A Model for Application Slowdown Estimation in On-Chip Networks and Its Use for Improving System Fairness and Performance

Xiyue Xiang[†] Saugata Ghose[‡] Onur Mutlu[§‡] Nian-Feng Tzeng[†]
[†]*University of Louisiana at Lafayette* [‡]*Carnegie Mellon University* [§]*ETH Zürich*

***Abstract*—In a network-on-chip (NoC) based system, the NoC is a shared resource among multiple processor cores. Network requests generated by different applications running on different cores can interfere with each other, leading to a *slowdown* in performance of each application. The degree of slowdown introduced by this interference varies for each application, as it depends on (1) the sensitivity of the application to NoC performance, and (2) network traffic induced by other applications running concurrently on the system. In modern systems, NoC interference is largely uncontrolled, and therefore some applications *unfairly* slow down much more than others. This can lead to overall system performance degradation, prevent fair progress of different applications, and cause starvation of unfairly-treated applications. Our goal is to accurately model the slowdown of each application executing on the system due to NoC interference *at runtime*, and to use this information to improve system performance and reduce unfairness.**

To this end, we propose the *NoC Application Slowdown* (NAS) *Model*, the first online model that accurately estimates how much network delays *due to interference* contribute to the overall stall time of each application. The key idea of NAS is to determine how the delays induced at each level of network data transmission overlap with each other, and to use the overlap information to calculate the net impact of the delays on application stall time. Our model determines the application slowdowns at runtime with a very low error rate, averaging 4.2% over 90 multiprogrammed workloads for an 8×8 mesh network. We use NAS to develop *Fairness-Aware Source Throttling* (FAST), a mechanism that employs slowdown predictions to control the network injection rates of applications in a way that minimizes system unfairness. Our results over a variety of multiprogrammed workloads show that FAST improves average system fairness and performance by 9.5% and 5.2%, respectively.

1. Introduction

A network-on-chip (NoC) is an essential component of a multicore system, providing an efficient substrate to interconnect multiple cores, private caches, and the shared last-level cache (LLC). In such systems, the NoC carries traffic to serve memory requests [42, 69], maintain cache coherence [14], and perform synchronization (for multithreaded applications) [36]. The NoC is the first point at which applications running on the cores contend with each other, as the NoC is shared among the cores. This causes *inter-application interference*, where the additional delays induced on the network requests causes each application to *slow down* compared to when it is running alone in the system. The amount of slowdown due to inter-application interference depends on both the sensitivity of an application to NoC contention, and the behavior of the other applications running concurrently. If left unmanaged, inter-application interference can lead to *unfairness* among the applications, i.e., some applications can slow down much more than others.[1] Such unfair slowdowns can lead to (1) overall system performance degradation (since some cores make very slow progress and system utilization goes down) [11, 41, 43, 44, 49], and (2) poor quality-of-service (since unfairly-treated applications can not only starve for long periods of time but also might not attain their performance requirements) [25, 30, 31, 63, 64, 67].

A number of runtime mechanisms can provide *fairness* to applications running together, but the mechanisms first need to find out how much slowdown each application suffers. Many prior works (e.g., [17, 63, 64]) quantitatively define the *slowdown* of an application as the ratio of its execution time when the application runs together with other programs (i.e., *shared execution time*) over its execution time when it runs by itself on the system (i.e., *alone execution time*). In a multicore system, estimating the slowdown amount at runtime is challenging. While the shared execution time can be measured easily by counting the number of cycles elapsed in the system while the application is running, the alone execution time is unknown at runtime unless it is obtained through profiling, where the application must be run *in isolation* on an identical system. Obtaining such *a priori* knowledge of alone execution time for any given point during an application's execution can be costly, as it requires a second run of the application, and is infeasible in several environments (e.g., a virtualized cloud server where different remote users submit jobs at will for independent execution).

Past works propose mechanisms to *estimate* the alone execution time at *runtime* using various models (e.g., [5, 17, 41, 49, 63, 64]). Unfortunately, none of these models account for *NoC interference*, as they are designed for small-scale architectures that do not use an on-chip network. Prior slowdown models cannot be easily adopted for NoC interference, as they account mainly for centralized points of contention (e.g., the input port to a globally-shared cache, the memory controller). In contrast, a request within the NoC experiences interference in a *distributed* manner, where it contends with a different set of requests at every network hop [12, 13, 24, 25].

Our goal in this work is to develop a model that accurately and efficiently determines the slowdown of an application running on a NoC-based multicore system, without relying on a priori information. To this end, we propose the *NoC Application Slowdown* (NAS) *Model*. NAS determines the delays that take place for each network request at the different transaction granularities used within the NoC (i.e., per flit, per packet, and per request) due to inter-application interference. NAS then calculates what part of this delay contributes to slowdowns, accounting for latency overlaps and reordering at each granularity. By eliminating overlapping latencies, NAS can identify which of the requests actually fall along the critical path of application execution [22,27,35,60]. We find that NAS is an accurate online model of application slowdown, with an average error rate of only 4.2% for an 8×8 mesh network across a wide range of applications.

With the accurate online slowdown estimates NAS provides during execution, the system can enable a variety of mechanisms to control application slowdowns (e.g., ensure that resources are shared fairly among applications, or prioritize an application that is slowed down too much to meet its performance requirements). One example application is *source throttling*, where the rate at which each node injects requests into the NoC is controlled to improve system utilization. Previously-proposed throttling mechanisms do *not* have slowdown estimates available to them, and use more naive characteristics (e.g., network intensity [7], speed of execution [51, 52], injection rate [66]), which are oblivious to the application slowdowns within the system, to make throttling decisions. We show that more informed decisions should be made when choosing *which* applications to throttle by taking into account each application's slowdown.[2] To this end, we develop *Fairness-Aware Source Throttling* (FAST), a hardware mechanism that uses the NoC slowdown estimates generated by NAS at runtime to *intelligently* identify applications for which throttling both reduces network interference and incurs minimal performance degradation. FAST improves average system fairness and performance by 9.5% and 5.2%, respectively, for a variety of multiprogrammed workloads.

In this work, we make the following contributions:

- We provide a detailed breakdown of the interference delays experienced by a network request at different transaction granularities

[1]Many recent works quantify *unfairness* as the largest single application slowdown in a multiprogrammed workload [1, 6, 7, 11–13, 16, 17, 33, 34, 43, 56–58, 61–64, 67, 68, 70].

[2]For example, if an application spends most of the time executing instructions in the core, throttling such an application might not significantly reduce interference, as the application seldom generates network requests. In fact, throttling the application may adversely impact system performance, as the latency of its few outstanding requests might be on the critical path of execution [44, 49].

(i.e., flits, packets, and requests). The breakdown allows us to determine how delays within a network contribute to reducing overall application performance.

- We propose NAS, the first online model to accurately estimate application slowdowns due to inter-application interference within the NoC. NAS identifies how the interference delays incurred by the pieces of a network request overlap with each other, to accurately determine the effective impact of these delays on the critical path of application execution and hence application performance.
- We develop FAST, a new hardware mechanism that throttles NoC nodes based on NAS's slowdown estimates. FAST uses the slowdown information to throttle only those applications where throttling improves fairness and thus improves system utilization. We find that FAST improves both system fairness and performance.

