AÉRGIA: A NETWORK-ON-CHIP EXPLOITING PACKET LATENCY SLACK

A TRADITIONAL NETWORK-ON-CHIP (NOC) EMPLOYS SIMPLE ARBITRATION STRATEGIES, SUCH AS ROUND ROBIN OR OLDEST FIRST, WHICH TREAT PACKETS EQUALLY REGARDLESS OF THE SOURCE APPLICATIONS' CHARACTERISTICS. THIS IS SUBOPTIMAL BECAUSE PACKETS CAN HAVE DIFFERENT EFFECTS ON SYSTEM PERFORMANCE. WE DEFINE SLACK AS A KEY MEASURE FOR CHARACTERIZING A PACKET'S RELATIVE IMPORTANCE. AÉRGIA INTRODUCES NEW ROUTER PRIORITIZATION POLICIES THAT EXPLOIT INTERFERING PACKETS' AVAILABLE SLACK TO IMPROVE OVERALL SYSTEM PERFORMANCE AND FAIRNESS.

Reetuparna Das
Pennsylvania State University

Onur Mutlu
Carnegie Mellon University

Thomas Moscibroda
Microsoft Research

Chita R. Das
Pennsylvania State University

Network-on-Chips (NoCs) are widely viewed as the de facto solution for integrating many components that will comprise future microprocessors. It's therefore foreseeable that on-chip networks will become a critical shared resource in many-core systems, and that such systems' performance will depend heavily on the on-chip networks' resource sharing policies. Devising efficient and fair scheduling strategies is particularly important (but also challenging) when diverse applications with potentially different requirements share the network.

An algorithmic question governing application interactions in the network is the NoC router's arbitration policy—that is, which packet to prioritize if two or more packets arrive at the same time and want to take the same output port. Traditionally, on-chip network router arbitration policies have been simple heuristics such as round-robin and age-based (oldest first) arbitration. These arbitration policies treat all packets equally, irrespective of a packet's underlying application's 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 have unique and dynamic characteristics and demands. Different packets can have a significantly different importance to their applications' performance. In the presence of memory-level parallelism (MLP),[1,2] although the system might have multiple outstanding load misses, not every load miss causes a bottleneck (or is critical).[3] Assume, for example, that an application issues two concurrent network requests, one after another, first to a remote node in the network and then to a close-by node. Clearly, the packet going to the close-by node is less critical; even if it's delayed for several cycles in the network, its latency will be hidden from the application by the packet going to the distant node, which would take more time. Thus, each packet's delay tolerance can have a different impact on its application's performance.

In our paper for the 37th Annual International Symposium on Computer Architecture (ISCA),[4] we exploit packet diversity in criticality to design higher performance and more application-fair NoCs. We do so by differentiating packets according to their slack, a measure that captures the packet's importance to the application's performance. In particular, we define a packet's slack as the number of cycles the packet can be delayed in the network without affecting the application's execution time. So, a packet is relatively noncritical until its time in the network exceeds the packet's available slack. In comparison, increasing the latency of packets with no available slack (by deprioritizing them during arbitration in NoC routers) will stall the application.

Building off of this concept, we develop Aérgia, an 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. (We name our architecture after Aérgia, the female spirit of laziness in Greek mythology, after observing that some packets in NoC can afford to slack off.) We devise techniques to efficiently estimate a packet's slack dynamically. Before a packet enters the network, Aérgia tags the packet according to its estimated slack. We propose extensions to routers to prioritize lower-slack packets at the contention points within the router (that is, buffer allocation and switch arbitration). Finally, to ensure forward progress and starvation freedom from prioritization, Aérgia groups packets into batches and ensures that all earlier-batch packets are serviced before later-batch packets. Experimental evaluations on a cycle-accurate simulator show that our proposal effectively increases overall system throughput and application-level fairness in the NoC.

Motivation

The premise of our research is the existence of packet latency slack. We thus motivate Aérgia by explaining the concept of slack, characterizing diversity of slack in applications, and illustrating the advantages of exploiting slack with a conceptual example.

