009], a Preemptive Virtual Clock (PVC) mechanism is proposed to offer *fairness guarantees* without requiring per-flow buffering (like WFQ, Virtual Clock [Zhang 1995]) in routers or large queues at source nodes (like GSF). It tracks each application's bandwidth consumption over a time interval and prioritizes packets based on the consumed bandwidth. Moreover, preempting lower-priority messages is introduced to avoid priority inversion. However, PVC requires a dedicated ACK/NACK network and a window of outstanding transactions at each node to support the preemption. To address the scalability in performance isolation provisioning to thousand-core chips, the Kilo-NoC [Grot et al. 2011] proposes a heterogeneous low-diameter topology network with QoS support using per-flow virtual channel only in portions of the die so as to reduce QoS overheads. For heterogeneous multicore systems including Hardware Accelerators (HWAs), the Deadline-Aware Memory Scheduler for Heterogeneous Systems (DASH) [Usui et al. 2016] is developed to achieve the goal of meeting HWAs' deadlines while providing high CPU performance. Intertask interference due to aggregate resource sharing can be analytically bounded to offer predictable guarantees. For example, Kim et al. combined request-driven and job-driven approaches to provide a tight upper bound on the worst-case memory interference delay in a Commercial Off-the-Shelf (COTS)-based real-time multicore system [Kim et al. 2014].

Fair QoS approaches seek to provide impartial performance slowdowns for multiple threads or applications of equal importance by dynamically sharing resources. The concept of fair caching is proposed in Kim et al. [2004] to ensure the uniform impact of cache sharing on execution times to all co-scheduled threads. In managing the shared DRAM system, a Stall-Time Fair Memory (STFM) scheduler [Mutlu and Moscibroda 2007] is developed to offer performance fairness to all threads without compromising system performance by equalizing the runtime-estimated slowdowns due to inter-thread interference. Later, the same authors improved upon STFM via request batch scheduling in awareness of intrathread bank-level parallelism by grouping outstanding DRAM requests from a given thread as the basic unit of scheduling [Mutlu and Moscibroda 2008]. To accelerate network-sensitive applications and improve fairness, application ranking is proposed in Das et al. [2009] to rank applications using their stall-time criticality at regular intervals and thus to prioritize their packets. Additionally, packet batching is used to avoid starvation for low-priority packets. In Das et al. [2010], the same authors proposed a complementary packet slack-aware scheme which evaluates application's sensitivity to network delay and prioritizes delay-sensitive applications against insensitive ones by exploiting contention packets' available slack to improve overall system performance and fairness. In Wang and Martínez [2015], the CMP architecture functions as a "market" where each core bids for "priced" shared resources with its "budget." By allocating resources proportionally to the bids, the system avoids unfairness and treats each core in an unbiased manner.

Targeting performance fairness provisioning to multiple applications in a queuing NoC-based CMP, STC [Das et al. 2009] is the closest related work. Both techniques exploit application dynamism to achieve performance efficiency and fairness. While STC uses traffic priority levels indicating performance criticality to realize priority-based resource allocation, we use traffic rate to enable proportionality-based resource allocation. STC does not take traffic causality into consideration when ranking applications. This means that traffic initiated from the same application but generated from other nodes (e.g., reply traffic from a remote L2 or memory node) may be ranked and thus serviced differently. Our proposal suggests the importance of rate inheritance through memory hierarchy and cache coherency for consistent service. The third difference is that STC relies on a global centralized controller to perform online application ranking, which may potentially become a single point of failure and is thus not scalable. In contrast, we aim for a distributed solution by which each core profiles its own rate characteristics to be distributed to the network via request/reply packets, and the network utilizes such information to make proportional resource allocation.

Slowdown estimation-based QoS in multicore systems: To enable effective mechanisms that can provide predictable performance and fairness, a few recent works have developed slowdown estimation models. In Subramanian et al. [2013], a simple Memory Interference-induced Slowdown Estimation (MISE) model is presented to estimate application slowdowns caused by memory interference in a multi-programmed environment. The slowdown of an application is estimated as the ratio of its un-interfered and interfered request service rates. Then, new main memory request scheduling mechanisms exploiting MISE, MISE-QoS, and MISE-Fair are developed to provide soft QoS guarantees to one or more applications without compromising system performance and to minimize maximum slowdown to improve overall system fairness, respectively. Very recently, an accurate Application Slowdown Model (ASM) was proposed in Subramanian et al. [2015] by estimating the shared cache access rate of the application running alone on the system. A few use cases of the accurate ASM—namely, slowdown-aware cache partitioning, slowdown-aware memory bandwidth partitioning, and a soft slowdown guarantee scheme—are then presented to improve fairness and performance and provide slowdown guarantees over state-of-the-art cache partitioning

