Aérgia: Exploiting Packet Latency Slack in 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

Traditional Network-on-Chips (NoCs) employ simple arbitration strategies, such as round-robin or oldest-first, to decide which packets should be prioritized in the network. This is suboptimal since different packets can have very different effects on system performance due to, e.g., different level of memory-level parallelism (MLP) of applications. Certain packets may be performance-critical because they cause the processor to stall, whereas others may be delayed for a number of cycles with no effect on application-level performance as their latencies are hidden by other outstanding packets' latencies. In this paper, we define slack *as a key measure that characterizes the relative importance of a packet. Specifically, the slack of a packet is the number of cycles the packet can be delayed in the network with no effect on execution time. This paper proposes new router prioritization policies that exploit the available slack of interfering packets in order to accelerate performance-critical packets and thus improve overall system performance. When two packets interfere with each other in a router, the packet with the lower slack value is prioritized. We describe mechanisms to estimate slack, prevent starvation, and combine slack-based prioritization with other recently proposed application-aware prioritization mechanisms.*

We evaluate slack-based 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 21.0% over the commonly-used round-robin policy. Averaged over 56 randomly-generated multiprogrammed workload mixes, the proposed policy improves system throughput by 10.3%, while also reducing application-level unfairness by 30.8%.*

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, slack, criticality

ISCA'10, June 19–23, 2010, Saint-Malo, France.

1. Introduction

Network-on-Chips (NoCs) are widely viewed as the de facto solution for integrating a large number of components future microprocessors will consist of. In contrast to ad-hoc point-to-point global wiring, shared buses or monolithic crossbars, NoCs are scalable and have well-controlled and highly predictable electrical properties. It is thus foreseeable that on-chip networks will become one of the most critical shared resources in many-core systems, and that the performance of such systems will heavily depend on the resource sharing policies employed in the on-chip networks. Therefore, devising efficient and fair scheduling strategies is particularly important (but also challenging) when the network is shared by diverse applications, with potentially different requirements.

Managing shared resources in a highly parallel system is one of the most fundamental challenges that we face. While interference of applications is relatively well understood for many important shared resources, such as shared last-level caches [18, 30, 5, 4, 11] or memory bandwidth [26, 23, 24, 19], less is known about the interference behavior of applications in NoCs, and the impact of this interference on applications' execution times. One reason why analyzing and managing multiple applications in a shared NoC is challenging is that application interactions in a distributed system can be complex and chaotic, with numerous first- and second-order effects (queueing delays, different memory-level parallelism (MLP), burstiness, impact of the spatial location of cache banks, etc.) and hard-to-predict interference patterns that can have a significant impact on application-level performance.

A major algorithmic question that governs application interactions in the network is the NoC router's *arbitration policy*, i.e., which packet to prioritize if two or more packets arrive at the same time, and they want to take the same output port. Traditionally, arbitration policies used in on-chip networks have been very simple heuristics, including round-robin (RR) and age-based arbitration (oldest-first). These arbitration policies treat all packets equally, irrespective of source applications' characteristics. In other words, these policies arbitrate between packets as if each packet had exactly the same impact on application-level performance. In reality, however, applications can be diverse with unique and dynamic characteristics/demands. Different packets can have a vastly different importance to their respective application's performance. In the presence of memory level parallelism (MLP) [16, 22], although there might be multiple outstanding load misses in the system, not every load miss is a *bottleneck-causing (i.e. critical)* miss [14]. Assume, for example, that an application issues two concurrent network requests, one after another, first to a remote node in the network, and second to a closeby node. Clearly, the packet going to the closeby node is less critical, because even if it is delayed for several cycles in the network, its latency will be hidden from the application by the packet going to the distant node, which takes more time. Thus, the delay tolerance of each packet with regard to its impact on its application's performance can be different.

In this paper, we exploit this diversity of packets to design higher-performance and more application-fair NoCs. We do so by differ-

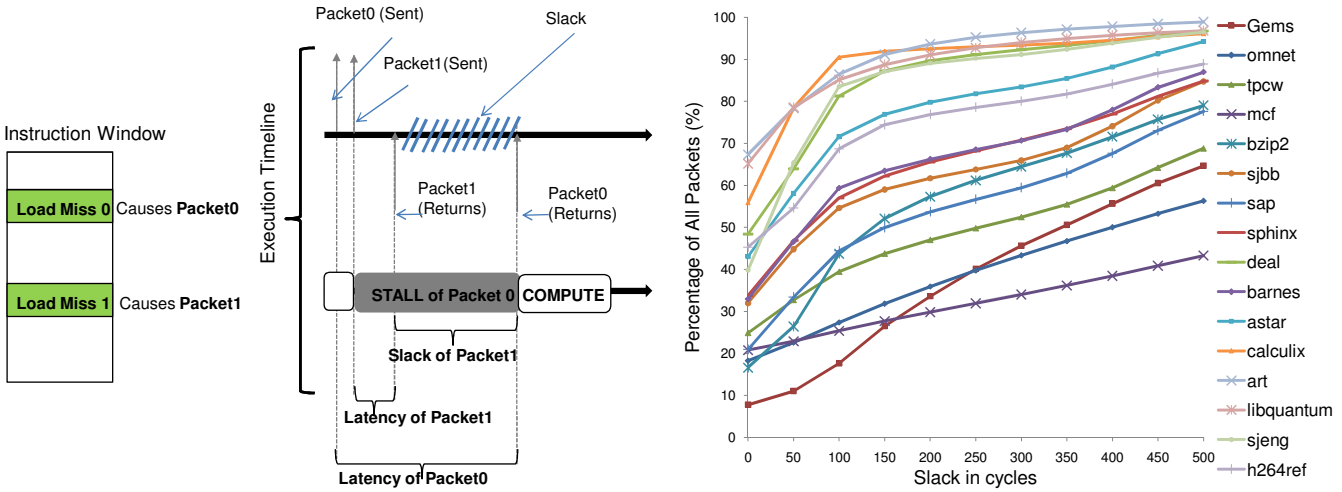

Figure 1: (a) The concept of slack, (b) Packet distribution of applications based on slack cycles

entiating packets based on their *slack*, a measure that captures the packet's importance to its application's performance. Particularly, we define the slack of a packet to be the number of cycles the packet can be delayed in the network without having an effect on the application's execution time. Hence, a packet is relatively non-critical until the time it spends in the network exceeds the packet's available slack. In comparison, increasing the latency of packets with no available slack (by de-prioritizing them during arbitration in NoC routers) will cause the application to stall.