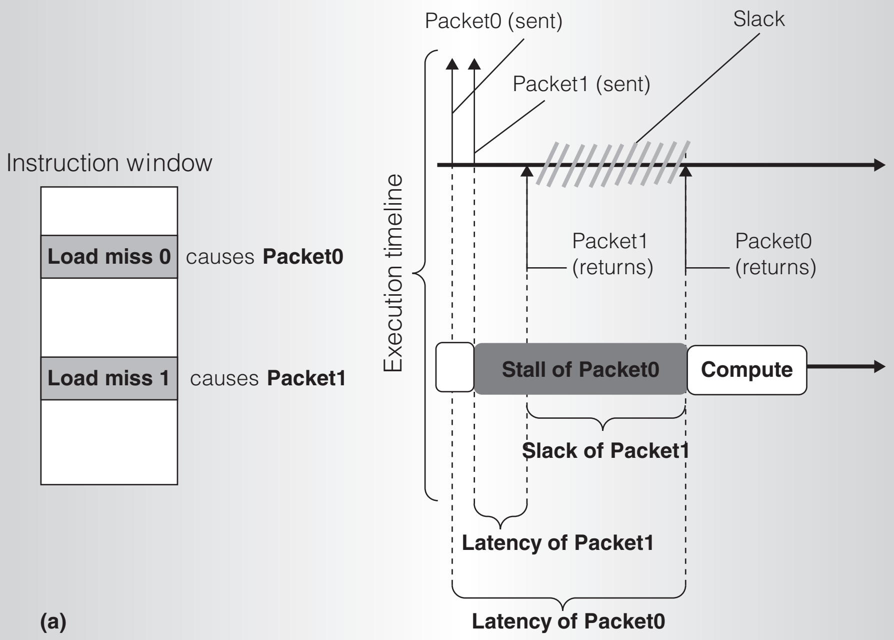
The concept of slack

Modern microprocessors employ several memory latency tolerance techniques (such as out-of-order execution[5] and runahead execution[2,6]) to hide the penalty of load misses. These techniques exploit MLP[1] by issuing several memory requests in parallel with the hope of overlapping future load misses with current load misses. In the presence of MLP in an on-chip network, the existence of multiple outstanding packets leads to packet latency overlap, which introduces slack cycles (the number of cycles a packet can be delayed without significantly affecting performance). We illustrate the concept of slack cycles with an example. Consider the processor execution timeline in Figure 1a. 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 Figure 1a's execution timeline, Packet1 has lower network latency than Packet0 and returns to the source earlier. Nonetheless, the processor can't commit Load0 and stalls until Packet0 returns. Thus, Packet0 is the bottleneck packet, which allows the earlier-returning Packet1 some slack cycles. The system could delay Packet1 for the slack cycles without causing significant application-level performance loss.

Diversity in slack and its analysis

Sufficient diversity in interfering packets' slack cycles is necessary to exploit slack. In other words, it's only possible to benefit from prioritizing low-slack packets if the network has a good mix of both high- and low-slack packets at any time. Fortunately, we found that realistic systems and applications have sufficient slack diversity.

Figure 1b shows a cumulative distribution function of the diversity in slack cycles for 16 applications. The *x*-axis shows the number of slack cycles per packet. 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 the number of slack cycles, indicating sufficient slack diversity within an application. For example, 17.6 percent of `Gems`'s packets have, at most, 100 slack cycles, but 50.4 percent of them have more than 350 slack cycles. Second, applications have different slack characteristics. For example, the packets of `art`



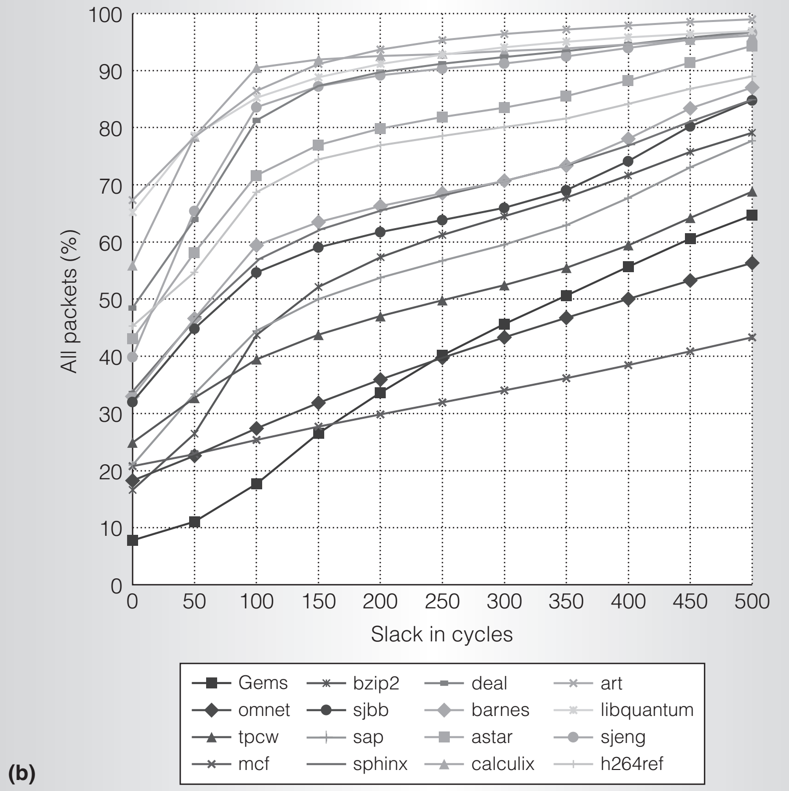


Figure 1. Processor execution timeline demonstrating the concept of slack (a); the resulting packet distribution of 16 applications based on slack cycles (b).

and `libquantum` have much lower slack in general than `tpcw` and `omnetpp`. We conclude that each packet's number of slack cycles varies within application phases as well as between applications.

Advantages of exploiting slack

We show that a NoC architecture that's aware of slack can lead to better system performance and fairness. As we mentioned earlier, a packet's number of slack cycles indicates that packet's importance or criticality to the processor core. If the NoC knew about a packet's remaining slack, it could make arbitration decisions that would accelerate packets with a small slack, which would improve the system's overall performance.