2. Background & Motivation

We introduce key concepts on interference and slowdowns, and motivate the need for a slowdown model for NoCs. First, we discuss prior application slowdown models (Section 2.1). Then, we study how interference in a NoC leads to application slowdowns (Section 2.2).

2.1. Interference and Slowdown

When multiple applications run concurrently on a multicore system, they interfere with each other at different shared resources, such as the NoC, last-level cache, and main memory. Ideally, the impact of such interference should be similar for equal-priority applications [17, 41, 44, 49]. However, applications have different sensitivities to interference at each of the shared resources, and thus the slowdown of each application can vary significantly, resulting in system *unfairness*.

Prior works [5, 17, 41, 49, 63, 64] propose methods to estimate an application's slowdown due to interference in the shared LLC capacity and/or main memory bandwidth, at runtime, on an interval-by-interval basis, in the presence of other co-running applications. While the *shared execution time* of an application can be obtained by direct measurement, estimating *alone execution time* is much more difficult since direct measurement of this requires a second run of the application on an identical system. As a result, prior works often rely on one of two methods to estimate an application's alone execution time.

The first method, which we call *fine-grained interference tracking*, estimates alone execution time by determining the number of cycles by which each application request is delayed due to interference and identifying the additional execution time the application incurs because of such request delays. Stall-Time Fair Memory (STFM) [49] estimates the additional memory stall time of an application in main memory by tracking the excess stall time due to each request at the memory controller. Fairness via Source Throttling (FST) [17] and Per-Thread Cycle Counting (PTCA) [5] take the LLC capacity into account to determine the alone execution time of an application. These methods (1) estimate the additional *contention misses* that occur due to interference in the cache, i.e., those misses that would otherwise have been a hit, if the application were running alone; and (2) use the additional contention miss estimate along with the additional stall time in the memory controller to estimate the additional stall time an application incurs in the presence of both LLC and main memory interference.

The second method, *coarse-grained interference tracking*, is based on the observation that an application's performance is correlated with some related heuristics, such as memory or cache service rates. *Slowdown* is estimated as the ratio of the service rate during shared execution to the rate during alone execution. Memory-Induced Slowdown Estimation (MISE) [63] cycles through each application during shared execution, periodically giving the maximum priority to that application's requests to main memory. It then records the memory request service rate of the application while it has maximum priority as an estimate of the service rate during alone execution. The Application Slowdown Model (ASM) [64] extends upon this by taking into account interference at both the LLC and main memory, using an auxiliary cache tag store for each application to account for the aggregate impact of contention misses on the cache request service rate.

Existing work has thus proposed online models for estimating application slowdowns due to interference at both main memory (e.g., STFM, MISE, ASM) and the shared LLC (e.g., FST, PTCA, ASM), but no prior work has modeled the effects of NoC interference on application slowdown. All of these prior works on slowdown estimation deal with only a *centralized* point of contention (e.g., the input port to a globally-shared cache, the memory controller). As we discuss in Section 3.1, interference in a NoC occurs in *distributed* manner [12, 13, 24, 25] across different network hops, and as a result, prior slowdown models cannot be adopted easily to incorporate the NoC interference. Therefore, we need to construct a new slowdown model designed to capture the distributed nature of interference in a NoC.

2.2. Impact of NoC-Level Inter-Application Interference

Figure 1 depicts a typical NoC-based multicore system. Each node in the network contains a core, a private L1 cache, a slice of the shared last-level cache (LLC), Miss Status Holding Registers (MSHRs), a network interface (NI), and a router. The NoC connects all nodes within a single chip. When an application running on the core needs data from one of the LLC slices, an L1 miss *request* is generated,[3] and an MSHR entry is allocated. The MSHR tracks all outstanding requests to the LLC [37], and the number of available MSHR entries effectively acts as a knob that controls the number of requests that can be injected into the NoC by the node.

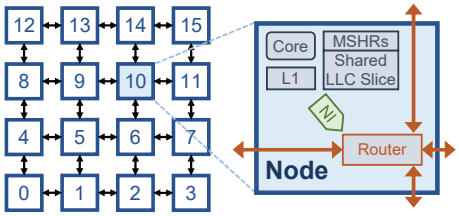


Fig. 1. NoC-based multicore system with a 2D mesh network topology.

An L1 miss request destined for a remote LLC slice is broken down into multiple *packets*. The packets travel through NoC routers. Each packet consists of one or more *flits*, the base unit of data movement within the NoC. A typical NoC router has only a small number of buffers, sometimes containing just a single register at each input/output port. Inter-application interference happens in two major ways in a router: (1) when its buffers are full, the router cannot accept new packets, rejecting an application's packet while another's packet is in the router buffers; or (2) the router can choose one application's packet to schedule over another's to the same output port. The backpressure caused by routers can propagate back to the core issuing the L1 miss request, and because of other applications' requests occupying resources in the network, the core might not be able to inject requests into the network, which is also a manifestation of inter-application interference in the NoC. In this work, we focus on quantifying such effects in a NoC caused by inter-application interference. When packets return to the requesting core, they are reassembled by the MSHR [17, 19].

Figure 2 demonstrates system unfairness resulting from inter-application interference for an example multiprogrammed workload that consists of 16 copies each of 4 applications. This figure shows the relation between application slowdown, shown on the left x-axis, and *network intensity* (i.e., MPKI, the number of L1 misses per kilo instructions), shown on the right x-axis. To compute slowdown of each of the four applications, we first obtain the alone execution time by running a single instance of the application separately on a 64-core system (see Section 5 for our methodology). We then obtain the shared execution time by running 16 instances each of the four applications concurrently on the 64-core system.

We make three observations from our case study. First, Figure 2 affirms that inter-application interference in the NoC can lead to significant application slowdowns, as high as 2.7× the alone execution time. Second, *leslie3d* has a slowdown of 2.7, which is 1.7× higher than the slowdown of *GemsFDTD*, which demonstrates unfair progress across applications. Third, while we expect that applications that generate more requests to the memory (i.e., those that have higher MPKIs) would incur bigger slowdowns, our observations show that the magnitude of the slowdown is *not* necessarily coupled with network traffic intensity. For example, although the MPKI of *mcf* is 1.7× higher than that of *lbm*, both applications slow down by a similar amount. Thus,

[3] An L1 miss request is generated for (1) an L1 read or write miss; or (2) a write hit in the local L1 cache, where the local node does not have the coherence permissions to modify the cache block.

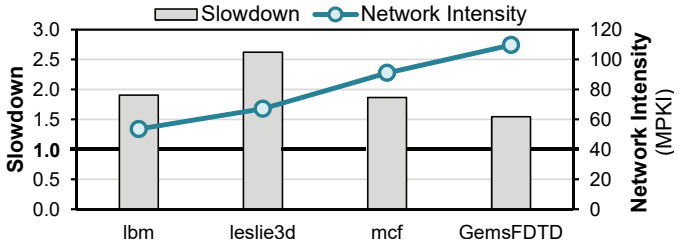


Fig. 2. Slowdown due to interference (left axis) and network intensity (right axis) of applications, when run together on a NoC-based 64-core system.

each L1 miss causes a smaller impact in *mcf* than in *lbm* (i.e., *mcf* is less sensitive). As we discuss in Section 4, throttling applications such as *mcf*, with high network interference and lower slowdowns, may alleviate network congestion and improve fairness without noticeably compromising performance.