We describe *Aérgia*[1], a new NoC architecture that contains new router prioritization mechanisms to accelerate the critical packets with low slack values by prioritizing them over packets with larger slack values. To this end, we devise techniques to efficiently estimate the slack of a packet dynamically. Prior to injection into the network, each packet is tagged with a priority that depends on the estimated slack of the packet. We propose extensions to routers to prioritize packets with lower slack at the contention points within the router (i.e., buffer allocation and switch arbitration). Finally, to ensure forward progress and starvation freedom from prioritization, *Aérgia* groups packets into batches, and ensures that all the packets from an earlier batch are serviced before packets from a later batch. Experimental evaluations on a cycle-accurate simulator show that our proposal effectively increases overall system throughput and application-level fairness in the NoC. To summarize, the main **contributions** of this paper are the following:

- We observe that the packets in the network have varying degree of slack, and analyze the correlation of slack to application characteristics, such as the application's MLP. We observe that existing arbitration policies schedule packets regardless of their slack, and that state-of-the-art global application-aware prioritization (STC) mechanisms proposed in [8] can sometimes lead to unfairness because they treat packets of an application equally.
- We propose novel, slack-aware prioritization mechanisms (online slack estimation, and slack-aware arbitration) to improve application-level system throughput and application-level fairness in NoCs. Our key idea is to dynamically estimate the available slack of each network packet and prioritize critical packets with lower available slack in the network.
- We qualitatively and quantitatively compare our proposal to previous local and global arbitration policies, GSF [21], and STC [8]. We show that our proposal provides the highest *overall system throughput* as well as the best *application-level fairness* (10.3% improvement in weighted speedup and 30.8% reduction in unfairness over the baseline round-robin policy across 56 diverse multiprogrammed workloads).
- We show that Aérgia is a fine-grained prioritization mechanism that can be used as a *complementary technique with any other prioritization substrate* to improve performance and fairness. Thus, using Aérgia within the STC framework improves both system performance and fairness, respectively by 6.7% and 18.1% compared to STC over 56 diverse multiprogrammed workloads.

[1]Aérgia is the female spirit of laziness in Greek mythology. Inspired from the observation that packets in Network-on-Chip can afford to slack (off), we name our architecture Aérgia.

2. Motivation

2.1 Concept of Slack

Modern microprocessors employ several memory latency tolerance techniques (e.g., out-of-order execution [35] and runahead execution [10, 25, 22]) to hide the penalty of load misses. These techniques exploit *Memory Level Parallelism* (MLP) [16] by issuing several memory requests in parallel with the hope of overlapping future load misses with current load misses [22]. In the presence of MLP in an on-chip network, the existence of multiple outstanding packets leads to overlap of packets' latencies, which introduces slack cycles. Intuitively, the *slack of a packet* is the number of cycles the packet can be delayed without affecting the overall execution time. For example, consider the processor execution time-line shown in Figure 1 (a). In the instruction window, the first load miss causes a packet (*Packet0*) to be sent into the network to service the load miss, and the second load miss generates the next packet (*Packet1*). In the execution time-line of Figure 1 (a), *Packet1* has lower network latency than *Packet0*, and returns to the source earlier. Nonetheless, the processor cannot commit *Load0* and stalls until *Packet0* returns. Thus, *Packet0 is the bottleneck packet*, which *allows the earlier-returning Packet1 some slack cycles* as shown in the figure. *Packet1* could be delayed for the slack cycles without causing significant application-level performance loss.

2.2 Advantages of Exploiting Slack

In this paper, we show that a network-on-chip architecture that is aware of slack can lead to better system performance and fairness. The number of slack cycles a packet has is an indication of the importance or criticality of that packet to the processor core. If the NoC were aware of the remaining slack of a packet, it could take arbitration decisions that would accelerate packets with a small slack, which would improve the overall performance of the system.

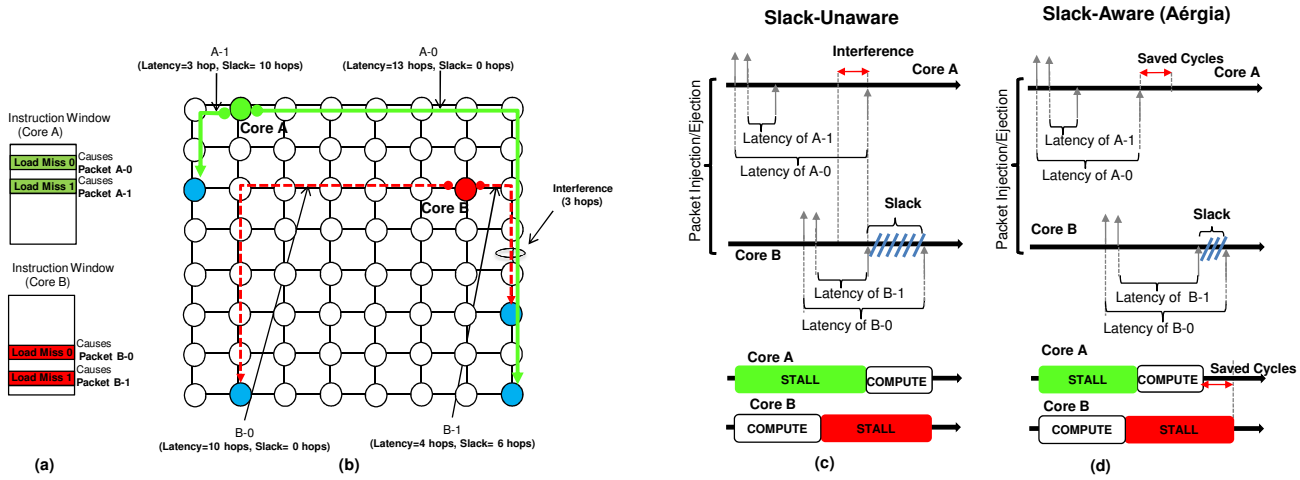

Figure 2: Conceptual example showing the advantage of incorporating slack into prioritization decisions in the NoC

Figure 2 shows a motivating example. Figure 2(a) shows the instruction window of two processor cores (Core A at node (1,2) and Core B at node (3,7)). The network consists of an 8x8 mesh as shown in Figure 2(b). Core A generates two packets (*A-0* and *A-1*). The first packet (*A-0*) is not preceded by any other packet and is sent to node (8,8), hence it has a latency of 13 hops and a slack of 0 hops. In the next cycle, the second packet (*A-1*) is injected towards node (3,1). This packet has a latency of 3 hops, and since it is preceded (and thus overlapped) by the 13-hop packet (*A-0*), it has a slack of at least 10 hops (13 hops - 3 hops). Core B also generates two packets, (*B-0* and *B-1*). The latency and slack of Core B's packets can be calculated similarly. *B-0* has a latency of 10 hops and slack of 0 hops. *B-1* has a latency of 4 hops and slack of 6 hops.

The concept of slack can be exploited when there is interference between packets with different slack values. In Figure 2(b), for example, packets *A-0* and *B-1* interfere. The critical question is, which of the two packets should be prioritized by the router, and which one should be queued/delayed?

Figure 2(c) shows the execution time-line of the cores with the baseline slack-unaware prioritization policy that prioritizes packet *B-1*, thereby delaying packet *A-0*. In contrast, if the routers were aware of the slack of packets, they would prioritize packet *A-0* over *B-1* because the former has smaller slack, as shown in Figure 2(d). Doing so would reduce the stall time of Core A *without significantly increasing* the stall time of Core B, thereby improving overall system throughput, as Figure 2 shows. The critical observation is that delaying a packet with a higher slack value can improve overall system performance by allowing a packet with a lower slack value to stall its core less.