Figure 2 shows a motivating example. Figure 2a depicts 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 8 × 8 mesh (see Figure 2b). Core A generates two packets (A-0 and A-1). The first packet (A-0) isn't preceded by any other packet and is sent to node (8, 8)—therefore, its latency is 13 hops and its slack is 0 hops. In the next cycle, the second packet (A-1) is injected toward node (3, 1). This packet has a latency of 3 hops, and because it's preceded (and thus overlapped) by the 13-hop packet (A-0), it has a slack of 10 hops (13 hops minus 3 hops). Core B also generates two packets (B-0 and B-1). We can calculate Core B's packets' latency and slack similarly. B-0 has a latency of 10 hops and slack of 0 hops, while B-1 has a latency of 4 hops and slack of 6 hops.

We can exploit slack when interference exists between packets with different slack values. In Figure 2b, for example, packets A-0 and B-1 interfere. The critical question is which packet the router should prioritize and which it should queue (delay).

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

Online estimation of slack

Our goal is to identify critical packets with low slack and accelerate them by deprioritizing packets with high-slack cycles.

Defining slack

We can define slack locally or globally.[7] *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. Computing a packet's global slack requires critical path analysis of the entire program, rendering it impractical in our context. So, we focus on local slack. A packet's ideal local slack could depend on instruction-level dependencies, which are hard to capture in the NoC. So, to keep the implementation in the NoC simple, we conservatively consider slack only with respect to outstanding network transactions. We define the term *predecessor packets*, or simply predecessors for a given packet *P*, as all packets that are still outstanding and that have been injected by the same core into the network earlier than *P*. We formally define a packet's available local network slack (in this article, simply called "slack") as the difference between the maximum latency of its predecessor packets (that is, any outstanding packet that was injected into the network earlier than this packet) and its own latency:

$$Slack(Packet_i) = \max_k Latency(Packet_{k\,\forall k=0\text{ to }NumberofPredecessors}) - Latency(Packet_i) \quad (1)$$

The key challenge in estimating slack is to accurately predict latencies of predecessor packets and the current packet being injected. Unfortunately, it's difficult to exactly predict network latency in a realistic system. So, instead of predicting the exact slack in terms of cycles, we aim to categorize