and memory scheduling schemes. Both works use rate-based slowdown proportional memory bandwidth allocation, which shares the same insight as our rate-proportional channel bandwidth allocation. Furthermore, a first NoC Application Slowdown (NAS) model was proposed in Xiang et al. [2016] to dynamically estimate per-application slowdown due to interference-induced network delay. Based on NAS, Fairness-Aware Source Throttling (FAST) is developed as an example QoS mechanism to control the network injection rates of different applications, thereby improving system utilization and reducing unfairness.

Other factors affecting QoS in multicore systems: In addition to resource allocation, we review a few representative works concerning influences of application-to-core mapping, memory controller placement, and network congestion control on multicore QoS.

Das et al. studied application-to-core mapping policies to reduce memory system interference in multicore processors [Das et al. 2013]. Based on application differentiation, lessons learned from this study include (i) Separation: Map network-latency-sensitive applications to separate node clusters from network bandwidth-intensive applications so that the former makes fast progress without heavy interference from the latter; (ii) Locality: Map those applications that benefit more from being closer to the memory controllers close to these resources. In facing the limited pin bandwidth for integrating memory controllers on many-core chips, Abts et al. investigated the impact of memory controller placement and explored the design space to demonstrate potential improvements in performance and predictability [Abts et al. 2009].

To achieve sustainable performance and support QoS in multicores, network congestion must be controlled and network saturation avoided. To improve load balance in adapting routing, a light-weight Regional Congestion Awareness (RCA) technique is proposed for NoCs to enhance performance through improved global network load balance [Gratz et al. 2008]. In contrast to network-centric congestion control, source throttling is another approach in which nodes injecting excessive traffic are temporarily throttled to help the network recover from congestion. In this regard, a self-tuned congestion control technique using a hill-climbing algorithm is developed to avoid network saturation. It uses global knowledge of buffer occupancy to estimate network congestion and control packet injection when the number of full buffers exceeds a dynamic threshold, which is automatically tuned according to delivered network throughput [Thottethodi et al. 2001]. While network-load information is used to guide source throttling in Thottethodi et al. [2001], application-level knowledge is also considered necessary to guide effective source throttling, as demonstrated in Moscibroda and Mutlu [2009]. An application-aware scalable throttling mechanism is presented in Nychis et al. [2012] to mitigate the effect of congestion in bufferless NoCs [Nychis et al. 2010]. It is also shown in Ausavarungnirun et al. [2011] that system performance and fairness in bufferless NoCs can be improved by Adaptive Cluster Throttling (batching applications into clusters and throttling different clusters alternately). Combining both network load and application knowledge in network intensity, a Heterogenous Adaptive Throttling (HAT) technique is proposed for the first time in Chang et al. [2012] to achieve best system performance and fairness in NoC-based multicore systems.

Fair queuing for QoS and fairness: SLFQ is adapted from SCFQ [Golestani 1994] with limited buffering in order to achieve cost-effective hardware implementation. By itself, SCFQ is an approximation of WFQ. WFQ [Demers et al. 1989], by another name, Packet-wise Generalized Processor Sharing (PGPS) [Parekh and Gallager 1993], is an approximation of GPS. In a GPS node, link bandwidth is allocated to multiple backlogged flows in proportion to their allocated weights. GPS is, however, defined in an idealized fluid system in which the traffic is infinitely divisible and multiple traffic flows can be served simultaneously. Despite being an ideal theoretic model, GPS has