2.3 Diversity in Slack and Its Analysis

The necessary condition to exploit slack is that there should be sufficient diversity in slack cycles of interfering packets. In other words, it is only possible to benefit from prioritizing low-slack packets if the network contains a good mix of both high- and low-slack packets at any time. Fortunately, we find that there exists sufficient slack diversity in realistic systems and applications.

Let us define the term *predecessor packets*, or simply *predecessors* for a given packet P as all packets that are i) still outstanding and ii) have been injected into the network earlier than P. Conversely, we call subsequent packets to be successor packets (or *successors*). The number of slack cycles for each packet varies within application phases as well as between different applications. This is primarily due to variations in 1) *latency of predecessors* and 2) *number of predecessors* for a given packet. Specifically, a packet is likely to have high slack if it has high-latency predecessors or if it has a large number of predecessors, since in both cases, the likelihood of its latency to be overlapped (i.e. its slack to be high) is higher.

The *latency of a predecessor* varies depending on whether it is an L2 cache hit or miss. An L2 miss has very high latency due to off-chip DRAM access, and therefore, causes very high slack to its successor packets (unless the successors are also L2 misses). Also, as demonstrated in the above example, predecessor latency varies depending on the distance or number of hops the predecessor packet has to traverse in the network. The *number of predecessor packets* is determined by the injection rate of packets (i.e., L1 cache miss rates) as well as the burstiness of packet generation of a source application running on a particular node.

Figure 1(b) shows a cumulative distribution function of the diversity in slack cycles for different applications.[2] The X-axis shows the number of slack cycles per packet.[3] The Y-axis shows the fraction of total packets that have *at least* as many slack cycles per packet as indicated by the X-axis. Two trends are visible. First, most applications have a good spread of packets with respect to number of slack cycles, indicating sufficient *slack diversity within an application*. For example, 17.6% of `Gems`' packets have at most 100 slack cycles, but 50.4% of them have more than 350 slack cycles. Second, different applications have different slack characteristics. For example, the packets of `art` and `libquantum` have much lower slack in general than `tpcw` and `omnetpp`.

3. Characterization and Online Estimation of Slack

The premise of this work is to identify critical packets with low slack and accelerate them by de-prioritizing packets with high slack cycles. In this section, (i) we formally define slack (ii) since slack itself cannot be directly measured, we perform detailed characterization of slack to find indirect metrics that correlate with slack (iii) we then show how these metrics can be quantized into specific priority levels and (iv) finally, we describe on-line techniques to dynamically approximate the indirect metrics (and hence slack) in an NoC.

[2]Although our final performance evaluation spans a suite of 35 applications, in all our characterization plots we show a subset of 16 applications for clarity. These 16 applications represent equally, heavy, medium, light utilization and network sensitive applications. We collected these statistics over 56 randomly generated workloads (See Section 6 for system configuration).

[3]The value on the X axis (slack cycles per packet) is proportional to end-to-end predecessor packet latency (which includes DRAM access latency and all network-on-chip transactions). Hence, we characterize slack up to the potentially maximum memory access latency of 1700 cycles.

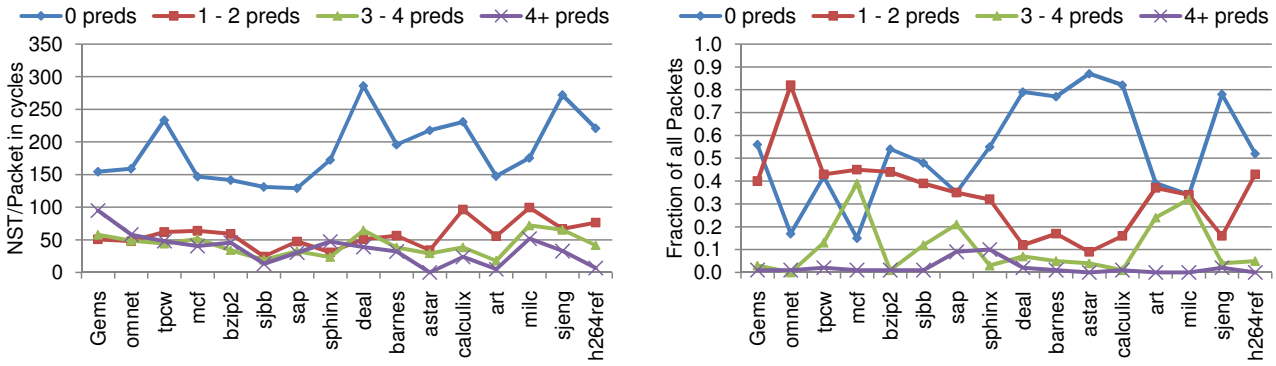

Figure 3: (a) Packet criticality based on number of miss-predecessors, (b) Distribution of packets based on number of miss-predecessors

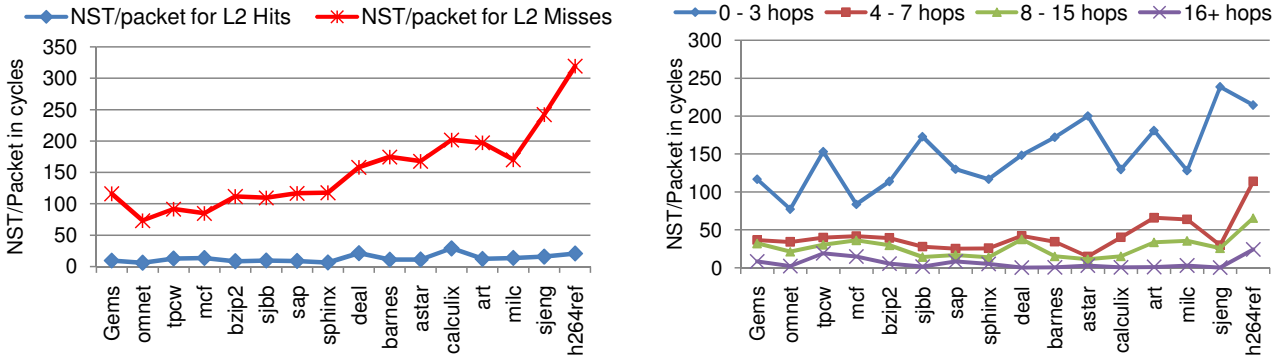

Figure 4: Packet criticality based on (a) L2 cache hit vs. miss (b) slack in terms of hops

3.1 Slack Definition

Slack can be defined locally or globally [13]. Local slack is the number of cycles a packet can be delayed without delaying *any subsequent instruction*. Global slack is the number of cycles a packet can be delayed without delaying the *last instruction in the program*. The computation of a packet's global slack requires critical path analysis of the entire program, rendering it impractical in our context. Hence, we focus on local slack. The ideal local slack of a packet could depend on instruction-level dependencies, which are hard to capture in the NoC. Hence, in order to keep the implementation in the NoC simple, we conservatively consider slack only with respect to outstanding network transactions. We formally define the available *local network slack* (or in this paper simply *slack*) of a packet to be the difference between the maximum latency of its predecessor packets (i.e., any *outstanding* packet that was injected into the network earlier than this packet) and its own latency:

$$Slack(Packet_i) = \max_k Latency(Packet_{k \; \forall \; k=0 \; to \; Numberof Predecessors}) - Latency(Packet_i) \quad (1)$$

The key challenge in estimating slack is to accurately predict latencies of predecessor packets and the current packet being injected. Unfortunately, it is *difficult to predict network latency exactly in a realistic system.* Our solution is that instead of predicting the exact slack in terms of cycles, we aim to categorize/quantize slack into different priority levels, based on *indirect metrics* that *correlate with the latency of predecessors and the packet being injected*. To accomplish this, we characterize slack as it relates to various indirect metrics.