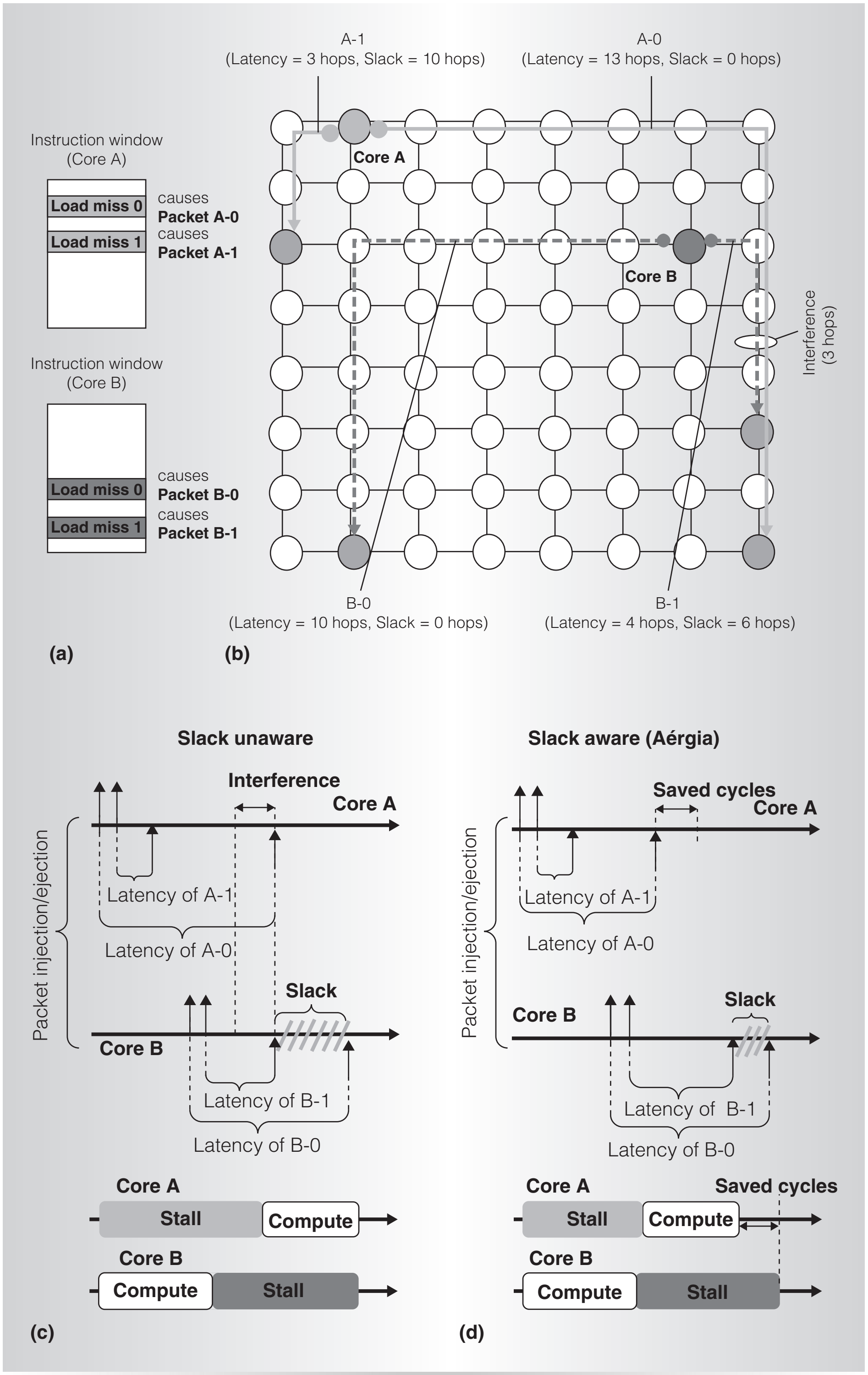


Figure 2. Conceptual example showing the advantage of incorporating slack into prioritization decisions in the Network-on-Chip (NoC). The instruction window for two processor cores (a); interference between two packets with different slack values (b); the processor cores' slack-unaware execution timeline (c); prioritization of the packets according to slack (d).

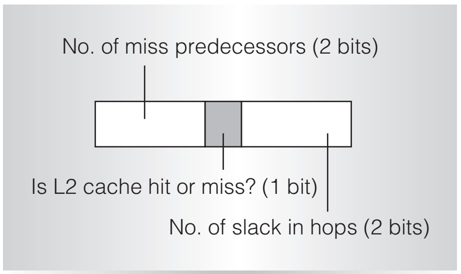

No. of miss predecessors (2 bits)
Is L2 cache hit or miss? (1 bit)
No. of slack in hops (2 bits)

Figure 3. Priority structure for estimating a packet's slack. A lower priority level corresponds to a lower slack value.

and quantize slack into different priority levels according to 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. We found that the most important factors impacting packet criticality (hence, slack) are the number of predecessors that are level-two (L2) cache misses, whether the injected packet is an L2 cache hit or miss, and 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 a packet is likely to have high slack if it has high-latency predecessors or a large number of predecessors; in both cases, there's a high likelihood that its latency is overlapped (that is, that its slack is high). This is intuitive because the number of miss predecessors tries to capture the first term of the slack equation (that is, Equation 1).

L2 cache hit/miss status. We observe that whether a packet is an L2 cache hit or miss correlates with the packet's criticality and, thus, the packet's slack. If the packet is an L2 miss, it likely has high latency owing to dynamic RAM (DRAM) access and extra NoC transactions to and from memory controllers. Therefore, the packet likely has a smaller slack because other packets are less likely to overlap its long latency.

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. Specifically:

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

Slack priority levels

We combine these three metrics to form the slack priority level of a packet in Aérgia. When a packet is injected, Aérgia computes these three metrics and quantizes them to form a three-tier priority (see Figure 3). Aérgia tags the packet's head flit with these priority bits. We use 2 bits for the first tier, which is assigned according to the number of miss predecessors. We use 1 bit for the second tier to indicate if the packet being injected is (predicted to be) an L2 hit or miss. And we use 2 bits for the third tier to indicate a packet's hop-based slack. We then use the combined slack-based priority level to prioritize between packets in routers.

Slack estimation

In Aérgia, assigning a packet's slack priority level requires estimation of the three metrics when the packet is injected.

Estimating the number of miss predecessors. Every core maintains a list of the outstanding L1 load misses (predecessor list). The miss status handling registers (MSHRs) limit the predecessor list's size.[8] Each L1 miss 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 and sets the L2 miss status bit accordingly. We compute the miss predecessor slack priority level 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 records L1 misses issued in the last 32 cycles and sets the maximum number of miss predecessors to 8. If a prediction error occurs, 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.

Estimating whether a packet will be an L2 cache hit or miss. At the time a packet is injected, it's unknown whether it will hit in the remote L2 cache bank it accesses. We use an L2 hit/miss predictor in each core to guess each injected packet's cache hit/miss status, and we 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. This priority bit within the packet header is corrected when the actual L2 miss status is known after the packet accesses the L2.

We developed two types of L2 miss predictors. The first is based on the *global branch predictor*.[9] A shift register records the hit/miss values for the last M L1 load misses. We then use this register to index a pattern history table (PHT) containing 2-bit saturating counters. The accessed counter indicates whether the prediction for that particular pattern history was a hit or a miss. The counter is updated when a packet's hit/miss status 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 resets after every M L1 misses. If the number of known L2 misses in the last M L1 misses exceeds a threshold T, the next L1 miss is predicted to be an L2 miss.

The global predictor requires more storage bits (for the PHT) and has marginally higher design complexity than the threshold predictor. For correct updates, both predictors require an extra bit in the response packet indicating whether the transaction was an L2 miss.