two desirable properties. One is to provide performance guarantees and the other fair bandwidth allocation for best-effort services. To make GPS implementable in packet systems where one packet from one queue is served at any given time, WFQ (PGPS) gives a packet approximation of GPS by serving packets in the increasing order of their finish times in the fluid GPS system. WFQ is, however, complex to implement since it depends on virtual time to emulate fluid GPS and has to keep track of the number of active flows at any time. SCFQ introduces an approximation algorithm by estimating the system's virtual time at any moment from the virtual finish time of the packet currently being serviced. As such, it greatly reduces the implementation complexity of WFQ while maintaining accuracy. WFQ and SCFQ can promise maximum packet delay through ensuring minimum bandwidth allocation per flow. To support multiple service classes such as guaranteed real-time, rate-adaptive best-effort, and controlled link-sharing services, GPS is extended to hierarchical GPS (H-GPS) to address resource sharing at different levels [Bennett and Zhang 1997]. H-GPS can be viewed as a hierarchical integration of one-level GPS servers. Similarly, to make H-GPS implementable in packet systems, Hierarchical Packet Fair Queueing (H-PFQ) algorithms are designed for approximation. Instead of explicitly guaranteeing a specific QoS measure such as maximum delay, the notion of service curve guarantee is developed and used to design a Service Curve-based Earliest Deadline (SCED) first scheduling policy to provide performance guarantees to connections in packet-switched networks with larger flexibility and schedulability [Sariowan et al. 1995]. SCED serves packets in the increasing order of their deadlines, calculated by flow burstiness constraint and per-flow guaranteed service curve. To ensure a memory scheduler with fairness and performance isolation to applications, fair queueing memory systems are designed using Network Fair-Queueing (NFQ) algorithms [Nesbit et al. 2006]. Later, a simple yet efficient DRAM management scheme was proposed in Rafique et al. [2007] which employs start-time fair queueing to improve the fairness of the NFQ scheme using virtual finish-time fair queueing.

8. CONCLUSION

We presented a new flow-centric approach to address the performance fairness problem in NoC-based CMPs. Our rationale is that efficient and fair resource sharing in a best-effort network must continuously, proportionally, and consistently respect a program's runtime resource access demand at finer time granularity. To this end, we introduced the notion of *aggregate flow*, on which we build up three coherent mechanisms. The rate profiling characterizes packet generation rates from L1/L2 caches, which are inserted into the packets of request messages. The rate inheritance allows causal reply packets to inherit the rates of corresponding initial request or next-level request messages. Finally, the routers implement SLFQ scheduling adapted from SCFQ [Golestani 1994] using each flow's runtime rate information for proportional channel bandwidth allocation without per-flow queueing.

We realized and evaluated our mechanisms in GEM5 with the PARSEC benchmarks running on a 64-node CMP. Compared to RR packet scheduling (p-RR) and STC-based application-aware packet prioritization [Das et al. 2009], our proposed approach (f-SCFQ+IHT) improves the system IPCs of all benchmark programs running stand-alone by 14.6% and 12.3% on average, respectively. When co-running two-, four-, and eight-program mixtures, we observe that (i) the respective average weighted speedup is enhanced by 47%, 45%, and 35% against p-RR and by 19%, 16%, and 11% against STC; (ii) the respective average IPC fairness (standard deviation of IPC slowdowns) is improved by $17\times$, $15\times$, and $12\times$ against p-RR and by $10\times$, $7\times$, and $6.5\times$ against STC. In particular, the respective IPC fairness is minimized to 0.02, 0.02, and 0.007, thus approaching to ideal fairness (zero standard deviation).

REFERENCES

- Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. 2009. Achieving predictable performance through better memory controller placement in many-core CMPs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 451–461.
- Niket Agarwal, Tushar Krishna, Li Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 33–42.
- Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Chris Fallin, and Onur Mutlu. 2011. Adaptive cluster throttling: Improving high-load performance in bufferless on-chip networks. *CMU SAFARI Technical Report No. 2011-006* (2011).
- Jon C. R. Bennett and Hui Zhang. 1997. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking* 5, 5 (1997), 675–689.
- Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.
- Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
- Ramazan Bitirgen, Engin Ipek, and José F. Martínez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of International Symposium on Microarchitecture (MICRO)*. 318–329.
- Kevin Kai-Wei Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu. 2012. HAT: Heterogeneous adaptive throttling for on-chip networks. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 9–18.
- Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 107–118.
- Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 280–291.
- Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2010. Aergia: Exploiting packet latency slack in on-chip networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 106–116.
- Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review* 19, 4 (1989), 1–12.
- Benoît Dupont de Dinechin, Yves Durand, Duco van Amstel, and Alexandre Ghiti. 2014. Guaranteed services of the NoC of a manycore processor. In *Proceedings of the International Workshop on Network-on-Chip Architectures (NoCArc)*. 0–5.
- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 335–346.
- Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3 (2008), 42–53.
- S. Jamaloddin Golestani. 1994. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of the International Conference on Computer Communications (INFOCOM)*. 636–646.
- Kees Goossens, John Dielissen, and Andrei Radulescu. 2005. Æthereal network on chip: Concepts, architectures, and implementations. *Design Test of Computers* 22, 5 (2005), 414–421.
- Paul Gratz, Boris Grot, and Stephen W. Keckler. 2008. Regional congestion awareness for load balance in networks-on-chip. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 203–214.
- Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. 2011. Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 401–412.
- Boris Grot, Stephen W. Keckler, and Onur Mutlu. 2009. Preemptive virtual clock: A flexible, efficient, and cost-effective QoS scheme for networks-on-chip. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 268–279.

- Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. 2007. A framework for providing quality of service in chip multi-processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 343–355.
- John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann.
- Victor Jimenez, Alper Buyuktosunoglu, Pradip Bose, Francis P. O'Connell, Francisco Cazorla, and Mateo Valero. 2015. Increasing multicore system efficiency through intelligent bandwidth shifting. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 39–50.
- Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 145–154.
- S. Kim, D. Chandra, and Y. Solihin. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 111–122.
- Jae W. Lee, Man Cheuk Ng, and Krste Asanović. 2008. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 89–100.
- Bin Li, Li-Shiuan Peh, Li Zhao, and Ravi Iyer. 2012. Dynamic QoS management for chip multiprocessors. *ACM Transation on Architecture and Code Optimization (TACO)* 9, 3 (2012), 17:1–17:29.
- Bin Li, Li Zhao, Ravi Iyer, Li Shiuan Peh, Michael Leddige, Michael Espig, Seung Eun Lee, and Donald Newell. 2011. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *Journal of Parallel and Distributed Computing* 71, 5 (2011), 700–713.
- Zhonghai Lu and Yi Wang. 2012. Dynamic flow regulation for IP integration on network-on-chip. In *Proceedings of the International Symposium on Networks on Chip (NoCS)*. 115–123.
- Paul Marchal, Diederik Verkest, Adelina Shickova, Francky Catthoor, Frédéric Robert, and Anthony Leroy. 2005. Spatial division multiplexing: A novel approach for guaranteed throughput on NoCs. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES)*. 81–86.
- Thomas Moscibroda and Onur Mutlu. 2009. A case for bufferless routing in on-chip networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2009), 196–207.
- Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. 2011. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 374–385.
- Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 3–14.
- Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 146–158.
- Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 63–74.
- Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. 2006. Fair queuing memory systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 208–219.
- Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith. 2008. Multicore resource management. *IEEE Micro* 28, 3 (2008), 6–16.
- George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu. 2010. Next generation on-chip networks: What kind of congestion control do we need? In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks*. Article No. 12.
- George P. Nychis, Chris Fallin, Thomas Moscibroda, Onur Mutlu, and Srinivasan Seshan. 2012. On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 407–418.
- Jin Ouyang and Yuan Xie. 2010. LOFT: A high performance network-on-chip providing quality-of-service support. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 409–420.
- Abhay K. Parekh and Robert G. Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *Transactions on Networking* 1, 3 (1993), 344–357.

- Sunghyun Park, Tushar Krishna, Chia-Hsin Chen, Bhavya Daya, Anantha Chandrakasan, and Li-Shiuan Peh. 2012. Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI. In *Proceedings of the Design Automation Conference (DAC)*. 398–405.
- Li-Shiuan Peh and William J. Dally. 2001. A delay model and speculative architecture for pipelined routers. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 255–266.
- Nauman Rafique, Won Taek Lim, and Mithuna Thottethodi. 2007. Effective management of DRAM bandwidth in multicore processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 245–255.
- Jennifer L. Rexford, Albert G. Greenberg, and Flavio G. Bonomi. 1996. Hardware-efficient fair queueing architectures for high-speed networks. In *Proceedings of the International Conference on Computer Communications (INFOCOM)*. 638–646.
- Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. 1995. Scheduling for quality of service guarantees via service curves. In *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)*. 512–520.
- Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. 2011. METE: Meeting end-to-end QoS in multicores through system-wide resource management. *ACM SIGMETRICS Performance Evaluation Review* 39, 1 (2011), 13.
- Allan Snaveley and Dean M. Tullsen. 2000. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 234–244.
- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 62–75.
- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. 2013. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 639–650.
- Mithuna Thottethodi, Alvin R. Lebeck, and Shubhendu S. Mukherjee. 2001. Self-tuned congestion control for multiprocessor networks. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 107–118.
- Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 65:1–65:28.
- Xiaodong Wang and José F. Martínez. 2015. XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 113–125.
- Xiyue Xiang, Saugata Ghose, Onur Mutlu, and Nian-Feng Tzeng. 2016. A model for estimating application slowdowns in NoCs and its use for improving network fairness and performance. In *Proceedings of the International Conference on Computer Design (ICCD)*.
- Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. 2014. Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 344–355.
- Hui Zhang. 1995. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE* 83, 10 (1995), 1374–1396.

Received May 2016; revised October 2016; accepted October 2016