Mutually Aware Prefetcher and On-Chip Network Designs for Multi-Cores

Junghoon Lee, Hanjoon Kim, Minjeong Shin, John Kim, and Jaehyuk Huh

Abstract—Hardware prefetching has become an essential technique in high performance processors to hide long external memory latencies. In multi-core architectures with cores communicating through a shared on-chip network, traffic generated by the prefetchers can account for up to 60% of the total on-chip network traffic. However, the distinct characteristics of prefetch traffic have not been considered in on-chip network design. In addition, prefetchers have been oblivious to the network congestion. In this work, we investigate the interactions between prefetchers and on-chip networks, exploiting the synergy of these two components in multi-cores. Firstly, we explore the design space of *prefetch-aware* on-chip networks. Considering the difference between prefetch and non-prefetch packets, we propose a priority-based router design, which selects non-prefetch packets first over prefetch packets. Secondly, we investigate *network-aware* prefetcher designs. We propose a prefetch control mechanism sensitive to network congestion—throttling prefetch requests based on the current network congestion. Our evaluation with full system simulations shows that the combination of the proposed prefetch-aware router and congestion-sensitive prefetch control improves the performance of benchmark applications by 11–12% with out-of-order cores, and 21–22% with SMT cores on average, up to 37% on some of the workloads.

Index Terms—Computer architecture, on-chip networks, flow controls, muti-cores, hardware prfetcher, memory hierarchies

1 Introduction

As external memory latencies have been orders of magnitude longer than on-chip cache latencies, hardware prefetching techniques have been widely used in microprocessors to hide long off-chip memory latencies. Recent high performance processors often include one or more hardware prefetchers to predict different data access patterns. Such hardware prefetching techniques, in essence, speculatively bring cachelines to the on-chip caches or separate buffers, before memory instructions require the data during their executions [30], [27], [33], [29].

Meanwhile, as the number of cores in processors increase, on-chip networks with high bandwidth have emerged to replace traditional buses or dedicated wires [11]. The on-chip network is a resource shared by the cores and prefetchers in the memory hierarchy. There have been many studies to improve the overall available bandwidth of such interconnection networks and to make the networks resilient for temporary congestion on certain paths [19], [13].

However, there has been very little study on the interaction between these two components–the prefetcher and the on-chip network. Prior studies to improve on-chip networks have not considered the *distinct characteristics* of network traffic generated by prefetchers. Unlike request and data traffic for instruction execution, prefetch traffic is essentially speculative and can be useless if prediction is incorrect. Even for successfully predicted prefetch requests, the data can be actually used hundreds or thousands cycles after the initiation of prefetch requests. Even though there have been several studies that have looked at prioritized traffic or prioritized arbitration and its impact on on-chip network performance [3], [22], [7], [28], no prior work have investigated the impact of prioritized arbitration when considering prefetch traffic and its impact on overall performance.

In addition to the current prefetch oblivious network designs, conventional prefetchers are also insensitive to *the status of networks*. Prefetchers increase on-chip traffic significantly with prefetch requests and data, but they do not consider the negative effect of the increased on-chip network traffic. On-chip network is a shared resource among the cores on a chip, and the congestion caused by a core can affect the performance of other cores significantly. When the network is congested, speculative prefetches can degrade overall performance by further increasing traffic. Although Ebrahimi et al. investigated the effect of prefetching on other shared resources such as memory bandwidth [16], the importance of mutual awareness between prefetchers and on-chip networks has been largely neglected so far.

In this paper, we investigate how the two components, on-chip networks and hardware prefetchers, affect each other, and explore the design space of mutually aware networks and prefetchers. To the best of our knowledge, this is one of the first studies to investigate the impact of prefetching on on-chip networks and evaluate the interactions between on-chip networks and prefetchers. First of all, we study network designs partitioned for two distinct traffics generated by cores and prefetchers. Our results show that partitioning network

- *J. Lee, H. Kim, J. Kim, and J. Huh are with the Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 335 Gwahak-ro (373-1 Guseong-dong), Yuseong-gu, Daejeon 305-701, Republic of Korea. E-mail: {junghoon.lee, hanj, jjk12, jhhuh}@kaist.ac.kr*
- *J. Kim is also with the Division of Web Science and Technology, Korea Advanced Institute of Science and Technology (KAIST), 335 Gwahak-ro (373-1 Guseong-dong), Yuseong-gu, Daejeon 305-701, Republic of Korea.*
- *M. Shin is with LG Electronics Inc., LG Twin Towers 20, Yoido-dong, Youngdungpo-gu, Seoul 150-721, Korea. E-mail: minjeong.shin@lge.com*

Manuscript received 03 June 2012; revised 04 Jan. 2013; accepted 16 Apr. 2013. Date of publication 25 Apr. 2013; date of current version 07 Aug. 2014. Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.99

resources to avoid interference between the two types of traffic can actually *degrade* the overall performance.