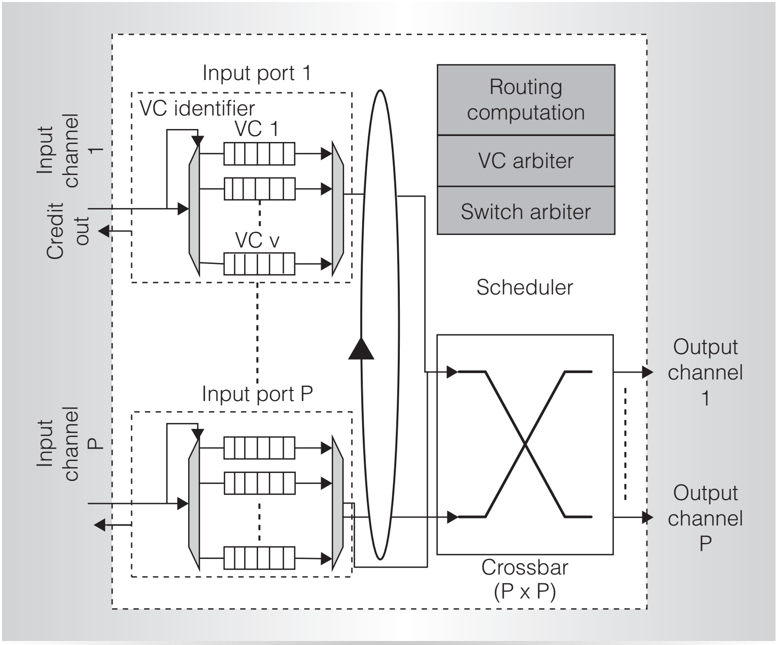


Figure 4. Baseline router microarchitecture. The figure shows the different parts of the router: input channels connected to the router ports; buffers within ports organized as virtual channels; the control logic, which performs routing and arbitration; and the crossbar connected to the output channels.

Estimating hop-based slack. We calculate any packet's hops per packet by adding the X and Y distance between source and destination. We then calculate a packet's hop-based slack using Equation 2.

Aérgia NoC architecture

Our NoC architecture, Aérgia, uses the slack priority levels that we described for arbitration. We now describe the architecture. Some design challenges accompany the prioritization mechanisms, including starvation avoidance (using batching) and mitigating priority inversion (using multiple network interface queues).

Baseline

Figure 4 shows a generic NoC router architecture. The router has P input channels connected to P ports and P output channels connected to the crossbar; typically $P = 5$ for a 2D mesh (one from each direction and one from the network interface). The routing computation unit determines the next router and the virtual channel (V) within the next router for each packet. Dimension-ordered routing (DOR) is the most common routing policy because of its low complexity and deadlock freedom. Our baseline assumes *XY* DOR, which first routes packets in *X* direction, followed by *Y* direction. The virtual channel arbitration unit arbitrates among all packets requesting access to the same output virtual channel in the downstream router and decides on winners. The switch arbitration unit arbitrates among all input virtual channels requesting access to the crossbar and grants permission to the winning packets and flits. The winners can then traverse the crossbar and be placed on the output links. Current routers use simple, local arbitration policies (such as round-robin or oldest-first arbitration) to decide which packet to schedule next (that is, which packet wins arbitration). Our baseline NoC uses the round-robin policy.

Arbitration

In Aérgia, the virtual channel arbitration and switch arbitration units prioritize packets with lower slack priority levels. Thus, low-slack packets get first preference for buffers as well as the crossbar and, thus, are accelerated in the network. In wormhole switching, only the head flit arbitrates for and reserves the virtual channel; thus, only the head flit carries the slack priority value. Aérgia uses this header information during virtual channel arbitration stage for priority arbitration. In addition to the state that the baseline architecture maintains, each virtual channel has an additional priority field, which is updated with the head flit's slack priority level when the head flit reserves the virtual channel. The body flits use this field during switch arbitration stage for priority arbitration.

Without adequate countermeasures, prioritization mechanisms can easily lead to starvation in the network. To prevent starvation, we combine our slack-based prioritization with a "batching" mechanism similar to the one from our previous work.[10] We divide time into intervals of *T* cycles, which we call *batching intervals*. Packets inserted into the network during the same interval belong to the same batch (that is, they have the same batch priority value). We prioritize packets belonging to older batches over those from younger batches. Only when two packets belong to the same batch do we prioritize them according to their available slack (that is, the slack priority levels). Each head flit carries a 5-bit slack priority value and a 3-bit batch number. We use adder delays 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 45- nanometer technology.

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 on the basis of their slack priority levels. Queues belonging to higher-priority packets take precedence over those belonging to lower-priority packets. Such prioritization reduces priority inversion and is especially needed at memory controllers, where packets from all applications are buffered together at the network interface and where monolithic queues can cause severe priority inversion.

Comparison to application-aware scheduling and fairness policies

In our previous work,[10] we proposed Stall Time Criticality (STC), an application-aware coordinated arbitration policy to accelerate network-sensitive applications. The idea is to rank applications at regular intervals based on their network intensity (L1-MPI, or L1 miss rate per instruction), and to prioritize all packets of a nonintensive application over all packets of an intensive application. Within applications belonging to the same rank, STC prioritizes packets using the baseline round-robin order in a slack-unaware manner. Packet batching is used for starvation avoidance. STC is a coarse-grained approach that identifies critical applications (from the NoC perspective) and prioritizes them over noncritical ones. By focusing on application-level intensity, STC can't capture fine-grained packet-level