3. NAS: NoC Application Slowdown Modeling

Motivated by our observations in Section 2, we first propose the *NoC Application Slowdown (NAS) Model*, which estimates, *at runtime*, the application slowdowns caused by inter-application interference in the NoC. To our knowledge, NAS is the first model of interference-induced slowdowns in a NoC. We first provide an overview of NAS, describing the challenges for modeling interference-induced slowdown in a NoC (Section 3.1). We then elaborate on how NAS deals with those challenges to obtain accurate slowdown estimates, studying how interference affects delays at the flit level (Section 3.2), packet level (Section 3.3), request level (Section 3.4), and application level (Section 3.5).

3.1. Overview

Application slowdown is computed as the ratio of the shared execution time (t_{shared}) to the alone execution time (t_{alone}) for some portion of execution [17]:

$$slowdown = \frac{t_{shared}}{t_{alone}} = \frac{t_{shared}}{t_{shared} - \Delta t_{stall}} \quad (1)$$

For runtime slowdown estimation mechanisms, t_{shared} can be measured on-the-fly simply by using a counter of the elapsed execution time while the application is running. The key challenge is to estimate t_{alone}. Akin to FST [17], NAS determines the alone execution time by estimating the amount of *additional* stall time introduced by inter-application interference (Δt_{stall}).

NAS is different from all prior work on slowdown estimation in two aspects. First, interference in a NoC occurs in a *distributed* manner, as an in-flight request often contends with a *different set of requests* at *different nodes* it visits along the network. Specifically, a request may contend with a different set of requests in flight at every hop, where each contending request may involve multiple flits and packets originating from different nodes. Second, NAS must estimate the amount by which the additional delay impacting each network request affects the slowdown of the *overall application*. Interference-induced network delay is important only if it causes an increase in the application's stall time. In modern processors, out-of-order execution allows multiple memory requests to be in flight concurrently (i.e., memory-level parallelism [9, 23, 38, 44–46, 48, 55]), and the latencies of these requests can overlap and be (partially) hidden, without contributing or only partially contributing to the application stall time [13, 17, 22, 48]. The only latencies that contribute to application slowdowns are those requests, known as *critical loads* [22, 35, 60], whose latencies are *not fully overlapped* by other requests and thus fall along the critical path of execution time. This is in contrast to prior work, which correlates certain events (e.g., a cache miss caused by contention [5,17,64] and/or delays at the main memory controller [49, 63, 64]) to slowdowns.

In NAS, as each flit travels through the network, we use new counters that track how long a flit waits as a result of losing arbitration to another flit *from a different application*, which we call *flit-level interference* (Section 3.2). We then use this information to determine *packet-level interference*, as each packet is made up of one or more flits (Section 3.3). Note that this is not as trivial as summing up flit-level interference, since latencies of multiple flits in a packet can be overlapped. We use packet-level interference to determine the overall *request-level interference*, combining the delays experienced by the multiple packets that make up a single request (Section 3.4). Once we have the interference delay for a request, we then determine what portion of this interference delay is responsible for prolonging the application stall time (Section 3.5). Finally, we can obtain Δt_{stall} and compute the slowdown using Equation 1. Overall, NAS can be implemented with low overhead, as shown in Section 6.1.

3.2. Flit-Level Interference

A flit is the smallest unit transferable through the NoC. Depending on the internal microarchitecture of a router, inter-application interference on a flit may occur at various stages within the NoC. In this work, we assume that the NoC uses conventional buffered routers with virtual channels that employ dimension-order routing [10], though NAS can be easily extended to more sophisticated NoCs. Flits are transferred independently across the network. NAS identifies interference at all arbitration points in every NoC router that the flit travels through. During arbitration, we say that a losing flit experiences inter-application interference from a winning flit only if the two flits are issued by different applications. We track three interference events: flit admission, virtual channel arbitration, and switch arbitration. We keep a counter within each flit, called Δt_{flit}, which tracks the number of cycles that the flit had to wait because it lost arbitration. For each of the three interference events, we increment Δt_{flit} for a losing flit only when the flit that wins arbitration is from a different application:

$$\Delta t_{flit} = \begin{cases} \Delta t_{flit} + 1, & \text{if } AppID_{flit} \neq AppID_{winning\_flit} \\ \Delta t_{flit}, & \text{otherwise} \end{cases} \quad (2)$$

3.3. Packet-Level Interference

Packet-level interference is determined when all of the constituent flits of a packet arrive at the destination node. We cannot simply add up the interference delay of the constituent flits to determine packet-level interference. This is because the transit latencies of flits overlap with each other. *When an application runs alone*, we observe that most of the time, a packet's flits arrive consecutively at each router, including the destination: they take a constant M cycles (where M is the number of flits in a packet) to be reassembled after the arrival latency of the first flit (t_{first_flit}), as shown in Figure 3a. Therefore, when an application runs together with other applications, we can calculate the interference delay of a packet (Δt_{packet}) using two parts: (1) the total amount of interference delay induced on the first flit to arrive (Δt_{first_flit}), and (2) the *increase* in reassembly latency (i.e., the portion of the reassembly latency *greater than* M, see Figure 3b). Hence, Δt_{packet} is obtained using the arrival times of the first and last flits ($T_{first\_arrival}$ and $T_{last\_arrival}$):

$$\Delta t_{packet} = \Delta t_{first\_flit} + \left(T_{last\_arrival} - T_{first\_arrival} - M\right) \quad (3)$$

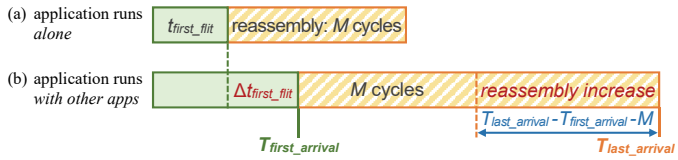


Fig. 3. Computing packet-level interference based on flit-level interference.

3.4. Request-Level Interference

A request being serviced by the NoC is made up of a sequence of packets. When a node issues a request, it generates a *control* packet that contains information about the memory being requested (e.g., the data address, the ID of the node containing the data, access type and size). When the control packet arrives at the *home node* of the data (i.e., the node where the LLC slice containing the requested data address resides), the home node performs a cache lookup, and then generates a *data* packet to send the results of the cache lookup back to the requestor node. The packets belonging to a request are serialized, as the data packet cannot be generated before the control packet arrives at the home node and the cache lookup completes.

To calculate the interference delay of a memory request $\Delta t_{request}$, we simply calculate the lumped sum of Δt_{packet} for the control and data packets, as the packet latencies do not overlap due to serialization:[4]

$$\Delta t_{request} = \Delta t_{control\_packet} + \Delta t_{data\_packet} \quad (4)$$

Since the control packets never return to the requestor node, we need a mechanism within the network that can perform the summation operation and carry the sum back to the requestor. We observe that the control and data packets associated with a memory request always form a *closed-loop path*. We take advantage of this by having a data packet *inherit* the Δt_{packet} value of its associated control packet (i.e., $\Delta t_{control\_packet}$) when the data packet is first generated. By doing this, when the data packet returns to the requestor node, it now contains the sum of $\Delta t_{control\_packet}$ and Δt_{data_packet}, which as we show in Equation 4 is the same as $\Delta t_{request}$. To facilitate Δt_{packet} inheritance, we add a small data structure to the network interface (*NI*).

