Application-Aware Prioritization Mechanisms for On-Chip Networks

Reetuparna Das§ Onur Mutlu† Thomas Moscibroda‡ Chita R. Das§
§Pennsylvania State University †Carnegie Mellon University ‡Microsoft Research
{rdas,das}@cse.psu.edu onur@cmu.edu moscitho@microsoft.com

Abstract

Network-on-Chips (NoCs) are likely to become a critical shared resource in future many-core processors. The challenge is to develop policies and mechanisms that enable multiple applications to efficiently and fairly share the network, to improve system performance. Existing local packet scheduling policies in the routers fail to fully achieve this goal, because they treat every packet equally, regardless of which application issued the packet.

This paper proposes prioritization policies and architectural extensions to NoC routers that improve the overall application-level throughput, while ensuring fairness in the network. Our prioritization policies are *application-aware*, distinguishing applications based on the *stall-time criticality* of their packets. The idea is to divide processor execution time into phases, rank applications within a phase based on stall-time criticality, and have all routers in the network prioritize packets based on their applications' ranks. Our scheme also includes techniques that ensure starvation freedom and enable the enforcement of system-level application priorities.

We evaluate the proposed prioritization policies on a 64-core CMP with an 8x8 mesh NoC, using a suite of 35 diverse applications. For a representative set of case studies, our proposed policy increases average *system throughput* by 25.6% over age-based arbitration and 18.4% over round-robin arbitration. Averaged over 96 randomly-generated multiprogrammed workload mixes, the proposed policy improves system throughput by 9.1% over the best existing prioritization policy, while also reducing application-level unfairness.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiprocessors; Interconnection architectures; C.1.4 [**Parallel Architectures**]: Distributed architectures

General Terms

Design, Algorithms, Performance

Keywords

On-chip networks, multi-core, arbitration, prioritization, memory systems, packet scheduling

1. Introduction

Packet-based Network-on-Chips (NoCs) are envisioned to be the solution for connecting tens to hundreds of components in a future many-core processor executing hundreds of tasks. Such an NoC is a critical resource, likely to be be shared by *diverse applications* running *concurrently* on a many-core processor. Thus, effective utilization of this shared interconnect is essential for improving *overall system performance* and is one of the important challenges in many-core system design.

MICRO'09, December 12–16, 2009, New York, NY, USA.

Traditionally, NoCs have been designed [22, 14, 16, 13, 17] to optimize *network-level performance metrics*, such as saturation throughput or average packet latency. These metrics capture inherent performance characteristics of the network itself, but they are not directly related to the performance metrics observable at the application-level or system-level. There are various reasons as to why optimizing for network parameters may not be adequate to improve system performance. First, average packet latency may not be indicative of network-related stall-time at the processing core (i.e. number of cycles during which the processor cannot commit instructions waiting for an in-transit network packet [19]). This is because much of the packets' service latencies might be hidden due to memory-level parallelism [11]. Second, the network's saturation throughput might not be critical given the self-throttling nature of CMPs: a core cannot inject new requests into the network once it fills up all of its miss request buffers). Third, average network throughput may not accurately reflect system performance because the network throughput demands of different applications can be vastly different.

To exemplify these points, Figure 1 shows performance in terms of different metrics for a 64-node multicore processor, with 32 copies of two applications, `leslie` and `omnetpp`, running together. The figure contrasts three network-level performance metrics (throughput demanded/offered and average packet latency) with two application-level performance metrics (execution-time slowdown and network related stall-time slowdown each application experiences compared to when it is run alone on the same system; these metrics correlate with system-level throughput as shown in [27, 8]). Even though the network-level performance metrics of the two applications are very similar, the slowdown each application experiences due to sharing the NoC is vastly different: `leslie` slows down by 3.55X, whereas `omnetpp` slows down by 7.6X. Hence, designing the network to optimize network-level metrics does not necessarily provide the best application and system-level throughput.

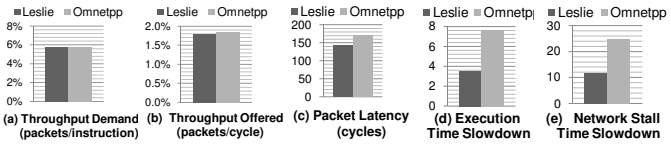

Figure 1: Different Metrics for Network Performance Analysis

A key component of a router that can influence application-level performance is the arbitration/scheduling unit. Commonly-used arbitration policies in NoCs are *application-oblivious* and *local*. Such application-oblivious local policies lead to reduced overall system performance because they 1) fail to harness application properties to improve system throughput, 2) can lead to un-coordinated and contradictory decision-making by individual routers. First, once a packet is injected into the network, scheduling decisions taken by routers do not consider which applications issued the packets. As a result, packets from applications that are particularly sensitive to network packet latency are treated with the same priority as other, less-critical packets, which can substantially decrease application-level performance of the sensitive application. Second, each router makes packet scheduling decisions using information local to that router. For example, a router can schedule packets from different vir-

tual channels in a round-robin fashion over cycles, schedule a packet from the least recently used virtual channel, or schedule the locally-oldest packet first. As a result, a packet may be prioritized in one router only to be delayed by the next router. Consequently, one application might be prioritized in one router and delayed in a second one, whereas another application can be prioritized by the second router and delayed in the first, which leads to an overall degradation in system throughput.

To mitigate the above disadvantages of existing approaches, we propose a new substrate to enable *application-aware prioritization* in an NoC by *coordinating* the arbitration/scheduling decisions made by different routers. Our approach is based on the concept of *stall-time criticality* (STC). A network packet has high stall-time criticality if an increase in its network latency results in a high increase of application-level stall-time. In our approach, each router schedules packets based on which application the packets belong to. Specifically, the STC-policy combines two mechanisms: *application ranking* and *packet batching*. Applications are ranked based on the stall-time criticality of their network packets. Each router prioritizes packets according to this rank: packets of an application where network performance is critical (higher-ranked applications) are prioritized over packets from lower-ranked applications. Prioritization is enforced across the network in a coordinated and consistent fashion because all routers use the same ranking. The packet batching mechanism ensures starvation freedom: periodically, a limited number of network packets are grouped into batches. Each router prioritizes packets belonging to older batches, thereby ensuring that no application starves due to interference from other, higher-ranked (or potentially aggressive) applications.

Experimental evaluations show that our proposal effectively increases overall system throughput and application-level fairness in the NoC. In addition, our application-level prioritization substrate can be seamlessly extended to enforce application priorities assigned by the operating system. In comparison to existing application oblivious routing policies that cannot easily enforce system-level priorities, this is an added advantage. Thus, the main **contributions** of this paper are the following:

• We observe that common network-level metrics can differ substantially from application-level performance metrics and identify the *stall-time criticality of network packets* as a key concept that affects overall system performance.

• We show that local, application-oblivious arbitration policies in NoC routers can degrade application-level system throughput and fairness. We further show that global policies based on network parameters lead to poor application performance, and can lead to unfair performance between applications with different memory access patterns.

• We propose novel prioritization mechanisms to improve application-level system throughput without hurting application-level fairness in NoCs. Our key idea is to dynamically identify applications that benefit the most from prioritization in the network due to higher stall-time criticality of their packets and coordinate the routers to prioritize these applications' packets.

• We qualitatively and quantitatively compare our mechanisms to previous local and global arbitration policies, including age-based prioritization and globally synchronized frames [17]. We show that our proposal provides the highest *overall system throughput* (9.1% higher than the best existing policy over 96 diverse workloads) as well as the best application-level fairness.

2. Background

In this section, we provide a brief background on NoC architectures along with current packet scheduling and arbitration policies. For an in-depth introduction to NoCs, we refer the reader to [5].

A Typical Router: A generic NoC router architecture is illustrated in Figure 2(a). The router has P input and P output channels/ports; typically $P = 5$ for a 2D mesh, one from each direction and one from the network interface (NI). The Routing Computation unit, RC, is responsible for determining the next router and the virtual channel within the next router for each packet. The Virtual channel Arbitration unit (VA) arbitrates amongst all packets requesting access to the same VCs and decides on winners. The Switch Arbitration unit (SA) arbitrates amongst all VCs requesting access to the crossbar and grants permission to the winning packets/flits. The winners are then able to traverse the crossbar and are placed on the output links.

A Typical Network Transaction: In a general-purpose chip multiprocessor (CMP) architecture, the NoC typically interconnects the processor nodes (a CPU core with its outermost private cache), the secondary on-chip shared cache banks, and the on-chip memory controllers (See Figure 2(b)). The processor sends request packets to cache bank nodes and receives data packets via the NoC. If the requested data is not available in the cache, the cache bank node sends request packets to the on-chip memory controller nodes via the NoC, and receives response data packets from the controller once data is fetched from off-chip DRAMs. Each packet spends *at least* 2-4 cycles at each router depending on the number of stages in the router.

Packet Arbitration Policies within an NoC Router: Each incoming packet is buffered in a virtual channel (VC) of the router, until it wins the virtual channel arbitration stage (VA) and is allocated an output VC in the next router. Following this, the packet arbitrates for the output crossbar port in the switch arbitration stage (SA). Thus, there are two arbitration stages (VA and SA) where the router must choose one packet among several packets competing for either a common 1) output VC or 2) crossbar output port.[1] Current routers use simple, local arbitration policies to decide which packet should be scheduled next (i.e., which packet wins arbitration). A typical policy, referred to as **LocalRR** in this paper, is to choose packets in different VCs in a round-robin order such that a packet from the next non-empty VC is selected every cycle. An alternative policy, referred to as **LocalAge** in this paper, chooses the oldest packet.