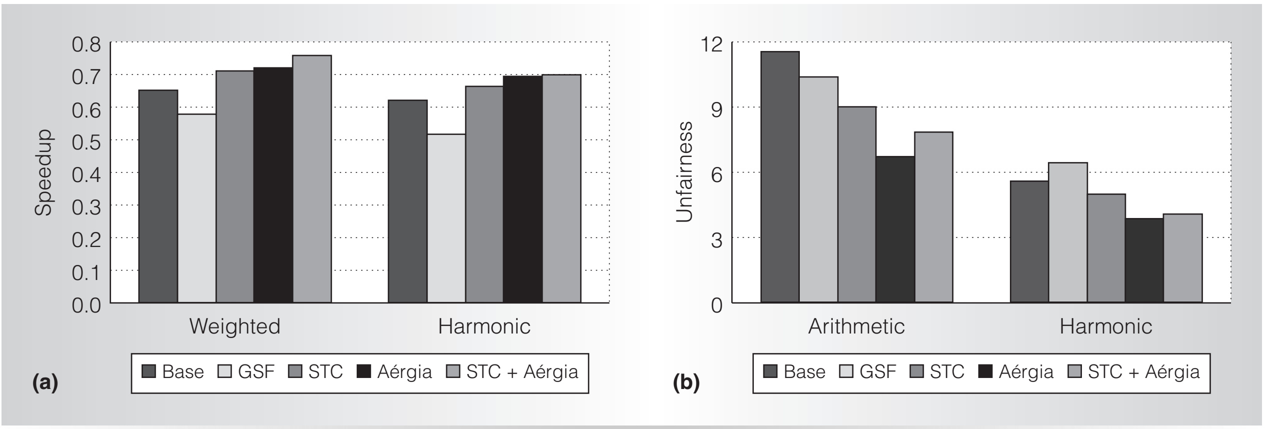


Figure 5. Aggregate results across 56 workloads: system speedup (a) and network unfairness (maximum network slowdown of any application) (b).

criticality that a slack-aware mechanism like Aérgia exploits at the individual packets' granularity. However, we find that STC and Aérgia are complementary, and we can combine them to exploit both algorithms' strengths. The combined STC+Aérgia architecture prioritizes higher-ranked applications over lower-ranked applications (as STC does) and, within the same ranking class, it uses slack-based prioritization (as Aérgia does).

We quantitatively compare Aérgia to a state-of-art fairness scheme called Globally Synchronized Frames (GSF).[11] GSF provides prioritization mechanisms within the network to ensure guarantees on minimum bandwidth and minimum network delay that each application experiences and to ensure that each application achieves equal network throughput. To accomplish this, GSF employs the notion of frames, which is similar to our concept of batches. Within a frame, GSF doesn't distinguish between different applications; in fact, it doesn't 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. Of course, GSF and Aérgia have different goals: GSF aims to provide network-level quality of service (QoS), whereas Aérgia optimizes system throughput by exploiting packet slack.

Performance evaluation

We evaluated our scheme on 35 applications consisting of SPLASH, SPEC CPU2006, SPEC-OMP, Commercial Workloads, and Windows applications on a 64-core system with an 8 × 8 mesh NoC using an integrated x86 CMP/NoC simulator. We comprehensively compare our proposed substrate to existing local scheduling policies (round-robin and age-based arbiters) and the state-of-the-art fairness-oriented policy (GSF[11]) and application-aware prioritization policy (STC[10]). Our ISCA 2010 paper[4] provides insight into why our proposal improves system performance and fairness via detailed case studies and analyses. In this article, we provide the overall results. Figure 5 summarizes our main results averaged across 56 workload mixes consisting of 8 homogenous workloads and 48 heterogenous workloads, with applications picked randomly from different categories. Aérgia improves average system throughput by 10.3 percent and 11.6 percent (weighted and harmonic, respectively) compared to the baseline (round robin), while also improving network fairness by 30.8 percent. Aérgia combined with STC improves average system throughput by 6.5 percent and 5.2 percent (weighted and harmonic, respectively) compared to STC alone, while also improving network unfairness (in terms of maximum network slowdown any application experiences) by 18.1 percent. Thus, we conclude that Aérgia provides the best

Research related to Aérgia

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

Instruction criticality and memory-level parallelism (MLP)

Much research has been done to predict instruction criticality or slack[1-3] and to prioritize critical instructions in the processor core and caches. MLP-aware cache replacement exploits cache-miss criticality differences caused by differences in MLP.[4] Parallelism-aware memory scheduling[5] and other rank-based memory scheduling algorithms[6] reduce application stall time by improving each thread's bank-level parallelism. Such work relates 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 owing to on-chip networks' distributed nature.

Prioritization in on-chip and multiprocessor networks

We've extensively compared our approach to state-of-the-art local arbitration, quality-of-service-oriented prioritization (such as Globally-Synchronized Frames, or GSF[7]), and application-aware prioritization (such as Stall-Time Criticality, or STC[8]) policies in Networks-on-Chip (NoCs). Bolotin et al. prioritize control packets over data packets in the NoC but don't distinguish packets on the basis of available slack.[9] Other researchers have also proposed frameworks for quality of service (QoS) in on-chip networks,[10-12] which can be combined with our approach. Some other work proposed arbitration policies for multichip multiprocessor networks and long-haul networks.[13-16] They aim to provide guaranteed service or fairness, while we aim to exploit slack in packet latency to improve system performance. In addition, most of the previous mechanisms statically assign priorities and 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, similar to our previous work.[8] Other researchers have explored using batching and frames in disk scheduling,[17] memory scheduling,[5] and QoS-aware packet scheduling[7,12] to prevent starvation and provide fairness.