Figure 4 shows an example of how control and data packets are used to serve an L1 miss from a remote LLC slice. In the example, when *Node S* incurs an L1 read miss, it injects a *request* (i.e., control) packet to fetch data from *Node D* (the home node of the data). The request packet triggers a cache access and retrieves the desired block of data from the LLC slice within *Node D*.

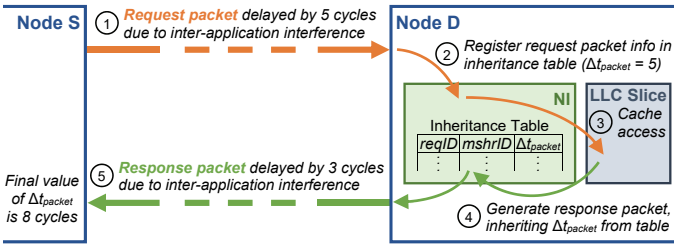


Fig. 4. Packet delay inheritance from a control packet generated by a memory request to a data packet fulfilling the request.

The interference delay of the entire memory request is carried back to *Node S* by the *response* packet from *Node D* in five steps:

1) The request packet is delayed by 5 cycles due to inter-application interference. The delay is recorded in the packet's Δt_{packet} field.
2) Every packet associated with a request can be uniquely identified by the tuple {*reqID*, *mshrID*}, where *reqID* is the index of the node issuing the request and *mshrID* is simply the MSHR index of the request in that node [20]. In NAS, when the request packet arrives at *Node D*, the router ejects the packet and registers the packet's *reqID*, *mshrID*, and Δt_{packet} fields into the *inheritance table*.
3) The LLC access is initiated inside *Node D*.
4) When the LLC access completes, the Δt_{packet} value is retrieved from the inheritance table by using the *reqID* and *mshrID* as the lookup index, and the Δt_{packet} value is appended to the payload data during packetization of the response packet.
5) Any NoC interference experienced by the response packet is added to the Δt_{packet} field stored in the packet. As a result, Δt_{packet} contains the *cumulative* interference delay of both the request and response packets. Upon receiving the response packet, *Node S* can simply extract Δt_{packet} from the packet to get the total interference delay the request incurred.

3.5. Application Stall Time

Once NAS has the total interference delay experienced by a request, it must determine how much of the request's delay actually increases the application stall time, i.e., contributes to Δt_{stall}. As prior work has shown [13, 22, 35, 55], only the interference delay of critical memory requests should increment Δt_{stall}, and the interference delay of non-critical requests should be simply ignored. Even for a critical load, not all of its interference delay contributes to Δt_{stall} due to overlapping of the load latency [13, 22, 35, 46–48, 55].

[4]We ignore the delays of write-back packets and invalidation packets destined to a sharer, since such operations do *not* fall on the critical path of execution.

For out-of-order processors, an application stall occurs if the instruction window, or re-order buffer (ROB), is full, or if no load/store (LSQ) queue entries are available for the next memory request, at which point we say the request *becomes critical*. Once one of these events occurs, the oldest memory request (as tracked in the ROB or LSQ) becomes critical, as until this request is retired (i.e., committed), the instruction window, ROB, and/or LSQ cannot be freed up [22, 35, 45]. From the application point of view, a critical request blocks forward progress until $T_{service\_shared}$ — the moment when the request is completed and retired. We ignore any interference delay prior to when the request becomes critical ($T_{critical\_shared}$). Hence, the increase in application stall time due to a single request $\Delta t_{stall\_per\_request}$ is computed as:

$$\Delta t_{stall\_per\_request} = \min\left(T_{service\_shared} - T_{critical\_shared}, \Delta t_{request}\right) \quad (5)$$

NAS computes Δt_{stall} as the sum of $\Delta t_{stall\_per\_request}$ of *all requests* issued during the application execution time that is sampled:[5]

$$\Delta t_{stall} = \sum_i \Delta t_{stall\_per\_request,i} \quad (6)$$

NAS then computes the slowdown induced by NoC-level interference by using Δt_{stall} in Equation 1.

4. FAST: Fairness-Aware Source Throttling

NAS can be used to develop many different types of network mechanisms (e.g., packet scheduling, source throttling, data mapping) that can improve fairness, performance, and quality-of-service (QoS). In this work, we examine one such application, where the slowdown predictions of NAS are used to control source throttling decisions. State-of-the-art throttling mechanisms [7, 51, 52] rely on heuristics that are only somewhat correlated to application slowdown to decide which network nodes to throttle. Having the application slowdown information from NAS can enable us to throttle in a more informed and sophisticated manner. Prior work has shown that incorporating slowdown information into fairness mechanisms can improve QoS and system utilization [17, 32, 49, 63, 64, 67]. In this work, we aim to show one example where slowdown estimates from NAS can be used to improve network fairness and thereby improve system utilization (and thus performance). To this end, we develop *Fairness-Aware Source Throttling* (FAST). The key idea of FAST is to throttle down only applications whose slowdown is not significantly impacted by the application's network injection rate. FAST throttles source nodes up or down simply by increasing or decreasing the number of MSHR entries available at each core.

4.1. Application Classification

FAST classifies applications into two categories, *latency-sensitive* or *throughput-sensitive*. A latency-sensitive application spends most of its time within the processor and has lower MLP [9, 12, 23, 40, 45, 46, 55]. Any delay of its memory requests is likely to fall along the critical path of execution time, causing the application to stall. Adding delay to a single miss in this case is more likely to extend the application stall time. In contrast, a *throughput-sensitive* application tends to exert more pressure on the network. The latencies of L1 misses of a throughput-sensitive application (especially one that has high network intensity) are far more likely to overlap due to memory-level parallelism, especially when their requests traverse longer distances (e.g., in a larger network). Throttling down throughput-sensitive applications has the potential to alleviate network interference without a significant performance penalty to such applications (since such applications make relatively slow progress anyway as they are network-throughput-bound). As a result, latency-sensitive applications experience lower interference, leading to fewer stalls and thus higher system performance.

FAST utilizes the number of L1 misses per cycle (MPC) to classify applications into the two categories and make throttling decisions. MPC can be obtained easily at each core with a single counter. We determine the threshold of MPC (*MPCThreshold*) for classifying applications empirically (see Section 6.2).

[5]There is no overlap of $\Delta t_{stall\_per\_request}$ because at most one request can be critical within a processor at any given time, and we have already eliminated the stall time before a request becomes critical from $\Delta t_{stall\_per\_request}$.

FAST is motivated by **two key observations**: (1) we observe that throttling latency-sensitive applications not only fails to improve fairness, but in fact adversely impacts system performance; and (2) *blindly* throttling the throughput-sensitive applications also increases unfairness. Unfairness increases due to two reasons. First, the slowest application, which determines the unfairness of a system, is often throughput-sensitive, and throttling it further (as can be done by HAT [7]) would increase system unfairness. FAST utilizes the slowdown estimates from NAS to directly identify the slowest applications, preventing them from being throttled down. Second, throttling down an application can negatively impact application performance, as having fewer available MSHR entries increases the likelihood that no free entries are available (i.e., a network request cannot be issued), potentially causing the application to stall. To minimize this, FAST employs a new metric, NoC stall time criticality (Section 4.2), to select the applications whose execution is less sensitive to the number of MSHR entries.