3.2 Slack Characterization

To characterize slack, we measured criticality of packets in *network stall cycles per packet*, i.e. NST/packet, which is the average number of cycles an application stalls due to an outstanding network packet. We found that the three most important factors impacting packet criticality (hence, slack) are 1) the number of predecessors that are L2 cache misses, 2) whether the injected packet is an L2 cache hit or miss, and 3) the number of a packet's extraneous hops in the network (compared to its predecessors).

Number of miss-predecessors: We define a packet's *miss predecessors* as its predecessors that are L2 cache misses. We found that the number of miss-predecessors of a packet correlates with the criticality of the packet. This is intuitive as the number of miss predecessors tries to capture the *first term of the slack equation*, i.e. Equation (1). Figure 3(a) shows the criticality (NST/packet) of packets with different number of miss-predecessors. Packets with zero miss-predecessors have the highest criticality (NST/packet) and likely the lowest slack because their latencies are unlikely to be overlapped by those of predecessors. Indeed, the difference in significant: packets with zero predecessors have much higher criticality than packets with one or more predecessors.

Figure 3(b) shows the fraction of packets that have a certain number of predecessors. The fraction of packets with more than four predecessors is very low. Thus, a mechanism that uses levels of priority based on the number of miss-predecessors need not contain many levels. Note that, since the number of miss-predecessors captures only the first term of slack equation and ignores the second term (latency of the injected packet), the correlation between it and packet criticality is imperfect; for example, in `Gems`, packets with 4+ miss-predecessors appear to have higher criticality than packets with 0 miss-predecessors. Therefore, we use two additional indirect metrics to take into account the latency of a newly injected packet.

L2 cache hit/miss status: We observe that whether a packet is an L2 cache hit or miss correlates with how critical the packet is, and hence the packet's slack. If the packet is an L2 miss, it likely has high latency due to DRAM access and extra NoC transactions to/from memory controllers. The packet therefore likely has smaller slack as its long latency is less likely to be overlapped by other packets.

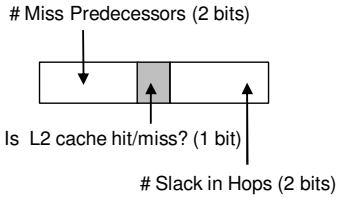

Figure 5: Priority structure for estimating slack of a packet. A lower priority level corresponds to a lower slack value.

Figure 4(a) quantifies the criticality (NST/packet) of packets based on their L2 hit/miss status. Across all applications, L2-miss packets have at least ten times higher NST/packet than L2-hit packets. Note that two priority levels suffice to designate a packet as cache hit or miss.

Slack in terms of number of hops: Our third metric captures both terms of the slack equation, and is based on the *distance* traversed in the network. This metric was illustrated in our motivating example in Section 2.2, where we approximated latency and slack as the number of hops between a source and destination. Specifically:

$$Slack_{hops}(Packet_i) = \max_k Hops(Packet_{k\ \forall k=0\ to\ NumberofPredecessors}) - Hops(Packet_i) \quad (2)$$

Figure 4(b) shows that such hop-based slack indeed correlates with the criticality of packets (NST/packet). Packets with a smaller hop-based slack (0-3 hops) have significantly higher criticality than packets with a larger hop-based slack in almost all applications. For this reason, we use hop-based slack as the third component for determining the slack priority level of a packet.

3.3 Slack Priority Levels

Our analysis shows the above three metrics correlate with slack. We combine these metrics to form the *slack priority level* of a packet in Aérgia. When a packet is being injected, the above three metrics are computed and *quantized* to form a three-tier priority, shown in Figure 5. The head flit of the packet is tagged with these priority bits. (i) We use 2 bits for the first tier, which is assigned based on the number of miss-predecessors. (ii) We use a single bit for the second tier to indicate if the packet being injected is (predicted to be) an L2 hit or miss. (iii) We use 2 bits for the third tier to indicate the hop-based slack of a packet. The combined slack-based priority level is then used to prioritize between packets in routers (see Section 4).

3.4 Slack Estimation

In Aérgia, assignment of a slack priority level to a packet requires the estimation of the above-described three metrics when the packet is injected.

3.4.1 Estimating the number of miss-predecessors

Every core maintains a list of the outstanding L1 load misses (predecessor list). Note that the size of this list is bounded by the size of the miss status handling registers (MSHRs) [20]. Each of these L1 misses is associated with a corresponding *L2-miss-status* bit. At the time a packet is injected into the NoC, its actual L2 hit/miss status is unknown because the shared L2 cache is distributed across the nodes. Therefore, an L2 hit/miss predictor is consulted (as described below) and the *L2-miss-status* bit is set accordingly. The miss-predecessor slack priority level is computed as the number of outstanding L1 misses in the predecessor list whose *L2-miss-status* bits are set (to indicate a predicted or actual L2 miss). Our implementation keeps track of L1 misses issued in the last 32 cycles and sets the maximum number of miss-predecessors to 8.

In case of an error in prediction, the *L2-miss-status* bit is updated to the correct value when the data response packet returns to the core. In addition, if a packet that is predicted to be an L2 hit actually results in an L2 miss, the corresponding L2 cache bank notifies the requesting core that the packet actually resulted in an L2 miss so that the requesting core updates the corresponding *L2-miss-status* bit accordingly. To reduce control packet overhead, the L2 bank piggy-backs this information to another packet traveling through the requesting core as an intermediate node.

3.4.2 Estimating whether a packet will be L2 cache hit or miss

At the time a packet is injected, it is not known whether or not it will hit in the remote L2 cache bank it accesses. We use an L2 hit/miss predictor in each core to guess the cache hit/miss status of each injected packet, and set the second-tier priority bit accordingly in the packet header. If the packet is predicted to be an L2 miss, its L2 miss priority bit is set to 0, otherwise it is set to 1. Note that this priority bit within the packet header is corrected when the actual L2 miss status is known when the packet accesses the L2.