References

1. S.T. Srinivasan and A.R. Lebeck, ''Load Latency Tolerance in Dynamically Scheduled Processors,'' *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture,* IEEE CS Press, 1998, pp. 148-159.
2. B. Fields, S. Rubin, and R. Bodík, ''Focusing Processor Policies via Critical-Path Prediction,'' *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), ACM Press, 2001, pp. 74-85.
3. B. Fields, R. Bodík, and M. Hill, ''Slack: Maximizing Performance under Technological Constraints,'' *Proc. 29th Ann. Int'l Symp.*

system performance and network fairness over various workloads, whether used independently or with application-aware prioritization techniques.

To our knowledge, our ISCA 2010 paper[4] is the first work to make use of the criticality of memory accesses for packet scheduling within NoCs. We believe our proposed slack-aware NoC design can influence many-core processor implementations and future research in several ways.

Other work has proposed and explored the instruction slack concept[7] (for more information, see the ''Research related to Aérgia'' sidebar). Although prior research has exploited instruction slack to improve performance-power trade-offs for single-threaded uniprocessors, instruction slack's impact on shared resources in many-core processors provides new research opportunities. To our knowledge, our ISCA 2010 paper[4] is the first work that attempts to exploit and understand how instruction-level slack impacts the interference of applications within a many-core processor's shared resources, such as caches, memory controllers, and on-chip networks. We define, measure, and devise easy-to-implement schemes to estimate slack of memory accesses in the context of many-core processors. Researchers can use our slack metric for better management of other shared resources, such as request scheduling at memory controllers, managing shared caches, saving energy in the shared memory hierarchy, and providing QoS. Shared resource management will become even more important in future systems (including data centers, cloud computing, and mobile many-core systems) as they employ a larger number of increasingly diverse applications running together on a many-core processor. Thus, using slack for better resource management could become more beneficial as the processor industry evolves toward many-core

Computer Architecture, IEEE Press, 2002, pp. 47-58, doi:10.1109/ISCA.2002.1003561.
4. M. Qureshi et al., ''A Case for MLP-Aware Cache Replacement,'' *Proc. 33rd Ann. Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 167-178.
5. O. Mutlu and T. Moscibroda, ''Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,'' *Proc. 35th Ann. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 63-74.
6. Y. Kim et al., ''ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers,'' *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture* (HPCA 10), IEEE Press, 2010, doi:10.1109/HPCA.2010.5416658.
7. J.W. Lee, M.C. Ng, and K. Asanovic, ''Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks,'' *Proc. 35th Ann. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 89-100.
8. R. Das et al., ''Application-Aware Prioritization Mechanisms for On-Chip Networks,'' *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2009, pp. 280-291.
9. E. Bolotin et al., ''The Power of Priority: NoC Based Distributed Cache Coherency,'' *Proc. 1st Int'l Symp. Networks-on-Chip* (NOCS 07), IEEE Press, 2007, pp. 117-126, doi:10.1109/NOCS.2007.42.
10. E. Bolotin et al., ''QNoC: QoS Architecture and Design Process for Network on Chip,'' *J. Systems Architecture,* vol. 50, nos. 2-3, 2004, pp. 105-128.
11. E. Rijpkema et al., ''Trade-offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip,'' *Proc. Conf. Design, Automation and Test in Europe* (DATE 03), vol. 1, IEEE CS Press, 2003, pp. 10350-10355.
12. B. Grot, S.W. Keckler, and O. Mutlu, ''Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip,'' *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2009, pp. 268-279.
13. K.H. Yum, E.J. Kim, and C. Das, ''QoS Provisioning in Clusters: An Investigation of Router and NIC Design,'' *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), ACM Press, 2001, pp. 120-129.
14. A.A. Chien and J.H. Kim, ''Rotating Combined Queueing (RCQ): Bandwidth and Latency Guarantees in Low-Cost, High-Performance Networks,'' *Proc. 23rd Ann. Int'l Symp. Computer Architecture* (ISCA 96), ACM Press, 1996, pp. 226-236.
15. A. Demers, S. Keshav, and S. Shenker, ''Analysis and Simulation of a Fair Queueing Algorithm,'' *Symp. Proc. Comm. Architectures & Protocols* (Sigcomm 89), ACM Press, 1989, pp. 1-12, doi:10.1145/75246.75248.
16. L. Zhang, ''Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks,'' *Proc. ACM Symp. Comm. Architectures & Protocols* (Sigcomm 90), ACM Press, 1990, pp. 19-29, doi:10.1145/99508.99525.
17. H. Frank, ''Analysis and Optimization of Disk Storage Devices for Time-Sharing Systems,'' *J. ACM,* vol. 16, no. 4, 1969, pp. 602-620.

processors. We hope that our work inspires such research and development.