4.2. NoC Stall Time Criticality: A New Metric

Prior work deems L1 MPKI (i.e., network intensity) to be a good indicator of application performance [7, 11, 33]. However, as stated in Section 2.2, application performance is not necessarily coupled with network intensity directly (see Figure 2). In fact, many factors cause applications with the same MPKI to slow down at different rates. Our goal is to design an improved fairness mechanism, which can gauge the performance impact of throttling the rate at which an application can issue L1 miss requests on the application's slowdown. While slowdown informs us of the *aggregate effect* that network interference has on an application, it alone cannot be used to determine by how much throttling the L1 miss issue rate can affect the application's slowdown.

We introduce a new metric, called *NoC stall time criticality* (STC_{noc}), which quantitatively expresses how much *each* L1 miss (i.e., a request that requires an MSHR entry) of an application contributes to the overall application slowdown:

$$STC_{noc} = \frac{slowdown}{\text{L1 miss count}} \tag{7}$$

For an application with a lower STC_{noc}, each L1 miss request contributes less to application slowdown, and thus the application tends to be less sensitive to NoC-level interference. FAST uses STC_{noc} to proactively estimate the expected interference impact of an L1 miss on an application's performance.

4.3. Source Throttling

Algorithm 1 summarizes the high-level decision flow of FAST. At the beginning of every epoch (100K cycles for our experiments), FAST sorts all of the applications by *slowdown*. It also categorizes an application as latency-sensitive if the application's MPC value falls below *MPCThreshold*, and as throughput-sensitive otherwise (as explained in Section 4.1). FAST throttles up *NumThrottleUp* applications that have the largest slowdowns and any other latency-sensitive applications, by allowing these applications to use *all* of their MSHR entries.

Algorithm 1 Fairness-Aware Source Throttling

At the beginning of each epoch:
classify applications as throughput- or latency-sensitive based on *MPC*
rank applications based on their *slowdown*
throttle up *NumThrottleUp* highest-slowdown applications
throttle up all other latency-sensitive applications
if (*max_slowdown* - *min_slowdown*) > *slowdown_threshold*) **then**
  throttle down *NumThrottleDown* throughput-sensitive applications probabilistically based on STC_{noc} value
end if

If the difference in slowdown between the fastest and slowest applications exceeds the *slowdown_threshold* value, a high amount of disparity exists between applications. Only in this case, FAST changes the maximum rate of request injection into the network for each core by throttling down *NumThrottleDown* throughput-sensitive applications. We proactively estimate the performance impact of throttling based on STC_{noc}, to reduce interference while keeping the performance loss of the throttled-down application low. As discussed in Section 4.2, an application with a lower STC_{noc} value is less likely to have its slowdown increase significantly if we throttle down the rate at which the application can issue L1 miss requests.

FAST probabilistically selects an application for throttling based on the inverse of its STC_{noc} value (i.e., an application with a smaller STC_{noc} value has a higher probability of being throttled down than an application with a larger STC_{noc} value). The probabilistic nature of the algorithm avoids starving applications with the lowest STC_{noc} values, as such applications are *not always* throttled and can still make forward progress. When an application is first throttled down (i.e., it was previously using all of its MSHR entries), FAST reduces the maximum number of MSHR entries the application can use down to 50%, to quickly relieve network interference. If an application that is already throttled down is selected in subsequent epochs to be further throttled down, FAST starts to *gradually* bring the application's network utilization down, by decrementing the number of MSHR entries available to the application by 1 in each epoch, as more drastic reductions to an already-throttled application can significantly increase its slowdown. When an application has only 10% of its MSHR entries available, it cannot be throttled down further.

Note that *NumThrottleDown* and *NumThrottleUp* are used to prevent over- and under-throttling. We provide the values we use for these parameters in Section 6.2. The *slowdown_threshold* parameter (set empirically to 0.2) is used to trigger throttling. These parameters can be tuned dynamically at runtime to adjust the tradeoff between fairness and performance in FAST, which we leave for future work.

4.4. Fail-Safe Mechanism

FAST includes a fail-safe provision to prevent unnecessary system performance loss and system fairness penalty. FAST tracks the change in unfairness ($\Delta unfairness$) and the average change in slowdown of applications ($\sum \Delta slowdown/N$, where N is the number of cores in the system) from the previous epoch to the current epoch. The throttling decisions made by FAST for the current epoch are considered to be "good" if:

$$\frac{\sum_{i=0}^{N} \Delta slowdown_i}{N} + \Delta unfairness \leq 0 \tag{8}$$

By using Equation 8, FAST considers a decision to be good in one of three scenarios:

1) Both average system slowdown and unfairness (i.e., the largest slowdown of any single application) decrease.
2) The *decrease* in system unfairness (i.e., the amount by which the slowdown of the application with the largest slowdown is reduced) outweighs the increase in the average slowdown of applications. In this case, system fairness is deemed to be improved at the cost of *reasonable* performance degradation.
3) The reduction in average slowdown of applications outweighs the *increase* in system unfairness (i.e., the largest slowdown of any single application). In this case, system performance is deemed to be improved at the cost of *reasonable* fairness penalty.

If a decision does not meet any of these three requirements, FAST sets the MSHR quotas back to those of the last "good" decision.

4.5. Impact of FAST on Fairness and Performance

FAST improves system fairness by throttling down *throughput-sensitive* applications with *smaller slowdowns*. Throughput-sensitive applications tend to generate more L1 miss requests into the network. Limiting the number of requests from these applications can effectively reduce interference to other applications, thus speeding up execution. By using the slowdown estimates provided by NAS to pick applications with smaller slowdowns, FAST avoids throttling the throughput-sensitive applications that are already running much slower than the others. FAST proactively throttles down those applications that incur a lower performance penalty based on our new metric, STC_{noc}. By judiciously avoiding applications where throttling can be harmful, FAST ensures that applications with greater slowdowns can ramp up their execution, to improve system fairness and performance.

5. Methodology

We evaluate NAS using a modified version of NOCulator [2,21,50], an open-source cycle-accurate network-on-chip simulator. We faithfully model a directory-based MOESI coherence protocol. We also model a perfect shared LLC (i.e., all requests hit in the home LLC

slice) to stress the network, as done in many prior works (e.g., [7, 19, 20,42,51,52]).[6] Each node in the network contains a core, a private L1 cache, and a slice of the shared LLC, modeling a static non-uniform cache architecture (S-NUCA) [29]. Each control packet consists of one flit, while each data packet contains four flits. Table 1 lists the main system parameters used in our evaluation.

Table 1. Major system configuration parameters.

Processor	3-issue out-of-order core, 128-entry instruction window, 128-entry ROB
L1 Caches	64KB per core, 4-way set associative, 64B block size, 2-cycle latency, 16 MSHRs
L2 Cache	Perfect shared LLC, S-NUCA [29], 5-cycle latency
Coherence	MOESI protocol
Network	2D mesh topology (4×4, 8×8), dimension-order routing
Router	8 virtual channels (VCs), 4 flits/VC, 2-cycle latency

To build our multiprogrammed workloads, we use PinPoints [54] to obtain instruction traces from representative phases of applications from the SPEC CPU2006 benchmark suite [26]. Each core runs an instance of an application, where the simulation warms up the cache for 1M cycles and then runs for another 5M cycles.