Why scheduling and arbitration policies impact system performance? In addition to the router pipeline stages, a packet can spend *many cycles* waiting in a router until it wins a VC slot and gets scheduled to traverse the switch. While its packets are buffered in remote routers, the application stalls waiting for its packets to return. Thus, packet scheduling or arbitration policies at routers directly impact application performance.

Why scheduling and arbitration policies impact system fairness? A scheduling or arbitration policy dictates which packet *the router chooses* and hence also which *packet is penalized*. Thus, any policy which gives higher priority to one packet over another might affect the fairness in the network. Current routers use locally fair policies that are application-agnostic. Locally fair policies need not be globally fair. For example, the LocalRR might seem a very fair policy locally, but could lead to high unfairness in asymmetric topologies.

3. Motivation

Existing NoC routing policies are implicitly built on the paradigm that every packet in the network is equally important and hence, packets from one application should not a-priori be prioritized over packets from other applications. In this paper, we challenge this paradigm by observing how different packets (even within the same application, but particularly across different applications) can have vastly different impact on application-level performance. In this section, we describe three reasons that can cause differences in *stall-time criticality (STC)* of packets. Combined, they provide the motivation for our application-aware approach to NoC packet scheduling, which is described in Section 4.

[1] Note that in wormhole switching only the head flit arbitrates for and reserves the VC; the body flits thus do not require VC allocation. However, all flits have to compete for the switch. For simplicity, our discussions will be in terms of packets instead of flits.

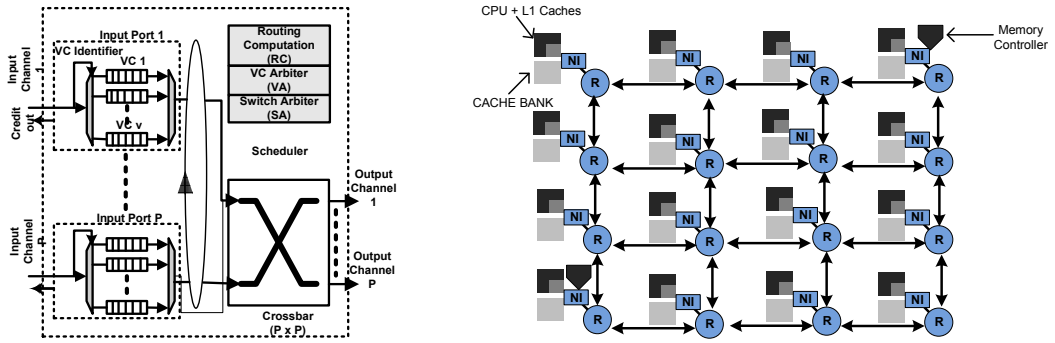

Figure 2: Generic NoC router and topology: (a) Baseline Router (b) Example CMP Layout for a 4x4 Mesh.

Different Stall-Time Criticality due to Different Memory-Level Parallelism: Modern microprocessors employ several memory latency tolerance techniques [7, 15, 31, 21] to hide the penalty of load misses. These techniques issue several memory requests in parallel in the hope of overlapping future load misses with current load misses. The notion of issuing and servicing several requests in parallel is called *Memory Level Parallelism* (MLP) [11]. The concept of MLP affects the criticality of load requests. Although there might be multiple outstanding load misses in the system, not every load miss is the *bottleneck-causing (i.e. critical)* miss [9]. Typically the oldest load is the most critical to the processor, whereas many other misses might be completely overlapped with earlier requests. Previous research [28, 25] has established that not all load misses are equally critical to the processor. *Hence, increasing the latency of these non-critical misses (i.e. packets) in the network will have less impact on application execution time, compared to increasing the latency of critical misses.*

Moreover, every application has *different degrees of MLP*. For example Figure 3(a) shows the execution timeline of an application with no MLP (e.g. `libquantum`): it sends one packet to the network after a number of instructions. In the absence of other concurrent requests, all of the packet latency is directly exposed as stall-time to the processor. Consequently, each packet is *stall-time critical* to the application execution time. Figure 3(b), on the other hand, shows an application with a high degree of MLP (e.g. `Gems`). It sends bursts of concurrent packets to the network. Much of the packet latency for this application is overlapped with other packets. Hence, the application stalls for much fewer cycles per packet (e.g. stall cycles of packet C is zero because it is delivered long before it is needed by the processor). Thus, on an average, a packet of the second application (`Gems`) is less stall-time critical than a packet of the first application (`libquantum`).

Consider a scenario when we run `Gems` and `libquantum` together on a shared multi-core system. The packet latencies for all applications will increase considerably due to contention in the network, shared caches, and main memory. However, the increase in latency is likely to have a worse impact on `libquantum` than `Gems`, because each packet of the former is more critical than that of the latter. Thus, in a shared system it makes sense to prioritize packets of applications with higher stall-time criticality to gain overall application-level system throughput.

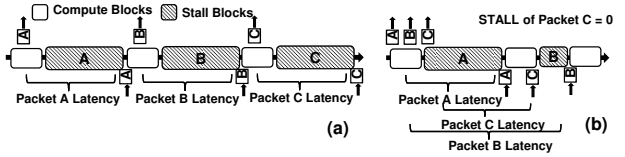

Figure 3: Execution timeline for: (a) an application with few packets, which do not overlap with each other, hence the packet latency is completely exposed to the processor as stall cycles, making each packet critical to the processor. (b) an application with packets that overlap with each other, hence some of the packet latency is hidden from the processor, accruing fewer stall cycles per packet. The packets are thus less critical.

Different Stall-Time Criticality due to Different Memory Access Latency: An application's STC is also affected by the *average latency of memory accesses*. Load misses that have high latency expose more number of stall cycles due to smaller probability of overlap with other misses. Therefore, if an application's packets frequently access remote nodes in the network or off-chip DRAM, the application is likely to have higher STC. In Figure 1, `Omnetpp` has higher stall-time criticality than `Leslie3d` because many of its memory accesses are L2 misses whereas `Leslie3d`'s are L2 hits.

Different Stall-Time Criticality due to Different Network Load: Applications can differ significantly in the amount of load they inject into the network. Applications can be "light" (low injection rate) or "heavy" (high injection rate). Hence, applications can demand different bandwidth from the network. Naturally, packets from light applications with few outstanding packets tend to be more stall-time critical because their cores can be better utilized if serviced faster. Prioritizing such *"shorter jobs"* (from the viewpoint of the network) ahead of more intense applications can increase system throughput.

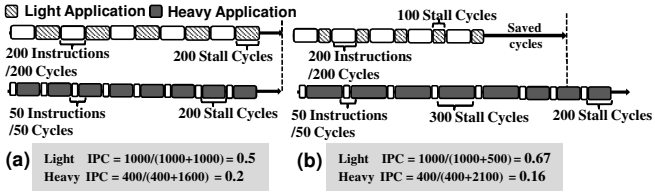

Figure 4: The left figure (a) shows execution timeline of two applications: a light application and a heavy application. The right figure (b) shows the potential savings with shortest-job-first scheduling of the two applications within the network.

Figure 4(a) illustrates this concept by showing the execution timelines of two applications. The light application has compute blocks of 200 instructions/200 cycles and stall blocks of 200 cycles. The heavy application has compute blocks of 50 instructions/50 cycles and stall blocks of 200 cycles (hence, 4x the injection rate of the first). Assume that the network prioritizes the lighter application (the striped packets in the network), and as a result each of its packets takes 100 cycles less, whereas each of the heavy application's contending packets takes equally (100 cycles) more, as shown in Figure 4(b). At the application level, this prioritization results in a significantly higher IPC throughput gain for the light application than the IPC throughput loss for the heavy application. As a result, overall system throughput improves by 18.6%/21.4% (as measured respectively by average IPC and total number of instructions completed in the a fixed number of cycles). This example shows that prioritizing light applications (i.e. "shorter jobs") within the network leads to better overall system performance by prioritizing applications that are likely to benefit more from faster network service in terms of application-level throughput (i.e. IPC).

4. Application-Aware Prioritization Substrate

4.1 Overview

Our goal is to improve application-level throughput by prioritizing packets belonging to stall-time critical applications over others in the network-on-chip. To enable this prioritization, we use two principles: *application ranking* and *packet batching*.

Application Ranking: In order to prioritize stall-time-critical applications, a central decision logic periodically forms a "ranking," i.e. *an ordering of all applications*. This ranking is determined using heuristics that capture the stall-time criticality of each application. We describe specific heuristics for maximizing overall system performance and enforcing system-level application priorities in Section 4.2. Once determined, ranks are then communicated to the routers. Routers use this ranking to determine which application's packets are prioritized at any given cycle in a router.

Packet Batching: Clearly, always prioritizing high-ranked applications can cause starvation to low-ranked applications. To prevent starvation, we propose using the concept of *batching*. Network packets are grouped into finite-size batches. A packet belonging to an older batch is always prioritized over a packet in a younger batch. As a result, reordering of packets across batches is not possible within a router. A batch also provides a convenient granularity in which the ranking of the applications is enforced. Several batching schemes are possible, and we describe the most effective ones we found in Section 4.3.

4.2 Ranking Framework

The goal of ranking is to enable the differentiation of applications within the network based on their characteristics. There are three important issues related to the ranking framework: 1) how to determine relative ranking of applications, 2) how frequently to recompute the ranking (i.e., the ranking interval size), and 3) how many ranking levels to support.

How to determine application ranking? In order to determine a good ranking, we need to estimate the stall-time-criticality (STC) of each application. We have explored a large number of heuristics to estimate the STC of applications. We describe the three best-performing heuristics below, along with their advantages and disadvantages. Note that each of the metrics used by these heuristics is computed over a time interval (called *ranking interval*) and the computed values are used to determine application rankings in the next interval.