We develop two types of L2 miss predictors. The first is based on the *global branch predictor* [36]. A shift register records the hit/miss values for the last "M" L1 load misses. This register is then used to index into a pattern history table (PHT) containing two-bit saturating counters. The accessed counter indicates whether or not the prediction for that particular pattern history is a hit or a miss. The counter is updated when the hit/miss status of a packet is known. The second predictor, called the *threshold predictor*, uses the insight that misses occur in bursts. This predictor updates a counter if the access is a known L2 miss. The counter is reset after every "M" L1 misses. If the number of known L2 misses in the last "M" L1 misses exceeds a threshold ("T"), then the next L1 miss is predicted to be an L2 miss. The global predictor requires more storage bits (PHT) and has marginally higher design complexity than the threshold predictor. Note that, for correct updates, both predictors require an extra bit in the response packet indicating whether or not the transaction was an L2 miss.

3.4.3 Estimating hop-based slack

The hops per packet for any packet is calculated by adding the X and Y distance between source and destination. The hop-based slack of a packet is then calculated using Equation (2).

4. Aérgia Network-on-Chip Architecture

4.1 Aérgia Routers

We propose a new network-on-chip architecture, *Aérgia*, that utilizes the described slack priority levels arbitration. In this section, we describe the proposed architecture, starting ground-up from the baseline. We also address some design challenges that come along with prioritization mechanisms: i) mitigating priority inversion (using multiple network interface queues) and ii) starvation avoidance (using batching).

4.1.1 Baseline

A generic NoC router architecture is illustrated in Figure 6 (see also [7]). 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. Dimension Ordered Routing (DOR) is the most commonly used routing policy for its low complexity and deadlock freedom. Our baseline assumes XY DOR, where packets are first routed in X direction followed by Y direction. The Virtual channel Arbitration unit (VA) arbitrates amongst all

packets requesting access to the same output VC in the downstream router and decides on winners. The Switch Arbitration unit (SA) arbitrates amongst all input 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. Current routers use simple, local arbitration policies (round-robin, oldest-first) to decide which packet should be scheduled next (i.e., which packet wins arbitration). Our baseline NoC uses the round-robin policy.

4.1.2 Arbitration

Slack Priority Levels: In Aérgia, the Virtual channel Arbitration (VA) and Switch Arbitration (SA) units prioritize packets with lower slack priority levels. Thus, low-slack packets get the first preference for buffers as well as the crossbar, and hence, are accelerated in the network. Since in wormhole switching, only the head flit arbitrates for and reserves the VC, the slack priority value is carried by the head flit only. This header information is utilized during VA for priority arbitration. In addition to the state maintained by the baseline architecture, each virtual channel has one additional priority field, which is updated with the slack priority level of the head flit when the head flit reserves the virtual channel. This field is utilized by the body flits during SA for priority arbitration.

Batching: Without adequate counter-measures, prioritization mechanisms can easily lead to starvation in the network. In order to prevent starvation, we combine our slack-based prioritization with a "batching" mechanism similar to [8]. Time is divided into intervals of T cycles, called *batching intervals*. Packets inserted into the network during the same interval belong to the same batch, i.e. have the same *batch priority* value. Packets belonging to older batches are prioritized over those from younger batches. Only if two packets belong to the same batch, they are prioritized based on their available slack, i.e. the slack priority levels. Each head flit carries a 5-bit slack priority value and a 3-bit batch number/priority, as shown in Figure 7. We use adder delays in [27] to estimate the delay of an 8-bit priority arbiter (P=5,V=6) to be 138.9 picoseconds and a 16-bit priority arbiter to be 186.9 picoseconds at 45nm technology.

4.1.3 Network Interface

For effectiveness, prioritization is necessary not only within the routers but also at the network interfaces. We split the monolithic injection queue at a network interface into a small number of equal-length queues. Each queue buffers packets with slack priority levels within a different programmable range. Packets are guided into a particular queue based on their slack priority levels. Queues belonging to higher priority packets are prioritized over those of lower priority packets. Such prioritization reduces priority inversion and is especially needed at memory controllers where packets from all applications buffered together at the network interface, and thus, monolithic queues can cause severe priority inversion.

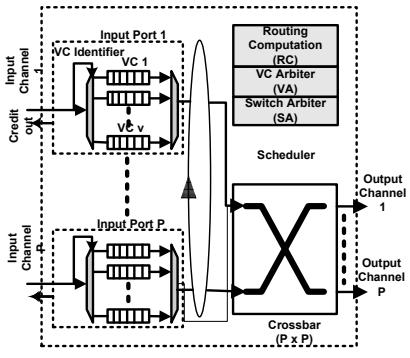

Figure 6: Baseline router microarchitecture

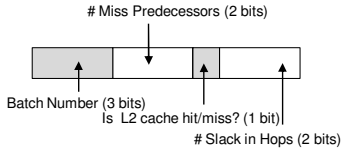

Figure 7: Final Aérgia priority structure

5. Qualitative Comparison to Application Aware Scheduling and Fairness Policies

Application-Aware Prioritization Mechanism (STC): Das et al. [8] proposed an application-aware coordinated arbitration policy (called STC) to accelerate network-sensitive applications. The key idea is to rank applications at regular intervals based on their network intensity (L1 miss rate per instruction, L1-MPI), and prioritize *all packets* of a non-intensive application over *all packets* of an intensive application. Within applications belonging to the same rank, packets are prioritized using the baseline round-robin order in a slack-unaware manner. Packet batching is used for starvation avoidance. The concept of STC and slack are complementary. There are several important differences between Aérgia and STC:

• Aérgia exploits slack at the fine granularity of *individual packets*. In contrast, STC is a coarse-grained approach that identifies critical applications (from the NoC perspective) and prioritizes them over non-critical ones. An application as a whole might be less critical than another, but even within an application there could be packets that impact its performance more than others. By focusing on application-level intensity, STC cannot capture this fine grained packet-level criticality that is exploited by a slack-aware mechanism like Aérgia.

• STC uses the L1-MPI of each application to differentiate between applications over a long ranking interval. This coarse-grained metric of application intensity does not capture finer-grained distinctions between applications/packets in both time and space. For example, this heuristic does not distinguish between applications that traverse a large vs. small number of hops. Two applications might have the same L1 miss rate but one of them might have higher intensity in the network because it either traverses longer distances or has much smaller L2 miss rate, thereby putting more pressure on the network. STC is unable to distinguish between such applications whereas Aérgia's packet based slack estimation takes into account these factors, thereby prioritizing applications/packets based on different aspects of criticality.

Note that STC and Aérgia are complementary and can be combined to take advantage of both algorithms' strengths. In this paper, we propose and evaluate such a combination. The combined architecture prioritizes higher ranked applications over lower ranked applications as STC does. Within the same ranking class, however, it uses slack-based prioritization, as Aérgia does. Our results show that the combination provides better performance and fairness than STC and Aérgia alone.