We profile 26 applications from the SPEC CPU2006 benchmark suite, and classify them based on their network intensity (i.e., MPKI) into one of three categories: *low*, *medium*, or *high*. We construct 90 multiprogrammed workloads for both 4×4 and 8×8 NoCs, with three groups of 30 applications each: Random (which includes *low*-, *medium*-, and *high*-intensity applications), Mixed (which includes only *medium*- and *high*-intensity applications), and Heavy (which includes only *high*-intensity applications). Each workload contains *multiple instances* of 4 different applications (4 instances per application for a 4×4 network, and 16 instances per application for an 8×8 network), which are mapped in an interleaved manner and run independently on all cores. We assume, without loss of generality, that each core runs only one application at any given time.[7]

6. Evaluation

We first demonstrate that our NAS model estimates application slowdowns with high accuracy (Section 6.1). We then show that FAST provides improved system fairness and performance (Section 6.2).

6.1. NAS Accuracy

To evaluate the accuracy of NAS, we execute our multiprogrammed workloads, and then calculate the *slowdown estimation error* for the entire execution (i.e., 5M cycles) of each instance of an application within the workload. The slowdown estimation error of each application instance is computed as:

$$error = \frac{\text{slowdown estimated by NAS} - \textit{actual slowdown}}{\textit{actual slowdown}} \quad (9)$$

$$\textit{actual slowdown} = \frac{IPC_{aloneActual}}{IPC_{shared}} \quad (10)$$

where $IPC_{aloneActual}$ is obtained by actually running a single instance of the application individually (as opposed to estimating IPC_{alone} at runtime). For every application, we calculate the mean over all of its instances in our multiprogrammed workloads, which is shown in Figure 5. Applications are sorted according to network intensity from left to right in ascending order. As NAS is the first work to estimate NoC-level application slowdown, we are unable to compare it with any other prior work.

We make two key observations from Figure 5. First, the slowdown estimation error is consistently low for all applications, with an average error of only 2.6% (or 4.2%) for a 4×4 (or 8×8) NoC. The maximum error observed is for *GemsFDTD* in an 8×8 NoC, at 31.7%. *GemsFDTD* is very network-intensive and easily induces network saturation. In such a scenario, interference results not only from other applications, but also from the large number of network requests issued by a GemsFDTD instance that compete with each other (i.e., *intra*-application interference). This leads to higher inaccuracy, as NAS primarily focuses on estimating *inter*-application interference, not intra-application interference. Second, the accuracy of the model remains high with a larger NoC that caters to high-intensity workloads (a setting where inter-application interference is expected to be pronounced). For example, the estimated error for high-intensity workloads is 6.6% in a 4×4 network, and this rises only slightly to 7.6% in an 8×8 network.

We also study the distribution of the estimated error across all 5,760 application instances (i.e., each of the individual programs in our workload bundles) from our 90 multiprogrammed workloads for the 8×8 network, as illustrated in Figure 6. The figure bins the NAS slowdown estimation error, and depicts on the y-axis the fraction of application instances whose NAS slowdown estimation error is less than the x-axis value but did not fall into an earlier bin. The error is lower than 10% for 66.0% of all application instances, and lower than 20% for 84.3% of them. Only 5.6% of the application instances have an error of 40% or more. Overall, we conclude that NAS estimates application slowdowns caused by NoC-level inter-application interference with high accuracy.

Hardware Complexity. NAS incurs slight overhead in both the control and data paths. However, as the area of a NoC is dominated by the data path [4, 10], we discuss hardware complexity on only the data path at each router, as well as the additional buffers needed at each NI and core, as shown in Table 2. NAS widens the data path of a conventional virtual channel router by 5.3%, as each flit needs to carry an 8-bit field to track Δt_{flit}, compared with the baseline design that has a 128-bit link for payload and a 24-bit link for routing information.

Table 2. Hardware cost of NAS for each network node.

Location	Component	Hardware Cost
Router	Δt_{flit}	5.3% wider data path
NI	$T_{first\_arrival}$ and $T_{last\_arrival}$	(16+16)×16 bits
	Inheritance table (Figure 4)	(6+4+8)×20 bits
Core	$\Delta t_{request}$	8 bits
	$T_{critical\_shared}$	16 bits
	Δt_{stall}	16 bits
Total Cost of NAS Per Node		**114 bytes + 5.3% router area**

For each NI, we add a 16-bit $T_{first\_arrival}$ and a 16-bit $T_{last\_arrival}$ field to each MSHR entry to track the arrival times of the first and the last flits of a packet. As discussed in Section 3.4, each NI also needs an inheritance table to pass the delay (Δt_{packet}) from the control packet to its associated data packet. Each inheritance table entry contains a 6-bit *reqID*, a 4-bit *mshrID*, and an 8-bit Δt_{packet} for each incoming control packet that requires an LLC access. Assuming that each router has 4 ports (excluding the local port), and that an LLC access takes 5 cycles, there are at most 20 control packets within the router at any given time. Thus, the table needs only 20 entries.

For each core, three registers are needed to record $\Delta t_{request}$, $T_{critical\_shared}$, and Δt_{stall}. Note that the size of each register is determined by considering the worst-case interference that could happen in the network. For example, Δt_{stall} is bounded by the epoch length, where 16 bits can cover the worst case (which is unlikely).

Overall, NAS requires 114 bytes of buffers at each node, with a 5.3% area overhead at each router. We conclude that the total cost of NAS is very modest.

6.2. Fairness Improvement with FAST

We evaluate FAST using the metrics of *unfairness* and *weighed speedup* on a 4×4 and an 8×8 system. *Unfairness* is simply the largest application slowdown (i.e., maximum slowdown [1,11–13,17,33,34]), and *weighed speedup* [18, 59] is calculated as:

$$\textit{weighted speedup} = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{aloneActual}} \quad (11)$$

[6] Using a perfect shared LLC allows us to focus on the interference in the NoC. However, NAS does not depend on such a configuration, and can also model systems with LLC misses to main memory. NAS can also be combined with other slowdown models for caches and main memory (e.g., ASM [64]) to capture the slowdown effects of interference throughout the system.

[7] We believe our mechanisms can be extended to take into account inter-thread dependencies in multithreaded workloads, using principles similar to prior works [15, 27, 28, 65]. We leave the design of such extensions and the evaluation of multithreaded workloads to future work.