1. Private cache misses per instruction (MPI): This heuristic ranks an application with a smaller number of *L1 cache misses per instruction* higher. The insight is twofolds: an application with a small number of L1 misses is 1) likely to issue requests relatively infrequently and hence prioritizing this application's requests allows the application's core to make fast progress without needing to wait for the network, and 2) likely to have low MLP and hence its requests are likely to be stall-time critical. This heuristic is not affected by system state or network characteristics such as which other applications are running on the system or the arbitration policy used within the routers. Therefore, it provides an easy-to-compute, accurate and stable characterization of the application's network demand.

2. Average number of outstanding L1 misses in Miss Request Queues (ReqQueue): The average number of outstanding L1 misses in the MSHRs [15] (i.e., miss request queues) of a core is indicative of 1) the overlap among memory accesses and hence the MLP of the application running on the core, and 2) the intensity of the application from the network perspective. The lower the number of outstanding misses in MSHRs for an application, the lower that application's MLP and the smaller its instantaneous demand from the network, and hence the higher its stall-time criticality. Using these observations, this heuristic ranks an application with a smaller average number of outstanding L1 misses higher.

In contrast to MPI, the ReqQueue heuristic is affected by the system state as well as the behavior of the network. It is important to be aware that this can have detrimental feedback effects: applications with higher number of L2 requests in their local queues (MSHRs) will get de-prioritized (since each request is considered less stall-time-critical). This causes these applications' queues to become longer, which in turn results in the applications to be assigned even lower ranks during the next ranking interval. In addition, we experimentally observe that the ReqQueue-based STC estimates are less stable than MPI-based estimates, fluctuating frequently across ranking intervals because the absolute number of outstanding L1 misses varies widely at any instant during the execution of an application even though L1 MPI can be relatively constant in a given phase.

3. Average stall cycles per packet (ASCP): This heuristic computes for each application the average number of cycles the application cannot commit instructions due to an outstanding packet that is being serviced by the network. Intuitively, ASCP is a direct indication of the network stall-time-criticality for the application. Therefore, the heuristic ranks the application with the largest ASCP the highest. Unfortunately, ASCP suffers from the same disadvantages as ReqQueue: the number of stall cycles a packet experiences is heavily influenced by 1) how much contention there is in the on-chip network, which depends on what other applications are concurrently running, and 2) the prioritization/arbitration policies used within the network. As such, the value of ASCP can fluctuate and is not fully indicative of an application's stall-time criticality in the network.

Supporting the above expected disadvantages, our evaluations in Section 8.5 show that the first heuristic, MPI, provides the best performance on average. Therefore, we use MPI as our *default* ranking heuristic.

Enforcing system-level priorities in the network via OS-determined application ranking: So far, we have discussed the use of the ranking substrate as a mechanism to improve system performance by ranking applications based on stall-time-criticality. We implicitly assumed that all applications have equal system-level priority. In a real system, the system software (the OS or VM monitor) may want to assign priorities (or, weights) to applications in order to convey that some applications are more/less important than others. We seamlessly modify the ranking scheme in our technique to incorporate system-assigned application weights. The system software converts the weight of applications to a ranking of applications, and conveys the rank assigned to each application to our mechanism via privileged instructions added to the ISA. This system-software-configurable ranking scheme allows system-level priorities to be enforced in the network, a functionality that does not exist in existing, application-oblivious prioritization mechanisms. Section 8.9 quantitatively evaluates and shows the effectiveness of this system-level priority enforcement.

How frequently to recompute ranking? Application behavior (and hence, network-related stall-time criticality) varies considerably over execution time [24]. Therefore, application ranking should be recomputed periodically to adapt to fine-grained phase changes within application behavior. To accomplish this, our mechanism divides the execution time into fixed time intervals (called *ranking intervals*), and re-computes the ranking at the beginning of each time interval. Intuitively, too large a ranking interval could miss changes in the phase behavior of applications and might continue to enforce a stale ranking that no longer optimizes system performance. On the other hand, too small a ranking interval could lead to high fluctuations in ranking. This can cause the rank of an application to change while many packets tagged with the application's previously-computed rank(s) are in the network, which has a negative effect on system throughput. The ranking interval can either be a static system parameter or can be computed adaptively at run-time. An adaptive

ranking interval might capture phase changes more accurately, and is likely to give superior performance than a static ranking interval, but it is more complex to implement. In this work, we use an empirically-determined static ranking interval for lower complexity.

The number of ranking priority levels R: Ideally, we would like to have as many ranking priority levels as applications (e.g., N=64 in a 64 core CMP), as this allows the network to distinguish between applications in a fine-grained manner. Specifically, each application can (if desired) be assigned a unique rank. In practice, however, the number of ranking levels may have to be constrained, especially in many-core systems consisting of large on-chip networks, because it negatively affects 1) flit and flit buffer sizes (as each flit needs to carry an encoded ranking with it) and 2) arbitration logic complexity and latency in the router (routers need to compare the encoded ranking of each flit to determine which one is of higher priority). In addition, in many cases it may not be desirable to assign different ranks to different applications, for example, if the applications have similar characteristics. For these reasons, we use fewer ranking levels than the number of applications, which imposes a partial rank order among applications by creating rank-equivalence classes.[2]

4.3 Batching Framework

In order to prevent starvation, we combine application-aware prioritization with a "batching" mechanism. Each packet is added to a batch depending on its time of arrival; and packets belonging to older batches are prioritized over packets from younger batches. Only if two packets belong to the same batch, they are prioritized based on their applications' rank order as described in the previous section. There are four important issues related to the batching substrate: 1) how to group packets into different batches, 2) how to prioritize among different batch numbers, 3) how frequently to group packets into batches (i.e., the batching interval size), and 4) how many batching levels to support.

How to group packets into batches? We consider two ways in which routers form batches in a coordinated fashion:

1. Time-Based (TB) Batching: In TB-batching, new batches are formed in a periodic, synchronous manner across all nodes in the network. All packets injected into the network (regardless of the source) during a T cycle interval (called *batching interval*) are grouped into the same batch. At the end of a batching interval, a new batch is started—all newly arriving packets are added to a new batch. Thus, packets injected in the first T cycles are assigned batch number *0*, those injected during the next T cycles are assigned batch number *1*, and so on. Assuming that the clocks of different nodes are synchronized, the same batch number is assigned by different nodes to packets injected in the same clock cycle. Hence, there is no need for global coordination or communication between routers for batch formation. *TB-batching is our default batching policy.*

2. Packet-Based (PB) Batching: Instead of starting a new batch based on a pre-determined, fixed time-interval, PB-batching starts a new batch whenever a certain number N packets have been injected into the current batch. That is, with PB-batching, the first N injected packets are assigned the batch number *0*, the next N packets the batch number *1*, and so on. Since the number of packets injected in each node is different, but batches are formed globally across *all nodes*, this scheme requires coordination among routers to keep the current batch number synchronized across all nodes. Each router communicates the number of newly injected packets (in the current batch) to a centralized decision logic, which keeps track of the total number of injected packets in this batch. Once N new packets have been injected, the centralized decision logic notifies the network routers to increment their batch IDs. As such, the hardware complexity of this mechanism is higher than TB-batching.

In both TB- and PB-batching, batch IDs start at 0 and are incremented at the end of every batching interval. After reaching the maximum supported batching level B, a *batch number wrap-around* occurs and the next batch number restarts from 0.

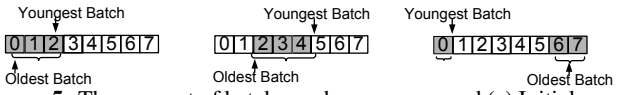

Figure 5: The concept of batch number wrap-around (a) Initial snapshot (b) Intermediate snapshot (c) Snapshot after wrap-around

How to prioritize across batches? Clearly, due to batch number wrap-around, the router cannot simply prioritize packets with lower batch numbers over others with higher batch numbers (see Figure 5). Instead, each router prioritizes packets based on *relative* order of their batch numbers. This relative order can easily be computed locally in the router using the *current batch-ID* and the packet's batch-ID. Specifically, if BID is the current injection batch-ID and PID is the packet's batch ID, a router can compute a packet's relative batch priority (RBP) as $RBP = (BID - PID)\ modulo\ B$. The packet with higher RBP is prioritized by the router. Note that priority inversion does not occur unless the number of batches present in the network exceeds the number of supported batching levels. By selecting a sufficiently large value for B, this can be easily ensured.

How frequently to group packets into batches? The size of the batching interval determines for how long some applications are prioritized over others (assuming rankings do not change). In general, the shorter the batching interval, the lower the possible starvation of lower-ranked applications but also the smaller the system-level throughput improvements due to the prioritization of higher-ranked applications. Hence, the choice of batching interval determines the trade-off between system performance and fairness, and can be tuned to accomplish system requirements. Note that the batching interval size can be made configurable by the operating system to dynamically trade-off between system performance and fairness.

How many batching levels B to support? As the number of supported batching levels B increases, the probability of priority inversion among batches decreases, but the flit size and router arbitration latency/complexity increases. Hence, B (in combination with the batching interval) should be chosen large enough to ensure that the number of incomplete batches never exceeds B. Based on our empirically determined batching interval size, we empirically set $B = 8$ in our evaluated system.

4.4 Putting It All Together: Application-Aware Prioritization Rules in a Router

Rule 1 summarizes the prioritization order applied by each router in order to enable application-aware prioritization. Each router prioritizes packets/flits in the specified order when making packet scheduling, arbitration, and buffer allocation decisions.