Globally Synchronized Frames (GSF): Since any prioritization mechanism is likely to impact the fairness of the system, we quantitatively compare Aérgia to a state-of-art fairness scheme, Globally Synchronized Frames (GSF) [21]. GSF provides prioritization mechanisms within the network to ensure 1) guarantees on minimum bandwidth and minimum network delay each application experiences, and 2) that each application achieves equal network throughput. To accomplish this, GSF employs the notion of frames, which

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, 260 cycle access, 4 on-chip Memory Controllers (one in each corner node).
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) 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 (NSTP): Average network stall-time per packet, Inj Rate: Average packets per 100 Instructions, Load: low/high depending on injection rate, NSTP: high/low, Bursty: high/low.

is similar to our concept of batches. Within a frame, GSF does not distinguish between different applications; in fact, it does not specify a prioritization policy. In contrast, Aérgia employs a slack-aware prioritization policy within packets of the same batch. As a result, Aérgia provides better system-level throughput by prioritizing those packets that would benefit more from network service (as shown in our evaluation). Note that the goals of GSF and Aérgia are different: GSF aims to provide network-level QoS, whereas Aérgia optimizes system throughput by exploiting the notion of packet slack.

6. Methodology

6.1 Experimental Setup

We evaluate our techniques using a trace-driven, cycle-level 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 a private write-back L1 cache. 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 (Aérgia) as well as previously proposed STC and GSF. 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 packets. We use the threshold predictor for Aérgia with M=4 and T=2.

6.2 Evaluation Metrics

We evaluate our proposal using several metrics. We measure **application-level system performance** in terms of weighted and harmonic speedup [12], 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 [8]. 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:

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

6.3 Application Categories and Characteristics

We use a diverse set of multiprogrammed application workloads comprising scientific, commercial, and desktop applications. In total, we study 35 applications, including SPEC CPU2006 benchmarks, applications from SPLASH-2 and SpecOMP benchmark suites, and four commercial workloads traces (`sap`, `tpcw`, `sjbb`, `sjas`). We choose representative execution phases using [28] for all our workloads excluding commercial traces, which were collected over Intel servers. To have tractable simulation time, we choose a smaller

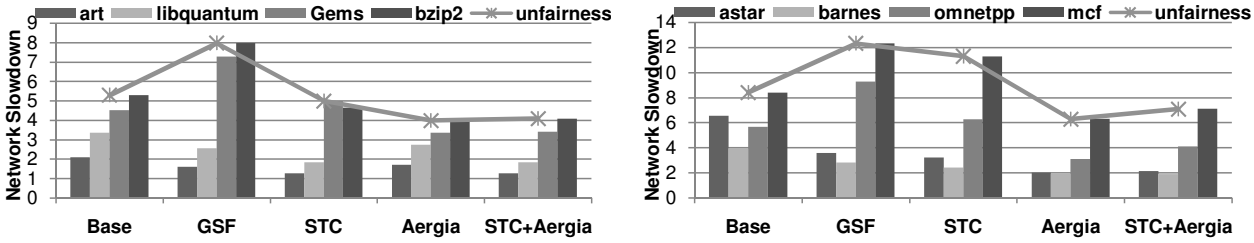

Figure 8: Network Slowdowns for (a) Case Study I on the left (b) Case Study II on the right

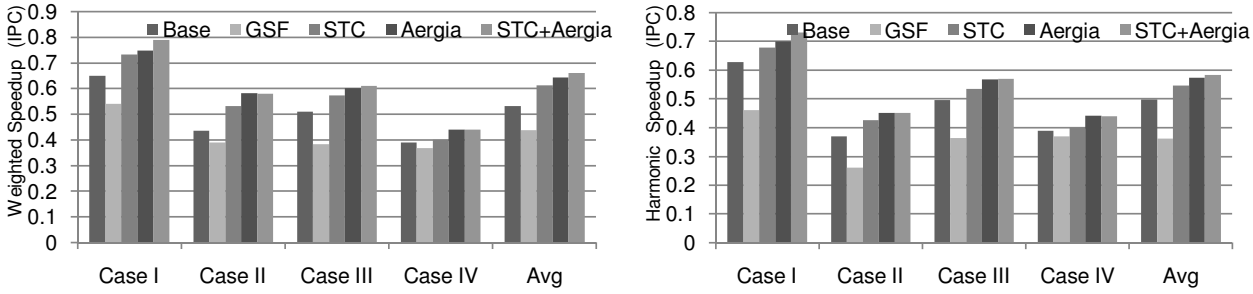

Figure 9: (a) Weighted Speedup for all Case Studies (b) Harmonic Speedup for all Case Studies

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 56 different workload combinations.

7. Performance Evaluation

We first compare Aérgia to the baseline, GSF, and STC, using four case studies to provide insight into the behavior of each scheme with different types of workloads. Figures 8 and 10 show the network slowdowns (the lower the better) of the individual applications. Figure 9 shows system performance (weighted speedup[4] and harmonic speedups) of the different schemes for all case studies. Section 7.5 reports aggregate results averaged over 56 different workloads, showing that the benefits of our scheme hold over a wide variety of workloads. Finally, Section 7.6 examines the effect of L2 miss predictor accuracy on system performance.

7.1 Case Study I: Heterogenous mix of heavy latency-tolerant and network-sensitive applications

We mix 16 copies each of two heavy applications (`GemsFDTD` and `bzip2`) and two network-sensitive applications (`libquantum` and `art`); and compare Aérgia to baseline, GSF and STC. We make several observations:

• Figure 8 (a) shows the network slowdowns of applications. The baseline algorithm slows down applications in hard-to-predict ways as local round-robin arbitration is application-oblivious. `bzip2` has the highest slowdown (and hence determines the unfairness index) due to heavy demand from the network and high L2 cache miss rate.

• GSF unfairly penalizes the heavy applications because they quickly run out of frames and stop injecting. Thus, the network slowdowns of `GemsFDTD` and `bzip2` increase by 1.6X and 1.5X over baseline. Since GSF guarantees minimum bandwidth to all applications, it improves the network slowdown of network-sensitive applications over the baseline (by 21% for `libquantum` and 26% for `art`) 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 reduction of network-sensitive applications' slowdowns. Overall, GSF degrades system throughput by 16.6% and application-level fairness by 50.9% because it unfairly penalizes the heavy applications.

• STC prioritizes the network-sensitive applications using ranking, and ensures, using batching, that heavy applications are not overly penalized. The result is that it significantly reduces the slowdown of network-sensitive applications (by 1.9X for `libquantum` and 1.6X for `art`). Heavy applications suffer marginally from lower ranking because 1) of batching 2) they are latency tolerant, i.e. have sufficient slack to tolerate de-prioritization in the network (See Figure 1 (b)), and 3) sensitive applications are light and hence negatively affect a small fraction of packets of heavy applications. STC improves weighted speedup by 12.6% over baseline.