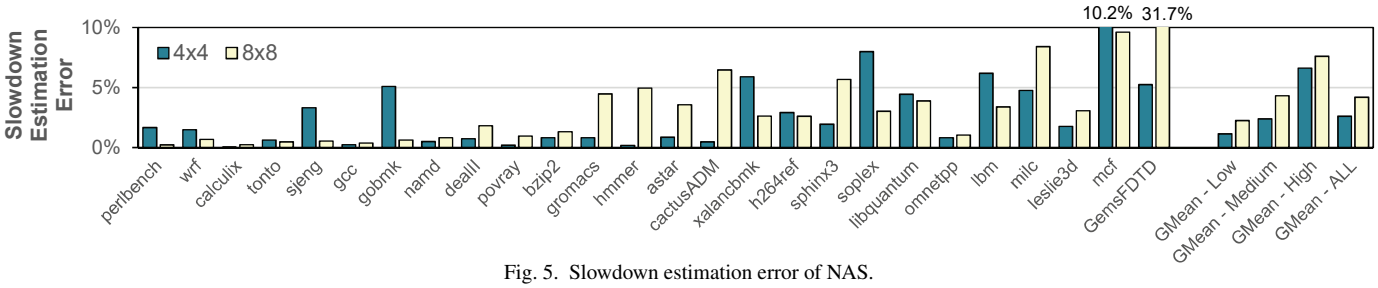


Fig. 5. Slowdown estimation error of NAS.

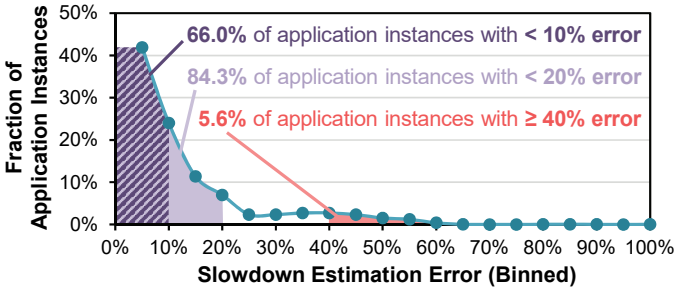
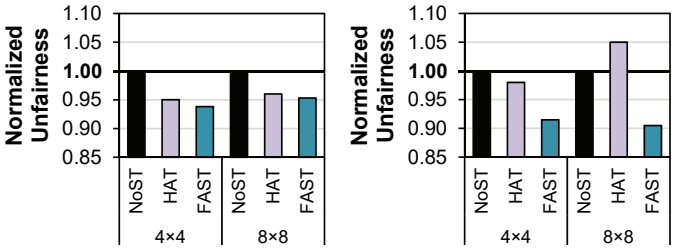


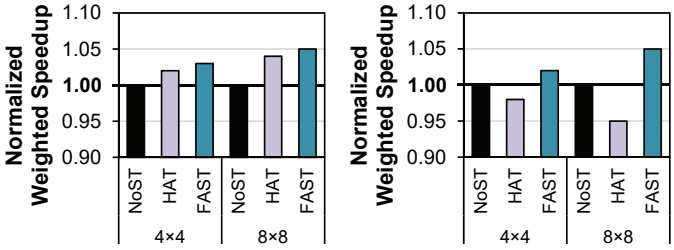
Fig. 6. Slowdown estimation error distribution for the 8×8 mesh network, with each point on the x-axis representing application instances whose error falls into a bin between $X-5\%$ and $X\%$.

where N is the number of cores in the system. As interference occurs more often in a network with higher load, we use only the Mixed and Heavy workloads (see Section 5) to evaluate FAST. We compare FAST with a baseline design without source throttling (denoted as *NoST*), and with HAT [7], a state-of-the-art source throttling design. Figure 7 shows the unfairness of the three mechanisms on the multiprogrammed workloads, and Figure 8 shows the weighted speedup. Values are normalized to the baseline NoST design.



(a) Mixed workloads (b) Heavy workloads

Fig. 7. Average unfairness, normalized to NoST (lower is better).



(a) Mixed workloads (b) Heavy workloads

Fig. 8. Average performance, normalized to NoST (higher is better).

Parameters. For FAST, we set the *epoch length* to 100K cycles. *NumThrottleDown* and *NumThrottleUp* are set to 4 for a 4×4 system and to 16 for an 8×8 system. *MPCThreshold* is set to 0.04 and 0.06 for Mixed and Heavy workloads, respectively. For HAT, we also set the *epoch length* to be 100K cycles, and set target network utilization at 5%. We find that HAT is very sensitive to *NonIntensiveCap*, as it determines the aggressiveness of throttling. We empirically find and select the best value of *NonIntensiveCap* for each workload category, for each network topology.

FAST vs. NoST. FAST improves both fairness and performance compared to the NoST baseline for all workloads. This fairness improvement occurs because throttling reduces network congestion and inter-application interference, allowing FAST to reduce the average packet delivery latency by 15.4%, which in turn accelerates the execution of slower applications. In fact, FAST provides higher fairness improvements over NoST for Heavy workloads in a larger network. Particularly, for an 8×8 NoC, FAST reduces unfairness by 9.5% on average (up to 17.6%) and improves performance by 5.2% on average (up to 7.5%). For Heavy workloads, inter-application interference occurs more frequently since the network is more congested, making FAST more effective. As each request needs to travel a longer distance in a larger network, interference becomes more pronounced, again making FAST more effective. We conclude that FAST provides the highest improvements over NoST in large networks with high memory intensity workloads.

FAST vs. HAT. FAST has much lower unfairness and higher performance than HAT. As Figures 7 and 8 show, HAT is less effective for Heavy workloads. In particular, FAST has 14.5% lower unfairness and 10.5% higher weighted speedup, on average, than HAT for an 8×8 network when running Heavy workloads. In fact, HAT performs even worse than NoST due to two reasons. First, unlike FAST, which takes application slowdown into account, HAT throttles applications with higher network intensity. If an application has both the highest network intensity and the maximum slowdown, throttling this application leads to greater unfairness. Second, HAT does not explicitly consider the negative effects of throttling on slowdown, whereas FAST proactively throttles down only those applications where the negative impact of throttling is minimal.

Multithreaded applications are expected to incur greater interference among their own threads due to additional coherence and synchronization traffic. Although we do not evaluate NAS or FAST on multithreaded applications, we believe that both NAS and FAST can be adapted to work with them, by incorporating some of the principles used by prior work [15, 27, 28, 65]. We leave the detailed study of multithreaded applications as future work.

Overall, we conclude that, thanks to the accurate slowdown estimates provided by NAS, FAST improves system fairness and performance over prior source throttling mechanisms.

7. Related Work

This work proposes the first application slowdown estimation model for on-chip networks, NAS, and makes use of slowdown estimates provided by NAS to drive a new fairness mechanism, FAST, to achieve higher system fairness and performance. We already discussed prior works on slowdown models, none of which account for NoC-level interference, in Section 2.1. NAS can be used in conjunction with other slowdown models (e.g., ASM [64]) to devise a mechanism that considers interference in the NoC, caches, and memory holistically. While prior work [53] has developed a model to estimate NoC latency, it does not solve the problem that we address, as we must capture the effects of inter-application interference at runtime.

Prior work on improving NoC fairness relies on quality-of-service heuristics to guide source throttling. Both HAT [7] and Nychis et al. [51, 52] throttle down applications with high network intensity (as determined by MPKI in HAT, and instructions per flit in Nychis et al.). HAT optimizes for network utilization, while Nychis et al. reduce the starvation rate (an indicator of network congestion). Unlike both of these works, NAS does not rely on network traffic heuristics; instead, it directly models the impact of interference on application slowdown

(and thus unfairness). Furthermore, NAS is a general performance model, and can be used for a wide variety of purposes.