Rule 1 Packet/flit prioritization rules in each router

1: **Oldest Batch First:** Packets belonging to older batches are prioritized over packets belonging to younger batches (the goal is to avoid starvation).
2: **Highest Rank First:** Packets with higher rank values (belonging to higher-ranked applications) are prioritized over packets with lower rank values (the goal is to maximize system performance)
3: **Local Router Rule:** If the above two rules do not distinguish the packets, then use the local policy at the router, e.g. LocalAge or LocalRR. Our default policy is LocalAge, but LocalRR performs very similarly (see Section 8.7).

5. Comparison with Existing Policies

Local, Application-Oblivious Policies: We demonstrate the difference between application-oblivious policies and our mechanism

[2]We have determined that 8 ranking priority levels does not add significant overhead to the router complexity, while providing sufficient granularity to distinguish between applications with different characteristics, so we set R=8 (see Section 6).

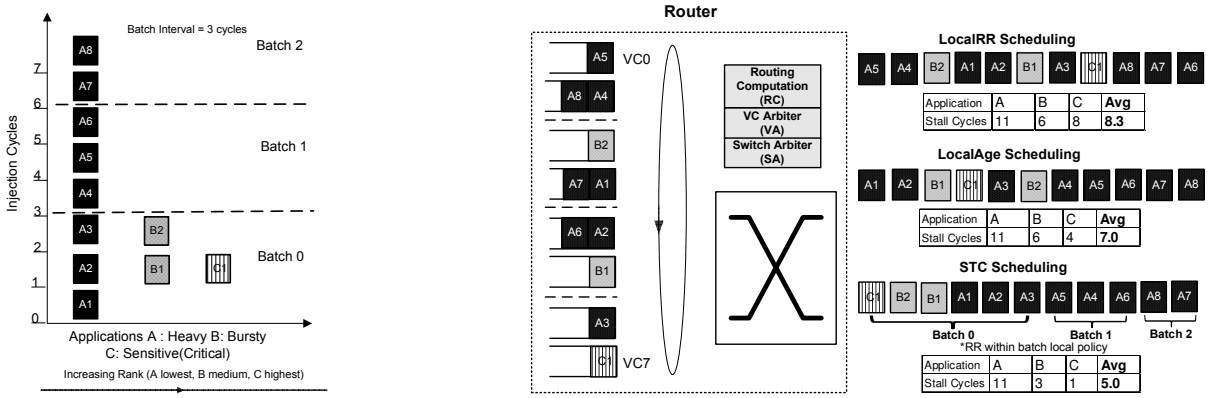

Figure 6: Example: (a) Injection order at cores (b) Scheduling order and timeline at a router for different policies

(STC) with a simplified example. This example abstracts away many details of the network and focuses on a single router, to simplify explanations and to provide insight. The example shows that our mechanism significantly reduces average stall time of applications within a single router compared to application-oblivious local policies. As our mechanism ensures that all routers prioritize applications in the same order, stall time reductions in each router are preserved across the entire NoC, thereby leading to an increase in overall system performance.

Consider an application mix of three applications: 1) *A*, a network-intensive application which injects packets at a steady rate, 2) *B*, a bursty application, and 3) *C*, a light, stall-time-critical application that infrequently injects packets. Figure 6(a) shows all the packets injected by each of these applications at their cores. To simplify the example, suppose that all these packets happen to arrive at the same router. Figure 6(b) shows this router in the network, where all injected packets are buffered. We examine the amount of time the router takes to service all packets of each application using three prioritization schemes: 1) LocalRR, 2) LocalAge, and 3) STC. The timelines in Figure 6(b) show for each scheme the service order of the packets in the router, the time it takes to service each application (i.e., each application's stall-time in this router), and the average time it takes to service all applications (i.e,. the average stall-time within this router).[3]

LocalRR services each VC in round-robin order in consecutive cycles and results in the topmost scheduling order. This order is completely application agnostic. It picks C's only packet (C1) after all other virtual channels are serviced since C1 occupies the last VC. Hence, LocalRR results in 8 stall cycles for the critical application C (11 cycles for A, 6 cycles for B). The average application stall time for LocalRR is 8.33 cycles.

LocalAge prioritizes the packets in oldest-first order (and uses LocalRR policy for equal-age packets). C1 is scheduled in the 4th cycle since it is one of the oldest packets. Thus, LocalAge improves the stall cycles of C by 2 cycles over LocalRR, thereby reducing average application stall time in this router to 7 cycles. Note, however, that LocalAge would have increased the average stall time had C1 been injected into the network much later than shown in this example. The key observation is that *the oldest packet in the network is not necessarily the most critical packet.* Even though A1 is the oldest packet in the network and it is scheduled first, it is not critical and its early scheduling does not help average stall time of applications.

Our mechanism, STC, ranks C higher than B, and B higher than A based on the MPI of each application. The router schedules the packets of applications in rank order. C1 is scheduled in the first cycle, which reduces C's stall time to 1 cycle; and B's packets are scheduled next, which reduces B's stall time to 3 cycles. Application A still stalls for the same amount of time because its last packet is serviced after 11 cycles. The average stall time reduces to 5 cycles, improving by 28% compared to the best previous local policy.

The example shows that unlike application-oblivious prioritization policies, STC can effectively prioritize critical applications that benefit the most from prioritization, and can therefore improve overall system performance.

Globally Synchronized Frames (GSF): GSF [17] provides prioritization mechanisms within the network to ensure 1) guarantees on minimum bandwidth and minimum network delay each application experiences and 2) each application achieves equal network throughput. To accomplish this, GSF employs the notion of frames, which is similar to our concept of batches. Time is quantized into equal-size frames (which are of limited number). Each source node (application) can inject an equal –yet limited– number of packets into a frame. Once an application fills its quota in a frame, it injects packets into the next frame. If an application runs out of the maximum number of supported frames to inject to (usually the case with very network intensive applications), it stalls until the oldest frame gets recycled. Each packet carries its frame number in the network. In the routers, packets belonging to the older frames are prioritized over others. Among packets belonging to the same frame, there is no prioritization based on sources, hence, there is no prioritization based on the application a packet belongs to.

Even though the concept of batching is similar to the concept of frames, our mechanism has several key differences from GSF. While the purpose of GSF is to provide bandwidth-fairness and guarantee a minimum level of bandwidth to each application, the purpose of STC is to improve system-level throughput within the network by prioritizing packets in an application-aware manner. There are four major advantages of our proposal compared to GSF:

1. *Application-aware prioritization within batches:* GSF does not employ any application-aware (or coordinated) prioritization in routers among packets belonging to the same frame. As a result, applications with non-critical packets can be prioritized over applications with critical packets. In contrast, STC employs criticality-aware ranking within each batch, enabling the fast servicing of critical packets over others. Hence, STC significantly improves system performance over GSF (see Section 8).

2. *No quota per application:* GSF imposes a quota on the number of packets each application can inject into a given frame (to ensure equal bandwidth across flows). However, this requirement can cause significant degradation in application-level throughput, especially when applications have largely different network intensity. Consider the case when a very network-intensive application is sharing the network with a bursty application. GSF divides each frame equally between them. The intensive application can inject only half of the packets within a frame, while bursty application might not in-

[3]For the purposes of this example, we make the simplifying assumption that an application stalls until all of its network packets are serviced, which is shown to be a reasonable assumption for long-latency cache misses [18].

ject *any* packets. Due to the limited number of available frames, the intensive application soon runs out of frames and stops injecting. As a result, the network becomes under-utilized even though half of every frame was not used at all. In fact, the intensive application is penalized even though the light application is not using any network bandwidth. In contrast to GSF, STC does not impose any quota on the number of packets injected into a batch by any application. Due to this, the probability of the system running out of batches is significantly smaller than in GSF. In the rare case when all batch numbers are exhausted, STC suffers from temporary priority inversion instead of stopping some applications from injecting. As a result, STC's application-level throughput is higher than GSF, especially with heterogeneous mixes of applications.

3. *No reserved buffer slots:* To provide tight latency guarantees and reduce frame completion time, GSF reserves a VC per port for the oldest frame. The reserved VC cannot be utilized by any other frame, even if the VC is empty. This reduces system throughput compared to our scheme, which does not require the reservation of any buffer slots.

4. *No global synchronization on the critical path of execution:* When the oldest frame completes in GSF, routers need to synchronize to update their local copies of the oldest frame number. This synchronization latency is on the critical path of execution in GSF due to two reasons: 1) the reserved VC goes unutilized and 2) an application that has run out of frames cannot start injection until this synchronization completes. In contrast, we do not use any global barrier network for synchronization, since we do not reserve any buffer for the oldest batch. Also, rank/batch formation updates are not on the critical path of execution in STC, because the routers are never prevented from injecting into the network and are allowed to use stale values for ranking.

The major advantage of GSF over STC is that it can provide hard bandwidth and latency guarantees to applications, at the expense of system-level throughput. STC does not provide any bandwidth or latency guarantees to applications; however, it uses batching to ensure a level of fairness. While hard guarantees are likely to be useful for real-time systems, high system throughput combined with high fairness and the ability to enforce system-level application priorities could be sufficient in general-purpose many-core systems. Hence, STC takes advantage of the relaxed constraints of general-purpose systems to improve system performance without degrading fairness.

6. Implementation

We use a central decision logic (CDL) that periodically gathers information from each node, determines a global application ranking and batch boundaries (for packet-based batching), and communicates the batch number and the node's rank back to each node. We assume the CDL hardware is located in the central node (i.e. (4,4) coordinate in an 8x8 mesh) of the NoC.