Managing shared resources in a highly parallel system is a fundamental challenge. Although application interference is relatively well understood for many important shared resources (such as shared last-level caches[12] or memory bandwidth[13-16]), less is known about application interference behavior in NoCs or how this interference impacts 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 (such as queueing delays, different MLP, burstiness, and the effect of spatial location of cache banks) and hard-to-predict interference patterns that can significantly impact application-level performance.

We attempt to understand applications' complex interactions in NoCs using application-level metrics and systematic experimental evaluations. Building on this understanding, we've devised low-cost, simple-to-implement, and scalable policies to control and reduce interference in NoCs. We've shown that employing coarse-grained and fine-grained application-aware prioritization can significantly improve performance. We hope our techniques inspire other novel approaches to reduce application-level interference in NoCs.

Finally, our work exposes the instruction behavior inside a processor core to the NoC. We hope this encourages future research on the integrated design of cores and NoCs (as well as other shared resources). This integrated design approach will likely foster future research on overall system performance instead of individual components' performance: by codesigning NoC and the core to be aware of each other, we can achieve significantly better performance and fairness than designing each alone.

We therefore hope Aérgia opens new research opportunities beyond packet-scheduling in on-chip networks. MICRO

Acknowledgments

This research was partially supported by National Science Foundation (NSF) grant CCF-0702519, NSF Career Award CCF-0953246, the Carnegie Mellon University CyLab, and GSRC.

References

1. A. Glew, ''MLP Yes! ILP No! Memory Level Parallelism, or Why I No Longer Care about Instruction Level Parallelism,'' *ASPLOS Wild and Crazy Ideas Session,* 1998; http://www.cs.berkeley.edu/~kubitron/asplos98/abstracts/andrew_glew.pdf.
2. O. Mutlu, H. Kim, and Y.N. Patt, ''Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance,'' *IEEE Micro,* vol. 26, no. 1, 2006, pp. 10-20.
3. B. Fields, S. Rubin, and R. Bodík, ''Focusing Processor Policies Via Critical-Path Prediction,'' *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), ACM Press, 2001, pp. 74-85.
4. R. Das et al., ''Aérgia: Exploiting Packet Latency Slack in On-Chip Networks,'' *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), ACM Press, 2010, pp. 106-116.
5. R.M. Tomasulo, ''An Efficient Algorithm for Exploiting Multiple Arithmetic Units,'' *IBM J. Research and Development,* vol. 11, no. 1, 1967, pp. 25-33.
6. O. Mutlu et al., ''Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors,'' *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE Press, 2003, pp. 129-140.
7. B. Fields, R. Bodík, and M. Hill, ''Slack: Maximizing Performance under Technological Constraints,'' *Proc. 29th Ann. Int'l Symp. Computer Architecture,* IEEE Press, 2002, pp. 47-58, doi:10.1109/ISCA.2002.1003561.
8. D. Kroft, ''Lockup-Free Instruction Fetch/Prefetch Cache Organization,'' *Proc. 8th Ann. Symp. Computer Architecture* (ISCA 81), IEEE CS Press, 1981, pp. 81-87.
9. T.Y. Yeh and Y.N. Patt, ''Two-Level Adaptive Training Branch Prediction,'' *Proc. 24th Ann. Int'l Symp. Microarchitecture,* ACM Press, 1991, pp. 51-61.
10. R. Das et al., ''Application-Aware Prioritization Mechanisms for On-Chip Networks,'' *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2009, pp. 280-291.
11. J.W. Lee, M.C. Ng, and K. Asanovic, ''Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks,'' *Proc. 35th Ann. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 89-100.
12. L.R. Hsu et al., ''Communist, Utilitarian, and Capitalist Cache Policies on CMPS: Caches as a Shared Resource,'' *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 06), ACM Press, 2006, pp. 13-22.
13. O. Mutlu and T. Moscibroda, ''Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,'' *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2007, pp. 146-160.
14. O. Mutlu and T. Moscibroda, ''Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,'' *Proc. 35th Ann. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 63-74.
15. Y. Kim et al., ''ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers,'' *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture* (HPCA 10), IEEE Press, 2010, doi:10.1109/HPCA.2010.5416658.
16. Y. Kim et al., ''Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,'' *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2010.

Reetuparna Das is a research scientist at Intel Labs. Her research interests include computer architecture, especially interconnection networks. She has a PhD in computer science and engineering from Pennsylvania State University.

Onur Mutlu is an assistant professor in the Electrical and Computer Engineering

Department at Carnegie Mellon University. His research interests include computer architecture and systems. He has a PhD in electrical and computer engineering from the University of Texas at Austin. He's a member of IEEE and the ACM.

Thomas Moscibroda is a researcher in the distributed systems research group at Microsoft Research. His research interests include distributed computing, networking, and algorithmic aspects of computer architecture. He has a PhD in computer science from ETH Zürich. He's a member of IEEE and the ACM.

Chita R. Das is a distinguished professor in the Department of Computer Science and Engineering at Pennsylvania State University. His research interests include parallel and distributed computing, performance evaluation, and fault-tolerant computing. He has a PhD in computer science from the University of Louisiana, Lafayette. He's a fellow of IEEE.

Direct questions and comments to Reetuparna Das, Intel Labs, Intel Corp., SC-12, 3600 Juliette Ln., Santa Clara, CA; reetuparna.das@intel.com.