Many works [2, 3, 8, 11–13, 19, 20, 24, 25, 34, 39, 40, 42, 43, 51, 52] propose NoC prioritization and thread/data mapping mechanisms to improve system fairness and performance. As our work is complementary to the underlying prioritization policy and thread/data mapping techniques, NAS and FAST can be seamlessly employed together with these mechanisms to further improve system performance and fairness.

8. Conclusion

An application executing on a NoC-based multicore system may suffer from significant interference due to shared resource contention from concurrently-executing applications, and hence experience slowdown. The degree of slowdown for each application can differ greatly, depending on both an application's sensitivity to NoC contention and the network traffic induced by other applications. As a result, there can be a large amount of unfairness in the system, with some applications slowing down much more than others, which can lead to poor quality-of-service and system performance degradation. We propose the NoC Application Slowdown (NAS) Model, which efficiently estimates the per-application slowdown at runtime in hardware. Building upon the runtime slowdown predictions provided by NAS, we propose Fairness-Aware Source Throttling (FAST), an example quality-of-service mechanism that utilizes estimates from NAS to determine how to control the network injection rates of different applications, and thereby reduce unfairness and improve system utilization. We believe that NAS can be used for many other purposes, and that its ability to accurately predict the effects of inter-application interference on application performance can enable the design of more predictable and controllable networks in the future.

Acknowledgments

We thank the anonymous reviewers and the SAFARI Research Group for their feedback. In particular, we thank Rachata Ausavarungnirun for his helpful discussions in the initial stages of this project. This research was partially supported by the NSF (grants 1212962 and 1423302), Intel, and VMware.

References

[1] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[2] R. Ausavarungnirun *et al.*, "Design and Evaluation of Hierarchical Rings with Deflection Routing," in *SBAC-PAD*, 2014.
[3] R. Ausavarungnirun *et al.*, "A Case for Hierarchical Rings with Deflection Routing: An Energy-Efficient On-Chip Communication Substrate," *PARCO*, 2016.
[4] J. Balfour and W. J. Dally, "Design Tradeoffs for Tiled CMP On-Chip Networks," in *ICS*, 2006.
[5] K. D. Bois *et al.*, "Per-Thread Cycle Accounting in Multicore Processors," *ACM TACO*, 2013.
[6] K. K. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[7] K. K. Chang *et al.*, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks," in *SBAC-PAD*, 2012.
[8] C.-L. Chou and R. Marculescu, "Contention-Aware Application Mapping for Network-on-Chip Communication Architectures," in *ICCD*, 2008.
[9] Y. Chou *et al.*, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *ISCA*, 2004.
[10] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2003.
[11] R. Das *et al.*, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.
[12] R. Das *et al.*, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.
[13] R. Das *et al.*, "Aergia: Exploiting Packet Latency Slack in On-Chip Networks," in *ISCA*, 2010.
[14] B. K. Daya *et al.*, "SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-Network Ordering," in *ISCA*, 2014.
[15] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
[16] E. Ebrahimi *et al.*, "Prefetch-Aware Shared Resource Management for Multi-Core Systems," in *ISCA*, 2011.
[17] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," in *ASPLOS*, 2010.
[18] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
[19] C. Fallin *et al.*, "MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect," in *NOCS*, 2012.
[20] C. Fallin *et al.*, "CHIPPER: A Low-Complexity Bufferless Deflection Router," in *HPCA*, 2011.
[21] M. Fattah *et al.*, "A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips," in *NOCS*, 2015.
[22] S. Ghose *et al.*, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in *ISCA*, 2013.
[23] A. Glew, "MLP Yes! ILP No!" in *ASPLOS WACI*, 1998.
[24] B. Grot *et al.*, "Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees," in *ISCA*, 2011.
[25] B. Grot *et al.*, "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-Effective QOS Scheme for Networks-on-Chip," in *MICRO*, 2009.
[26] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *CAN*, 2006.
[27] J. A. Joao *et al.*, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *ASPLOS*, 2012.
[28] J. A. Joao *et al.*, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *ISCA*, 2013.
[29] C. Kim *et al.*, "An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches," in *ASPLOS*, 2002.
[30] H. Kim *et al.*, "Bounding Memory Interference Delay in COTS-based Multi-Core Systems," in *RTAS*, 2014.
[31] H. Kim *et al.*, "Bounding and Reducing Memory Interference Delay in COTS-Based Multi-Core Systems," *RTS*, 2016.
[32] S. Kim *et al.*, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *PACT*, 2004.
[33] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[34] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[35] N. Kırman *et al.*, "Checkpointed Early Load Retirement," in *HPCA*, 2005.
[36] P. Kongetira *et al.*, "Niagara: A 32-Way Multithreaded SPARC Processor," in *MICRO*, 2005.
[37] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," in *ISCA*, 1981.
[38] C. J. Lee *et al.*, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
[39] J. W. Lee *et al.*, "Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks," in *ISCA*, 2008.
[40] A. K. Mishra *et al.*, "A Heterogeneous Multiple Network-on-Chip Design: An Application-Aware Approach," in *DAC*, 2013.
[41] T. Moscibroda and M. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
[42] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-Chip Networks," in *ISCA*, 2009.
[43] S. Muralidhara *et al.*, "Reducing Memory Interference in Multi-Core Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
[44] O. Mutlu *et al.*, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[45] O. Mutlu *et al.*, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.
[46] O. Mutlu *et al.*, "Runahead Execution: An Effective Alternative to Large Instruction Windows," *IEEE Micro*, 2003.
[47] O. Mutlu *et al.*, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.
[48] O. Mutlu *et al.*, "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," *IEEE Micro*, 2006.
[49] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[50] "NOCulator," https://github.com/CMU-SAFARI/NOCulator.
[51] G. Nychis *et al.*, "On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-Core Interconnects," in *SIGCOMM*, 2012.
[52] G. Nychis *et al.*, "Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?" in *HotNets*, 2010.
[53] M. Papamichael *et al.*, "FIST: A Fast, Lightweight, FPGA-Friendly Packet Latency Estimator for NoC Modeling in Full-System Simulations," in *NOCS*, 2011.
[54] H. Patil *et al.*, "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation," in *MICRO*, 2004.
[55] M. K. Qureshi *et al.*, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.
[56] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[57] V. Seshadri *et al.*, "The Dirty-Block Index," in *ISCA*, 2014.
[58] V. Seshadri *et al.*, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT*, 2012.
[59] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.
[60] S. T. Srinivasan and A. R. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors," in *MICRO*, 1998.
[61] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
[62] L. Subramanian *et al.*, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.
[63] L. Subramanian *et al.*, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
[64] L. Subramanian *et al.*, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
[65] M. A. Suleman *et al.*, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," in *ASPLOS*, 2009.
[66] M. Thottethodi *et al.*, "Self-Tuned Congestion Control for Multiprocessor Networks," in *HPCA*, 2001.
[67] H. Usui *et al.*, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM TACO*, 2016.
[68] H. Vandierendonck and A. Seznec, "Fairness Metrics for Multi-Threaded Processors," *IEEE CAL*, 2011.
[69] D. Wentzlaff *et al.*, "On-Chip Interconnect Architecture on the Tile Processor," in *MICRO*, 2007.
[70] J. Zhao *et al.*, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.