Rank Formation: Each core maintains a "rank-register" that contains the rank of the application running on this core. A new rank is computed at the end of every ranking interval. For this purpose, each processing core has additional logic and a set of hardware counters to measure the ranking metric (e.g., MPI) value that will be used by the CDL to compute rank of the nodes.[4] The counters are reset at the beginning of each ranking interval. At the end of the ranking interval, each core forms a rank-control information packet (RCIP) containing the ranking metric value(s) measured during that interval.[5] Each core then sends its RCIP packet to the CDL. Upon receiving all RCIP packets, the CDL computes the ranking among applications (see Section 4.2). In our implementation, there are $N = 64$ processors, but only $R = 8$ ranking levels. We use the standard k-means clustering algorithm [12] with four iterations ($O(4 * R * N)$ operations) to map the N processors to the R ranks. The CDL then sends the rank to each processing core using a rank-control update packet (RCUP), and upon receipt of this packet, a core updates its rank-register with the new rank. Note that the *rank formation process is not on the critical path* because the ranking interval is significantly larger than the time needed to compute a new ranking. Until a new ranking is received, each core simply uses the old ranking.

Batch Formation: Each node keeps a local copy of the Batch ID (BID) register containing the current (injection) batch number and maximum supported batch ID (MBID) register containing the maximum number of batching priority levels. Note that BID values across all nodes are the same. For Time-Based Batching, BID is simply incremented every T cycles. For Packet-Based Batching, BID is updated after every P packets that have been injected *across all nodes*. As this requires coordination among nodes, we use CDL for BID updates in this case. Each node periodically (every $U = 4000$ cycles[6]) sends a batch-control information packet (BCIP) to CDL. BCIP contains the number of packets that the router injected in the last U cycles. The CDL has a global packet counter register, which is incremented by BCIP data bits. When the CDL's counter reaches or exceeds P packets, CDL sends out a batch control update packet (BCUP) to each router. Upon receiving a BCUP packet, each router increments its BID. All control packets (BCIP, BCUP, RCUP) are very short (few bits of data) and sent infrequently, thus adding negligible load to the network and not hurting performance. Finally, note that BCIP and BCUP packets are *needed only for PB-batching*, not for our default TB-batching.

Priority Assignment and Enforcement: Before a packet is injected, a router tags it with a priority level using the rank and BID registers (3 bits each in our implementation). At each router, the priority bits in the flit header are utilized by the priority arbiters to allocate VCs and the switch. Each priority arbiter must support at least 6 bits of priority. Fast priority arbiters can be designed using high speed adders as comparators within the arbiters. We use adder delays in [23] to estimate the delay of an 8-bit priority arbiter ($P = 5$,$V = 6$) to be 138.9 picoseconds and the delay of an 16-bit priority arbiter to be 186.9 ps at 45nm technology.

7. Methodology

7.1 Experimental Setup

We evaluate our techniques using a trace-driven, cycle-accurate x86 CMP simulator. Table 1 provides the configuration of our baseline, which contains 64 cores in a 2D, 8x8 mesh NoC. The memory hierarchy uses a two-level directory-based MESI cache coherence protocol. Each core has private write-back L1 caches. The network connects the cores, L2 cache banks, and memory controllers. Each router uses a state-of-the-art two-stage microarchitecture. We use the deterministic X-Y routing algorithm, finite input buffering, wormhole switching, and virtual-channel flow control. We faithfully model all implementation details of the proposed prioritization framework (STC) as well as previously proposed GSF, LocalAge and LocalRR. The parameters used for GSF are: 1) active window size $W = 6$, 2) synchronization penalty $S = 16$ cycles, 3) frame size $F = 1000$ flits. The default parameters used for STC are: 1) ranking levels $R = 8$, 2) batching levels $B = 8$, 3) ranking interval = 350,000 cycles, 4) batching interval = 16,000 cycles, 5) BCIP packet sent every $U = 4000$ cycles. We model all extra control packet traffic. Section 8 evaluates important parameter sensitivity.

7.2 Evaluation Metrics

We evaluate our proposal using several metrics. We measure **application-level system performance** in terms of weighted and

[4]MPI ranking heuristic uses only two 16 bit registers, one for L1 misses and one for number of retired instructions.

[5]To bound the number of bits in RCIP to one flit size, all counters are scaled down. We divide all counters by 32, a value analytically determined assuming maximum bounds on ranking interval and L1 MPI.

[6]This value is determined empirically to balance the overhead of BCIP packets and batching interval.

Processor Pipeline	2 GHz processor, 128-entry instruction window
Fetch/Exec/Commit width	2 instructions per cycle in each core; only 1 can be a memory operation
L1 Caches	32 KB per-core (private), 4-way set associative, 128B block size, 2-cycle latency, write-back, split I/D caches, 32 MSHRs
L2 Caches	1MB banks, shared, 16-way set associative, 128B block size, 6-cycle bank latency, 32 MSHRs
Main Memory	4GB DRAM, up to 16 outstanding requests for each processor, 320 cycle access, 4 on-chip Memory Controllers.
Network Router	2-stage wormhole switched, virtual channel flow control, 6 VC's per Port, 5 flit buffer depth, 8 flits per Data Packet, 1 flit per address packet.
Network Topology	8x8 mesh, each node has a router, processor, private L1 cache, shared L2 cache bank (all nodes) 4 Memory controllers (1 in each corner node), 128 bit bi-directional links.

Table 1: Baseline Processor, Cache, Memory, and Network Configuration

#	Benchmark	NST/packet	Inj Rate	Load	NSTP	Bursty	#	Benchmark	NST/packet	Inj Rate	Load	NSTP	Bursty
1	wrf	7.75	0.07%	low	low	low	19	sjbb	8.92	2.20%	high	high	high
2	applu	16.30	0.09%	low	high	low	20	libquantum	12.35	2.49%	high	high	low
3	perlbench	8.54	0.09%	low	high	low	21	bzip2	5.00	3.28%	high	low	high
4	dealII	5.31	0.31%	low	low	high	22	sphinx3	8.02	3.64%	high	high	high
5	sjeng	8.35	0.37%	low	high	low	23	milc	14.73	3.73%	high	high	low
6	namd	4.59	0.65%	low	low	high	24	sap	4.84	4.09%	high	low	high
7	gromacs	5.19	0.67%	low	low	high	25	sjas	5.15	4.18%	high	low	high
8	calculix	7.10	0.73%	low	low	low	26	xalancbmk	15.72	4.83%	high	high	low
9	gcc	2.14	0.89%	low	low	high	27	lbm	8.71	5.18%	high	high	high
10	h264ref	12.41	0.96%	low	high	high	28	tpcw	5.64	5.62%	high	low	high
11	povray	2.07	1.06%	low	low	high	29	leslie3d	5.78	5.66%	high	low	low
12	tonto	3.35	1.18%	low	low	high	30	omnetpp	2.92	5.72%	high	low	low
13	barnes	7.58	1.24%	low	low	high	31	swim	10.13	6.06%	high	high	low
14	art	42.26	1.58%	low	high	low	32	cactusADM	8.30	6.28%	high	high	low
15	gobmk	4.97	1.62%	low	low	high	33	soplex	8.66	6.33%	high	high	low
16	astar	6.73	2.01%	low	low	low	34	GemsFDTD	4.82	11.95%	high	low	low
17	ocean	9.21	2.03%	low	high	high	35	mcf	5.53	19.08%	high	low	low
18	hmmer	6.54	2.12%	high	low	high							

Table 2: Application Characteristics. NST/packet: Average network stall-time per packet, Inj Rate: Average packets per 100 Instructions, Load: low/high depending on injection rate, Network Stall Cycles per Packet (NSTP): high/low, Bursty: high/low.

harmonic speedup [8], two commonly used multi-program performance metrics based on comparing the IPC of an application when it is run alone versus when it is run together with others. Hmean-speedup balances performance and fairness.

$$W.Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}\ ,\ H.Speedup = \frac{NumThreads}{\sum_i \frac{1}{IPC_i^{shared}/IPC_i^{alone}}}$$

Network stall cycles (NST) is the number of cycles the processor stalls waiting for a network packet [19]. To isolate effects of only the on-chip network, NST *does not include* the stall cycles due to off-chip DRAM access or on-chip cache access. We define **network-related slowdown** of an application as the network-stall time when running in a shared environment (NST^{shared}), divided by, network-stall time when running alone (NST^{alone}) on the same system. The application-level **network unfairness** of the system is the maximum network-related slowdown observed in the system:

$$NetSlowdown_i = \frac{NST_i^{shared}}{NST_i^{alone}}\ ,\quad Unfairness = \max_i NetSlowdown_i$$

7.3 Application Characteristics

We use a diverse set of multiprogrammed application workloads comprising scientific, commercial, and desktop applications. We use the SPEC CPU2006 benchmarks, applications from SPLASH-2 and SpecOMP benchmark suites, and four commercial workloads traces (`sap`, `tpcc`, `sjbb`, `sjas`). In total, we study 35 applications. We choose representative execution phases using PinPoints [24] for all our workloads excluding commercial traces, which were collected over Intel servers. In order to have tractable simulation time, we choose a smaller representative window of instructions (5 million per application), obtained by profiling each application. All our experiments study multi-programmed workloads, where each core runs a separate application. We simulate at least 320 million instructions across 64 processors.

Table 2 characterizes our applications. The reported parameters are for the applications *running alone* on the baseline CMP system without any interference. We categorize applications into three groups based on their network characteristics: applications with 1) high/low load are called *heavy/light*, 2) high Network Stall Cycles per Request Packet (NSTP) are called *sensitive*, and 3) bursty injection patterns are called *bursty*. Our aggregate results are based on 96 different workload combinations.