• Aérgia exploits slack and criticality at the fine granularity of individual packets and thus optimizes for instantaneous behavior of applications. Unlike STC, which improves only network-sensitive applications, **Aérgia improves the network slowdowns of all applications** (by 24% for `libquantum`, 21% for `art`, 26% for `GemsFDTD` and 35% for `bzip2` over the baseline). *This leads to a weighted speedup improvement of 15.1% over the baseline. By improving all applications, Aérgia reduces the maximum network slowdown, and thus has the best network fairness (32% over baseline and 25% over STC).* The best improvements are seen for `bzip2` for two reasons. First, although a heavy application, at least 30% `bzip2`'s packets are critical with fewer than 50 slack cycles. These critical packets are deprioritized by STC as well as the baseline because neither is aware of the slack of these packets and they both delay bzip2 simply because it is heavy. Second, `bzip2` has better performance than `GemsFDTD` because it has a higher fraction of critical packets than `GemsFDTD`, which are prioritized by using the available slack in all applications.

• As discussed in Section 5, Aérgia and STC are complementary techniques that exploit slack and criticality at different granularity. Thus, we evaluate a combination of both architectures (referred to as STC+Aérgia in figures). While STC prioritizes the light, network-

[4]Weighted speedup is divided by number of cores (=64) in results charts for clarity.

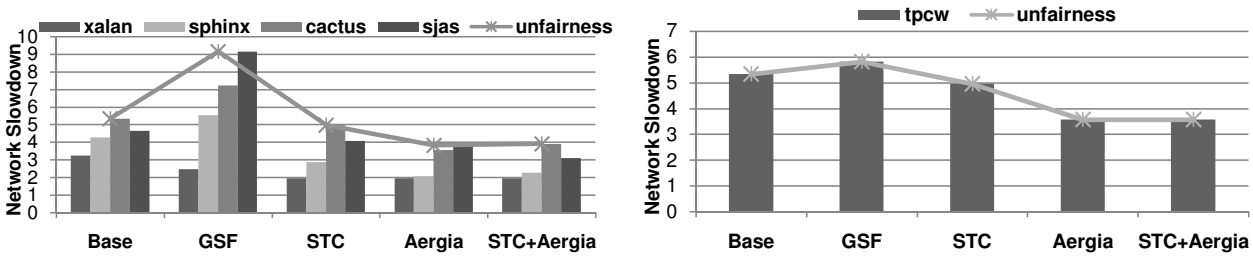

Figure 10: Network Slowdowns for (a) Case Study III on the left (b) Case Study IV on the right

sensitive applications effectively, Aérgia can exploit the differential slack within the packets of an application to improve the performance of heavy applications as well as light ones. As a result, the combination of both techniques provides the best performance (21.5% over baseline, 5.6% over Aérgia and 7.7% over STC) and almost similar fairness to Aérgia (23% and 19% better fairness over baseline and STC).

7.2 Case Study II: Heterogenous mix of heavy critical applications and light applications

We mix 16 copies each of two heavy applications (`omnetpp` and `mcf`) and two light applications (`astar` and `barnes`). The case study shows a scenario in which fairness is a problem with STC, while Aérgia is effective at prioritizing light applications that have high-slack packets. The following observations are in order:

• The baseline and GSF have similar trends as discussed in Case Study I. Although STC still provides 21.9% speedup by prioritizing the light applications (network slowdown of `astar`/`barnes` improves 2.0X/1.6X over baseline in STC), it degrades the fairness of the system by penalizing the heavy applications in this case study (STC increases unfairness by 1.35X over baseline). Some packets of these applications have high criticality (i.e. low slack), but STC deprioritizes all such packets because the applications they belong to are network-intensive.

These heavy applications have high criticality due to their high L2 miss rates, but STC deprioritizes them because they are network-intensive.

• As opposed to Case Study I, in this case study, Aérgia is more effective than STC in prioritizing light applications.[5] This result shows that to prioritize a light application, it is not necessary to prioritize *all* packets of that application as STC does. A prioritization mechanism that prioritizes only the critical/low-slack packets (like Aérgia) can provide even better speedups for a light application than a prioritization mechanism that prioritizes all packets of that light application over packets of other applications (like STC). This is because a finer-grained packet based prioritization mechanism can actually prioritize critical packets over the non-critical ones *within* the light application. `astar` and `barnes` have a small fraction of critical packets, which are prioritized over others with Aérgia but not distinguished from non-critical ones with STC. Aérgia also reduces slowdown of heavy applications because it can prioritize critical packets of such applications over non-critical packets. As a result, Aérgia improves weighted speedup by 33.6% and 9.4% over the baseline and STC, respectively.

• STC+Aérgia solves the fairness problem in STC because it reduces the network slowdown of heavy applications by prioritizing the important packets of such applications over less important packets of the same applications. Hence, it reduces the penalty the heavy applications have to pay for prioritizing the light applications in the original STC scheme. Overall STC+Aérgia provides 33.2% speedup over baseline, while reducing unfairness by 1.6X over STC (1.2X over baseline).

[5]Network slowdown of `astar`/`barnes` improves 3.2X/2.0X over baseline in Aérgia.

7.3 Case Study III: Heterogenous mix of heavy applications

In [8], authors demonstrate a case study 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. We run 16 copies each of four heavy applications (`xalan`, `sphinx`, `cactus`, and `sjas`) to demonstrate a similar scenario. Figure 10 (a) shows the network slowdowns for this case study. Aérgia provides better performance than all schemes for such workloads (18.1% weighted speedup over baseline), while providing the best fairness (26% over baseline). Aérgia is more effective in reducing the slowdowns of `sphinx` and `cactus` than STC by exploiting criticality of packets at much finer levels. Since Aérgia prioritizes based on the slack at the packet level, it can easily distinguish between the packets of different applications and prioritize the critical ones even though the intensities of applications are similar.

7.4 Case Study IV: Homogeneous mix with multiple copies of the same application

The purpose of this homogenous case study is to demonstrate effectiveness of Aérgia in a scenario when STC can provide minimal performance gains. We run *a copy of the same application, `tpcw` on all 64 nodes*. Figure 10 (b) shows the network slowdowns. STC treats all cores the same, i.e. at the same priority level, because the L1-MPIs of the cores are similar. Batching framework of STC provides 2.8% higher weighted speedup over baseline by reducing starvation that occurs in the baseline. In contrast, Aérgia can provide attractive performance gains (13.7% weighted speedup over baseline) even for such homogenous workloads. This is because, even though applications and their network intensities are the same, individual packet criticality is different. Hence, Aérgia takes advantage of the disparity in packet criticality within the same application class to provide better performance and fairness.