8. Performance Evaluation

We first compare our base approach (*STC-MPI* ranking with *TB*-batching) to three existing packet prioritization schemes: 1) local round-robin arbitration (*LocalRR*), 2) age-based arbitration (*LocalAge*) and 3) globally synchronized frames (*GSF*), using three case studies to provide insight into the behavior of each scheme with different types of workloads. Figure 7 shows the network slowdowns (the lower the better) of the individual applications. Figures 8 (a) and (b) show the system performance (weighted speedup[7] and harmonic speedup) of the four schemes for different case studies. Section 8.4 reports aggregate results averaged over 96 different workloads, showing that the benefits of our scheme hold over a wide variety of workloads.

8.1 Case Study I: Heavy applications mixed with network-sensitive applications

We mix 16 copies each of two heavy applications (`cactus` and `lbm`) and two network-sensitive applications (`art` and `libquantum`). The following observations are in order:

• The baseline LocalRR policy slows down all applications other than `art`, which has very high NST/packet when run alone (the potential negative impact of interference in the network is lower for `art` than for other applications). LocalRR slows down other applications in hard-to-predict ways as round-robin port allocation is application-oblivious.

• LocalAge policy significantly reduces the network slowdown of the heaviest application (`cactus`) compared to LocalRR, while increasing the slowdowns of all other applications. LocalAge implicitly prioritizes heavier applications because older packets in the network are more likely to be from the heavier applications. Overall, LocalAge reduces performance by 5.9%/3.8% (weighted/harmonic

[7]Weighted speedup is divided by number of cores (=64) for clarity.

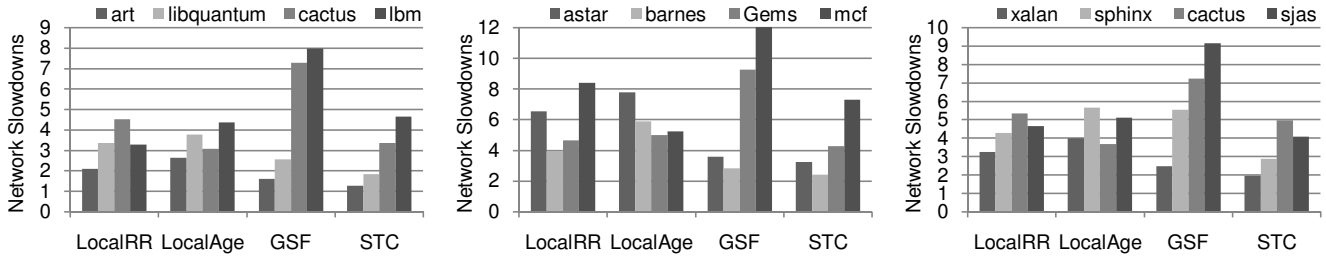

Figure 7: Network slowdown of applications: (a) Case Study I (b) Case Study II (c) Case Study III

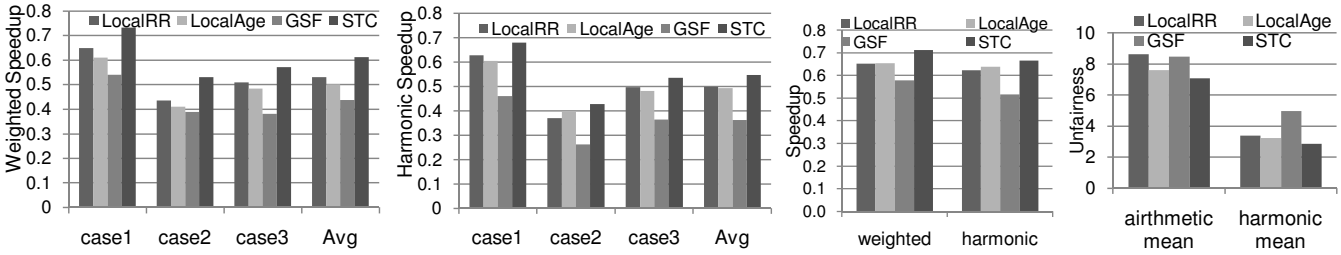

Figure 8: System performance results: (a) Weighted speedup of case studies (b) Harmonic speedup of case studies (c) Aggregate Speedup for 96 workloads and (d) Aggregate Unfairness for 96 workloads

speedup) over LocalRR as it delays light applications in the network. These applications could have made fast progress had their packets been quickly serviced. Hence, implicitly prioritizing network-heavy applications in the network leads to a loss of system throughput, motivating the need for better prioritization mechanisms. On the other hand, LocalAge is a fair policy, leading to the smallest maximum network slowdown: the heaviest application has the highest network slowdown in LocalRR (because it gets delayed significantly by other copies of `cactus` running on the system) and LocalAge reduces this maximum slowdown by implicitly prioritizing the heavy application.

• GSF unfairly penalizes the heavy applications, as explained in Section 5, because they quickly run out of frames and stop injecting. Thus, the network slowdowns of `cactus` and `lbm` increase by 1.6X and 2.4X over LocalRR. Since GSF guarantees minimum bandwidth to all applications, it improves the network slowdown of network-sensitive applications over both of the local policies (by 24.2% for `art` and 22.3% for `libquantum` over LocalRR) by ensuring that heavier applications do not deny service to the lighter applications. However, GSF does not prioritize any application within a frame, thus there is scope for further reducing the slowdowns of network-sensitive applications. Overall, GSF significantly degrades system throughput (by 16.6% over LocalRR) and application-level fairness (by 1.8X over LocalRR) because it unfairly penalizes the heavy applications, causing their cores to make very slow progress, thereby reducing system utilization.

• STC prioritizes the network-sensitive applications using ranking, and ensures, using batching, that the heavy applications are not overly penalized. The result is that it significantly reduces the slowdown of network-sensitive applications as their packets' prioritization reduces their network stall cycles per packet (NSTP), e.g. `art`'s NSTP reduces from 219.5 cycles (LocalRR) to 99.8. STC slightly degrades the network slowdown of the heavy applications (by 6.2%/9.4% over LocalAge). This increase is small for two reasons: 1) batching prevents the starvation of heavy applications, 2) heavy applications are more latency tolerant due to the lower stall-time criticality of their requests: for example, compared to the NSTP of `art` (219.5 cycles), `lbm`'s NSTP is only 36.2 cycles. Overall, STC improves system throughput by 12.8%/8.2% (weighted/harmonic) over LocalRR and 19.8%/12.4% over LocalAge, because it enables network-sensitive applications to make faster progress without significantly slowing down heavy applications that are slow to begin with.

8.2 Case Study II: Heavy applications mixed with light applications

We run 16 copies each of two heavy applications (`Gems` and `mcf`) with 16 copies each of two light applications (`astar` and `barnes`). The purpose is to show the behavior of STC when network-intensive applications are run together with compute-intensive applications, a likely scenario in future multi-core systems. We make the following key observations:

• LocalRR's network slowdowns are in general higher than in Case Study I, since `mcf` and `Gems` are much heavier workloads than `cactus` and `lbm`, thus slowing down themselves (i.e. other copies of `mcf` and `Gems`) and other applications more.

• The local policies and GSF show similar behavior compared to Case Study I. LocalRR is agnostic of applications, and hence, slows down different type of applications in difficult-to-predict ways: `astar` (light) and `mcf` (heavy) experience the highest slowdowns. LocalAge implicitly prioritizes heavy applications as packets from those applications are likely to be older. As a result, LocalAge reduces system performance by 9.1% (weighted speedup) compared to LocalRR. In contrast, GSF has the opposite effect: it implicitly slows down heavy applications, reducing both system performance (by 10.6% over LocalRR) and fairness (by 1.5X over LocalRR). Hence, we conclude that neither LocalAge nor GSF work well with a heterogeneous mix of applications as they tend to implicitly penalize light or heavy applications respectively, which leads to reduced overall system utilization.

• STC provides large system throughput improvements over all previous schemes by prioritizing light applications over heavy applications. Overall, STC's system performance improvement is 29.5% over LocalAge and 21.7% over LocalRR. STC greatly reduces the packet-latency/NSTP of the lighter applications (from 112.8/109.8 cycles with LocalAge to 41.9/38.4 for `barnes`) while only slightly increasing the packet-latency/NSTP of heavy applications (from 119.7/27.0 cycles to 123.4/29.4 for `Gems`). As a result, average stall-time of the applications in the network reduces, leading to the observed system throughput improvements. As batching ensures that no application's packets are delayed indefinitely, STC also provides the best fairness. We conclude that STC can provide *both the best system throughput and system fairness* when a set of memory-intensive and compute-intensive applications are executed concurrently.

8.3 Case Study III: Mix of heavy applications

We run 16 copies each of four heavy applications (`xalan`, `sphinx`, `cactus`, and `sjas`). The purpose of this case study is to show the dynamic behavior of STC: applications may have similar average behavior (e.g., all are heavy), but if their transient behaviors are sufficiently different, STC can still provide high performance gains. The following observations are in order:

• `sjas` is extremely bursty, as shown in Figure 9(c). It is penalized severely by GSF during its "heavy" bursts by running out of frames and being throttled, causing GSF to lose throughput (Figure 8(a)) and fairness (Figure 7(c)). As in previous case studies, LocalAge slows down the applications with relatively low injection rates (`sphinx` and `sjas`).

• The average injection rates (Figure 9(a)) of the four applications are similar: the ratio of the minimum and maximum injection rates in the mix is 1.6X (vs. 3.9X in Case Study II). The average L1 miss ratios (Figure 9(b)) of the four applications are also similar: the ratio of the minimum and maximum L1 miss rates in the mix is 1.9X (vs. 5.8X in Case Study I). One might, therefore, conclude that prioritizing lighter applications or those with lower MLP (i.e., "critical applications") might not be very advantageous. However, STC improves system throughput by 12.2% over LocalRR and 18.1% over LocalAge. This is because STC is able to adapt to the dynamic changes in the behavior of applications: even though average miss rates of applications are similar, each application's instantaneous miss rate varies over time (Figure 9(c)). Because STC determines application priorities on an interval basis, it is able to adapt its prioritization to correctly identify the applications that benefits from prioritization in each interval. For example, in Figure 9(c), during intervals 0-21, STC improves system throughput by prioritizing `sphinx`'s packets, whereas during intervals 40-60, STC similarly improves system throughput by prioritizing `sjas`'s packets. Hence, STC is able to identify and prioritize the lightest/critical application in a given interval, leading to improvements even when all applications seemingly have the same average behavior.

8.4 Overall Results Across 96 Multi-Programmed Workloads

Figures 8 (c) and (d) compare the four prioritization techniques averaged across 96 workload mixes. 48 workloads have a mix with four applications (16 copies each) and 48 workloads have a mix with eight applications (8 copies each). Each set of 48 workloads is spread out across the design space: 16 workloads represent low network utilization (all applications are light), 16 workloads represent medium network utilization (half light, half heavy) and 16 workloads represent high network utilization (all heavy). Within each of the three categories, applications are *randomly picked* to form the 16 workloads. The aggregate results are consistent with the observations made in the three case studies. On average, STC improves system throughput by 9.1%/4.3% (weighted/harmonic) compared to the best previous scheme (LocalAge), while also improving network fairness by 5.7%. The highest performance improvement of our mechanism is 33.7%, the lowest -1.8% over the best existing prioritization policy. We conclude that STC provides the best system performance and network fairness over a very wide variety of workloads.

8.5 Effect of Ranking Heuristics

Figures 10(a) and 11(a) show the effect of different ranking heuristics: ASCP, ReqQueue, and MPI (Section 4.2) and two further heuristics to provide comparison points, RoundRobin (assign ranks in a round-robin manner that changes every ranking-interval) and Random (assign ranks randomly every interval). First, all the heuristics perform better than or at least equivalent to LocalRR, LocalAge, and GSF. ASCP, which ranks the applications according to NST/packet, is the most inefficient and unfair STC-aware heuristic. There are two reasons: 1) the positive feedback loop discussed in Section 4.2, which leads to some applications to be unfairly stuck in a low rank for a long period of time and 2) ASCP accelerates applications with *higher stall-time per packet even if these applications are heavy*. As we saw before, prioritizing a heavy application in the network slows down almost all other applications. MPI provides the best performance because it effectively prioritizes light applications over heavy ones from the perspective of the network. We conclude that prioritizing only "light and stall-time-critical" applications can be done effectively in the network without significantly hurting other applications' performance. Overall, MPI provides a good trade-off between performance, fairness, and implementation complexity. Finally, note that RoundRobin and Random ranking, which do not take into account application behavior in ranking, result in much lower system performance than other ranking techniques. This further indicates that MPI is effective in capturing applications' network criticality and intensity.

8.6 Effect of Batching Policy

Figures 10(b) and 11(b) show the effect of different batching policies, discussed in Section 4.3, as well as not using batching. Not using batching (rightmost bars) results in the lowest system performance because it starves low-ranked applications for long periods of time. Note that there is not a significant performance or fairness difference between packet-based or time-based synchronized batching. Since time-based synchronized batching is simpler to implement, we conclude that synchronized time-based batching provides the best tradeoff between performance, fairness, and implementation complexity.

8.7 Effect of Local Arbitration Policy

Figures 10(c) and 11(c) show the effect of different local arbitration policies used with our mechanism. The local arbitration policy is invoked only if the rank and batch numbers of two packets are equal (this is the "Local Router Rule" specified in Section 4.4). We compare three local polices 1) *Age* (packet-age based prioritization) 2) *RoundRobin* (RR; round-robin port prioritization) and 3) *InstAge* (a new policy where a packet is assigned the age, i.e. current time minus the fetch time, of the instruction it is initiated by). Overall, there is no single best policy in terms of performance, fairness, and implementation complexity. The *Age* policy is the fairest and provides slightly better performance than the other two. However, it is more complex to implement because it requires 1) the age of a flit to be tracked in the network and 2) more complex arbitration logic to compare flit ages. The *RoundRobin* policy, has the lowest implementation cost, but has slightly lower performance and is slightly more unfair than the other two. The *InstAge* policy is in-between in terms of performance, fairness, and complexity. We conclude that our proposal's performance or fairness is not significantly affected by the local arbitration policy of the routers.

8.8 Effect of Ranking and Batching Intervals

The graphs in Figure 12 show the effect of different ranking and batching intervals on STC performance and fairness. Both intervals have a higher impact on fairness than performance. Figure 12 shows that: 1) performance is lower with smaller ranking intervals because highly fluctuating ranks eliminate the benefits of ranking, 2) unfairness increases for very large ranking intervals because it fails to adapt to changes in applications' network intensity, thereby unfairly penalizing bursty applications.

The two rightmost graphs in Figure 12 show that batching interval B determines the trade-off between fairness and performance. A smaller B leads to high fairness by reducing starvation to a minimum, especially for Case Study III, which contains applications with bursty behavior. On the other hand, a smaller B also leads to reduced system performance as the granularity at which higher-ranked applications are prioritized within the network (and hence system throughput maximized) becomes smaller. When B becomes too large, fairness starts to degrade since batching is essentially eliminated for long time periods.

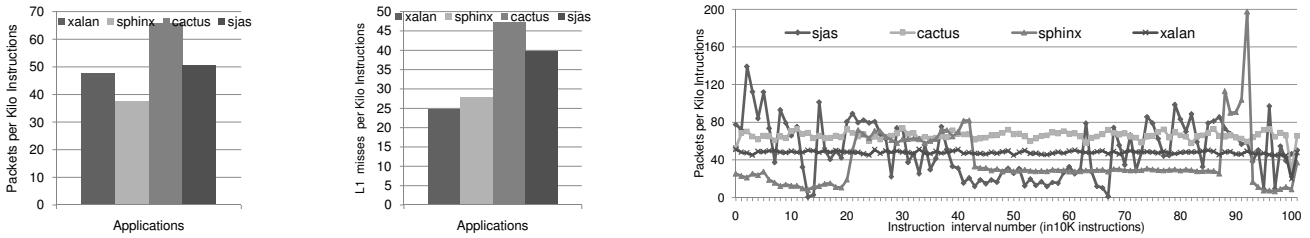

Figure 9: Injection rate (packets per kilo instructions) of Case Study III applications for an intermediate one million instructions window (a) Average Injection Rate (b) Average L1 mpki (c) Injection rate during each 10K-instruction interval

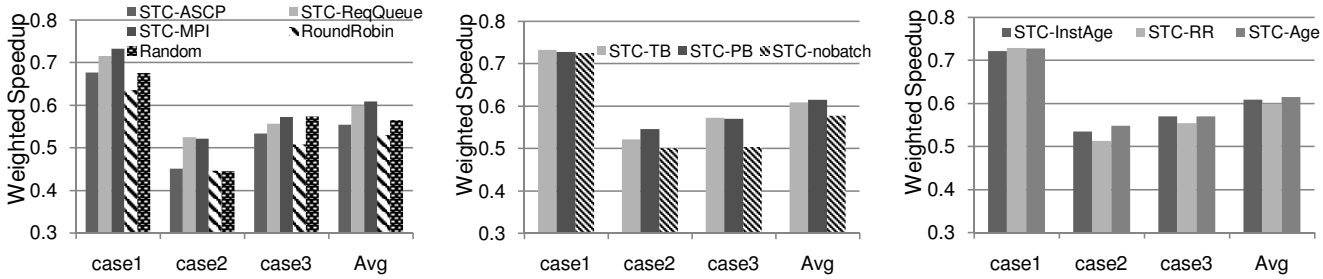

Figure 10: Performance impact of different (a) Ranking Heuristics (b) Batching Policies (c) Local Arbitration Polices

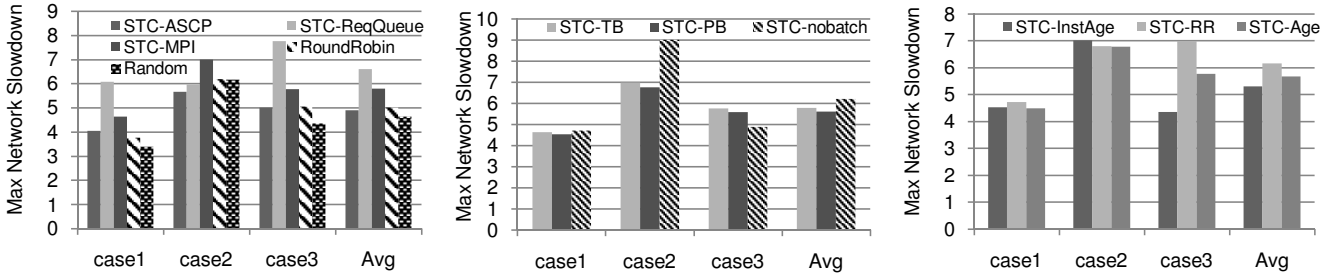

Figure 11: Fairness impact of different (a) Ranking Heuristics (b) Batching Policies (c) Local Arbitration Polices

8.9 Effect of Enforcing System-Level Application Weights/Priorities

We evaluated the effectiveness of our prioritization substrate in enforcing system-software-assigned application priorities for a variety of scenarios and present three representative case studies. Figure 13(a) shows the network slowdown of four groups of 16 xalan applications (8x8 network) where, each group has weights of 1, 2, 2, and 8, respectively. Figure 13(b) shows the network slowdown of eight groups of 8 xalan applications where each group respectively has weights of 1, 2, 3, ..., 8. LocalAge and LocalRR schemes treat all different-weight applications the same because they are application-unaware. As a result, all applications slow down similarly. In contrast, our mechanism enforces application weights as configured by the OS. Each application slows down inverse-proportionally to its weight: higher-weight (i.e., more important) applications experience the smallest slowdowns whereas lower-weight (i.e., less important) applications experience the highest slowdowns in the network.