7.5 Overall Results Across 56 Multi-Programmed Workloads

Figures 11 (a) and (b) compare all the schemes averaged across 56 workload mixes. The mixes consist of 12 homogenous workloads and 48 heterogenous workloads with applications picked randomly from different categories (See Table 2). The aggregate results are consistent with the observations made in the three case studies. Aérgia improves average system throughput by 10.3%/11.6% (weighted/ harmonic) compared to the baseline, while also improving network fairness by 30.8%. Aérgia combined with STC improves average system throughput by 6.5%/5.2% (weighted/ harmonic) compared to STC alone while also improving network fairness by 18.1%. We conclude that Aérgia provides the best system performance and network fairness over a very wide variety of work-

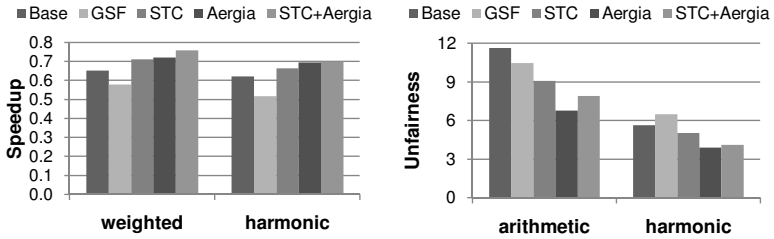

Figure 11: Aggregate results across 56 workloads (a) System Speedup (b) Network Unfairness

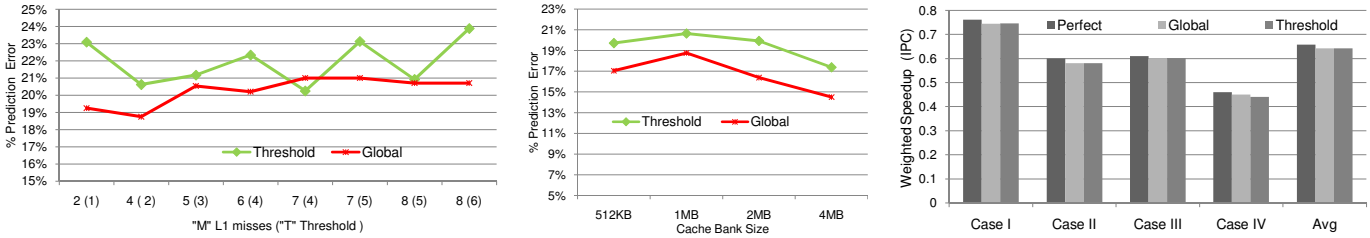

Figure 12: Analysis of L2 miss predictors: (a) error with parameters, (b) error with cache size, (c) effect of perfect prediction

loads whether used independently or in conjunction with application-aware prioritization techniques.

7.6 Effect of L2 Cache Miss Predictor

Figure 12 (a) shows the sensitivity of prediction accuracy to the parameters of the two predictors we developed: M (number of accesses in history) and T (threshold value, which is used only for the threshold predictor). Figure 12 (b) shows the sensitivity of prediction accuracy to size of cache banks for both predictors. Averaged over 56 workloads for 1MB L2 cache banks, the global predictor has an error of 18.7% (M=4) and the threshold predictor has an error of 20.7% (M=4, T=2). Figure 12 (c) shows the performance difference between perfect miss prediction, the global predictor, and the threshold predictor. Results show that only 2.4% performance improvement is possible if the predictor were perfect. This is mainly because the negative impact of a misprediction is limited because we update the L2 cache hit/miss priority field of a packet once the miss status is known after L2 access.[6] We conclude that the miss predictors we develop provide sufficient accuracy for slack estimation purposes.

8. Related Work

To our knowledge, no previous work characterized slack in packet latency and proposed mechanisms to exploit it for on-chip network arbitration. We briefly describe the most closely related previous work.

Instruction Criticality and Memory Level Parallelism: A large amount of research has been done to predict criticality/slack of instructions [32, 14, 13, 33] and prioritize critical instructions in the processor core and caches. MLP-aware cache replacement exploits cache miss criticality differences due to memory-level parallelism [29]. Parallelism-aware memory scheduling [24] and other rank-based memory scheduling algorithms [19] reduce application stall-time by improving bank-level parallelism of each thread. These works are related to ours only in the sense that we also exploit criticality to improve system performance. However, our methods (for computing slack) and mechanisms (for exploiting it) are very different due to the distributed nature of on-chip networks.

Prioritization in On-Chip and Multiprocessor Networks: We have already compared our approach extensively, both to state-of-the-art local arbitration, QoS-oriented prioritization (GSF [21]), and application-aware prioritization (STC [8]) policies in NoCs. Bolotin et al. [3] propose prioritizing control packets over data packets in the NoC, but do not distinguish packets based on available slack. Frameworks for QoS [2, 31, 17] on NoCs were proposed, which can possibly be combined with our approach. Arbitration policies [37, 6, 15, 1, 9, 38] have also been proposed in multi-chip multiprocessor networks and long-haul networks. Their goal is to provide guaranteed service or fairness, while ours is to exploit slack in packet latency to improve system performance. In addition, most of the above previous mechanisms *statically assign priorities/bandwidth* to different flows in off-chip networks to satisfy real-time performance and QoS guarantees. In contrast, our proposal *dynamically computes and assigns priorities to packets* based on slack.

Batching: We use packet batching in NoC for starvation avoidance, similarly to [8]. The idea of batching/frames has been used in disk scheduling [34], memory scheduling [24], and QoS-aware packet scheduling [21, 17] to prevent starvation and provide fairness.

[6]Thus, the priority is corrected for data response packets, which have much higher latency than request packets. Also, request packets to on-chip DRAM controller have the corrected priority.

9. Conclusion

The network-on-chip is a critical shared resource likely to be shared by diverse applications with varying characteristics and service demands. Thus, characterizing, understanding, and optimizing the interference behavior of applications in the NoC is an important problem for enhancing system performance and fairness. Towards this end, we propose fine-grained prioritization mechanisms to improve the performance and fairness of NoCs. We introduce the concept of *packet slack* and characterize it in the context of on-chip networks. In order to exploit packet slack, we propose and evaluate a novel architecture, called Aérgia, which identifies and prioritizes low-slack (critical) packets. The key components of the proposed architecture are techniques for online estimation of slack in packet latency and slack-aware arbitration policies. Averaged over 56 randomly-generated multiprogrammed workload mixes on a 64-core 8x8-mesh CMP, Aérgia improves overall system throughput by 10.3% and network fairness by 30.8% on average. Our results show that Aérgia outperforms three state-of-the-art NoC scheduling/prioritization/fairness techniques. We conclude that the proposed network architecture is effective at improving overall system throughput as well as reducing application-level network unfairness.

Acknowledgments

This research is supported in part by NSF grant CCF 0702519, CMU CyLab, and GSRC. We thank Satish Narayanasamy, Soumya Eachempati and the anonymous reviewers for discussions and 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] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA-11*, 2005.

[5] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS-21*, 2007.

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

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

[8] R. Das, O. Mutlu, T. Moscibroda, and C. Das. Application-Aware Prioritization Mechanisms for On-Chip Networks. In *MICRO-42*, 2009.

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

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

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ASPLOS-XV*, 2010.

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

[13] B. Fields, R. Bodík, and M. Hill. Slack: Maximizing performance under technological constraints. In *ISCA-29*, 2002.

[14] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *ISCA-28*, 2001.

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

[16] 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.

[17] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip. In *MICRO-42*, 2009.

[18] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *PACT-15*, 2006.

[19] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA-16*, 2010.

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

[21] 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.

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

[23] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

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

[25] 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.

[26] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.

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

[28] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *MICRO-37*, 2004.

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

[30] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO-39*, 2006.

[31] 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.

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

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

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

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

[36] T. Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO-24*, 1991.

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

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