Figure 13(c) shows that our proposal is also able to enforce application weights in a heterogeneous workload consisting of different applications. We conclude that our mechanism is effective at enforcing system-level application priorities within the network by allowing application-aware prioritization to be configurable by the system software.

Note that we also evaluated GSF by ensuring it allocates network bandwidth proportionally to each application's weight. While GSF is able to enforce application weights in the network, it does so at significantly reduced system throughput compared to STC, as shown in Figure 13(d). GSF also slows down the lowest-weight application significantly more than STC (see Figures 13(b) and 13(c)) because when this application runs out of frames, it cannot inject into the network.

9. Related Work

To our knowledge, no previous work proposed *application-level prioritization mechanisms* to optimize application-level system throughput in NoCs. Here, we briefly describe the most closely related previous work.

Prioritization and Fairness in On-Chip/Off-Chip Networks: We have already compared our approach extensively, both qualitatively and quantitatively, to existing local arbitration (LocalAge,LocalRR) and state-of-the-art QoS-oriented prioritization (GSF [17]) policies in NoC. Other frameworks for QoS [2, 26] have been proposed in on-chip networks. Mechanisms for QoS can possibly be combined with our approach. Similarly, a large number of arbitration policies [32, 4, 10, 1] have been proposed in multi-chip multiprocessor networks and long-haul networks. The goal of these mechanisms is to provide fairness or guaranteed service, while ours is to improve overall system throughput without degrading fairness, while being sufficiently configurable to allow the operating system to enforce application priorities.

Bolotin et al. [3] propose prioritizing control packets over data packets in the NoC, but do not distinguish packets based on which application they belong to. Our mechanism can be combined with their simple prioritization heuristic.

Flow-Based Prioritization in Off-Chip Networks: Previous work [32, 6, 33] explored mechanisms that statically assign priorities/bandwidth to different flows in off-chip networks to satisfy real-time performance and QoS guarantees. These works assume that each flow's priority is known a priori, and hence each packet's priority is simply set to this known priority. In contrast, our mechanism does not assume an application's priority is known. STC can dynamically determine the relative priority of each application (and hence each packet) to optimize overall system performance. Note that our scheme is still

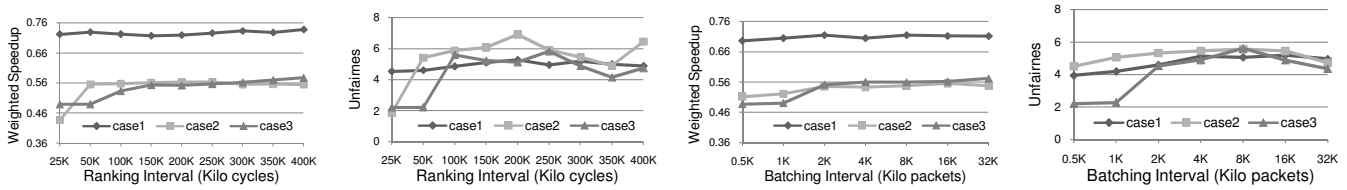

Figure 12: Effect of Ranking and Batching Intervals

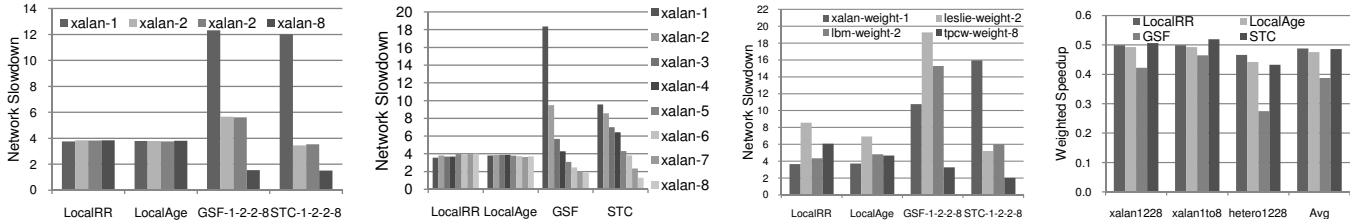

Figure 13: Enforcing OS Priorities:(a) A homogenous workload mix with weights 1-2-2-8 (b) A homogenous workload mix with weights 1-to-8 (c) A heterogeneous workload mix with weights 1-2-2-8 (d) Weighted Speedup of all mechanisms for the three workloads

able to enforce statically-known priorities, as Section 4.2 shows.

Criticality of Memory Accesses and Memory Level Parallelism: There has been extensive research on predicting criticality of memory accesses [28, 9, 29] and prioritizing critical accesses in the processor core and caches. It has also been shown that system performance can be improved by designing MLP-aware cache replacement [25] and memory scheduling policies [20]. Our work is related to these works only in the sense that we also exploit criticality and MLP to improve system performance. However, our mechanisms are very different due to the distributed nature of on-chip networks. To our knowledge, the concepts of criticality and MLP were not previously exploited in on-chip networks to improve system performance.

Batching: We propose packet batching in NoC for starvation avoidance. The general concept of batching has been used in disk scheduling [30] and memory scheduling [20] to prevent the starvation of I/O and memory requests. The concept of frames used in [17] is analogous to packet batching.

10. Conclusion

We introduce a novel, comprehensive application-aware prioritization framework to improve application-level system throughput in NoCs. We identify the concept of stall-time criticality of packets, and provide mechanisms to prioritize applications with critical packets across routers, while guaranteeing starvation freedom. Averaged over 96 randomly-generated multiprogrammed workload mixes on a 64-core 8x8-mesh CMP, the proposed policy improves overall system throughput by 9.1% on average (and up to 33.7%) over the best existing prioritization policy, while also reducing application-level unfairness. While effective at improving both system performance and fairness, our proposal is also configurable and thus enables the enforcement of system-level application priorities, without hurting overall performance. We conclude that the proposed prioritization framework provides a promising way to build many-core NoCs that provide high system performance, fairness, and flexibility.

Acknowledgments

This research is supported in part by NSF grant CCF 0702519. The authors would like to thank Microsoft for their generous support. We thank Miray Kas and the anonymous reviewers for suggestions.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. king Su. Myrinet - A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 1995.

[2] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Arch.*, 2004.

[3] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny. The Power of Priority: NoC Based Distributed Cache Coherency. In *NOCS'07*, 2007.

[4] A. A. Chien and J. H. Kim. Rotating Combined Queueing (RCQ): Bandwidth and Latency Guarantees in Low-Cost, High-Performance Networks. *ISCA-23*, 1996.

[5] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[7] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, 1997.

[8] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, May-June 2008.

[9] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *ISCA-25*, 2001.

[10] D. Garcia and W. Watson. Servernet II. *Parallel Computing, Routing, and Communication Workshop*, June 1997.

[11] A. Glew. MLP Yes! ILP No! Memory Level Parallelism, or, Why I No Longer Worry About IPC. In *ASPLOS Wild and Crazy Ideas Session*, 1998.

[12] J. A. Hartigan. *Clustering Algorithms*. Morgan Kaufmann, 1975.

[13] J. Kim, J. Balfour, and W. Dally. Flattened butterfly topology for on-chip networks. *MICRO-40*, 2007.

[14] J. Kim, C. Nicopoulos, D. Park, R. Das, Y. Xie, V. Narayanan, M. S. Yousif, and C. R. Das. A novel dimensionally-decomposed router for on-chip communication in 3D architectures. *ISCA-34*, 2007.

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

[16] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha. Express virtual channels: Towards the ideal interconnection fabric. In *ISCA-34*, 2007.

[17] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks. In *ISCA-35*, 2008.

[18] O. Mutlu, H. Kim, and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 2006.

[19] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO-41*, 2007.

[20] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA-35*, 2008.

[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.

[22] C. A. Nicopoulos, D. Park, J. Kim, N. Vijaykrishnan, M. S. Yousif, and C. R. Das. Vichar: A dynamic virtual channel regulator for network-on-chip routers. In *MICRO-39*, 2006.

[23] V. G. Oklobdzija and R. K. Krishnamurthy. *Energy-Delay Characteristics of CMOS Adders, High-Performance Energy-Efficient Microprocessor Design*, chapter 6. Springer US, 2006.

[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intelő itaniumő programs with dynamic instrumentation. In *MICRO-37*, 2004.

[25] M. Qureshi, D. Lynch, O. Mutlu, and Y. Patt. A Case for MLP-Aware Cache Replacement. In *ISCA-33*, 2006.

[26] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. *DATE*, 2003.

[27] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS-8*, 2000.

[28] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *MICRO-31*, 1998.

[29] S. Subramaniam, A. Bracy, H. Wang, and G. Loh. Criticality-based optimizations for efficient load processing. In *HPCA-15*, 2009.

[30] T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 1972.

[31] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 1967.

[32] K. H. Yum, E. J. Kim, and C. Das. QoS provisioning in clusters: an investigation of router and NIC design. In *ISCA-28*, 2001.

[33] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. *SIGCOMM*, 1990.