Instead of partitioning network resources, we propose prefetch-aware on-chip network. We first propose a simple source-level prioritization (SP) of demand requests over pending prefetch requests. In addition, we propose a priority-based arbitration mechanism for routers, called *Traffic-aware Prioritized Arbiter (TPA)*, which provides a higher priority to non-prefetch packets compared with prefetch packets. The proposed mechanisms exploit the available *slack* between when the prefetch is issued and when the actual prefetch data is accessed by the core. Prioritization with SP and TPA reduces the network latency for non-prefetch packets, which is more critical for performance than prefetch packets.

To complement the prefetch-aware network design, we propose a prefetch control mechanism, which adjusts the prefetch traffic based on the status of networks. The mechanism, called *Traffic-aware Prefetch Throttling (TPT)* throttles prefetch generations at the hardware prefetchers, depending on network congestion. Unlike the conventional prefetchers, which issue prefetch requests without considering network congestion, the mechanism identifies congested paths and dynamically adjusts the aggressiveness of prefetchers. Our results show that combination of prefetch-aware on-chip network and network-aware prefetcher can improve overall performance by up to 37%.

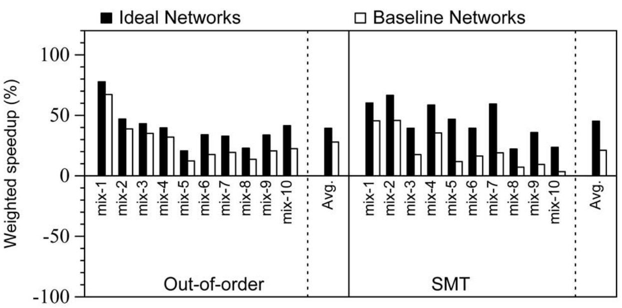
We evaluate the proposed mechanisms with a 16-core multi-core model simulated with a full system simulator. To assess the effectiveness of our designs with different network requirements, we use two core models, single-threaded out-of-order and dual-threaded simultaneous multithreading (SMT) cores, and two cache organizations, a NUCA-style shared L2, and private L2s with a shared L3.

In the rest of this paper, we first show the impact of prefetching on interconnection networks in Section 2. In Section 3, we propose prioritization techniques including prefetch-aware routers. In Section 4, we describe prefetch throttling mechanisms, which consider network congestion. We present prior work on prefetching and interconnection networks in Section 6, and conclude the paper in Section 7.

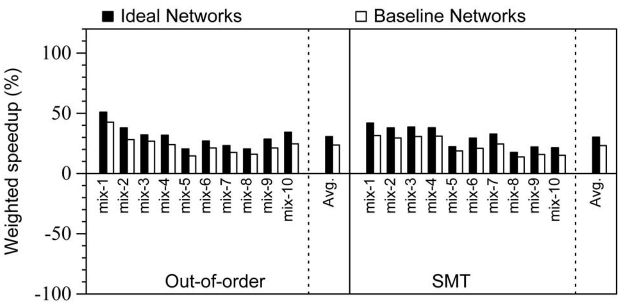
2 MOTIVATION

2.1 Prefetching Techniques

Hardware prefetchers predict data access patterns, and issue memory requests speculatively. In this paper, we use a commonly used prefetching technique based on stream detection, but the proposed techniques in this paper can be used with any other types of prefetchers. A stream prefetcher prefetches non-unit stride cache lines by dynamically tracing streams [27], [33]. When a cache miss occurs, the prefetcher creates a stream entry with a limited training window. If a subsequent miss occurs within the limited window, the stream entry is trained with the address difference between the current miss address and the prior address stored in the stream entry. Once a stream entry is trained, the stream prefetcher generates one or more prefetch requests. The number of prefetch requests generated by a trigger of the stream prefetcher is called the *degree* of prefetch. With multiple stream entries, a stream prefetcher can detect multiple streams simultaneously. The stream prefetcher used in this paper has the degree of four



(a) **Shared-L2** with stream-4 prefetchers



(b) **Shared-L3** with stream-4 prefetchers

Fig. 1. Speedups with prefetching: ideal vs. baseline.

[33]. In addition to the base stream prefetcher, we also use a combined prefetcher, which extends a stream prefetcher with a next line prefetching [29], [30]. The next line prefetcher simply issues memory requests for the next block address from the current miss address. The combined prefetcher generates prefetch requests more aggressively than the stream prefetcher.

The stream and combined prefetchers used in this paper show better performance on all workloads, compared to without the prefetchers (Fig. 1). In addition, Table 3 shows the accuracy and coverage of prefetchers with each application for the performance benefit.

2.2 Methodology

For the experimental results, we use the Simics full system simulator with the GEMS timing model [23], [24] and the Garnet on-chip network model [4]. We use two core models with different traffic behaviors in our evaluation–a 4-way out-of-order core (OoO), and dual-threaded simultaneous multithreading (SMT) as described in Table 1.

We use two shared cache models, `shared–L2` and `shared–L3`, to evaluate the proposed mechanism in cache architectures with different traffic behaviors. In `shared–L2`, the L2 cache is a shared L2 with 16 1 MB banks, with addresses statically mapped to each bank. For the cache coherence among L1 caches, the model uses a directory-based protocol. In `shared–L3`, the L2 cache is a private 256KB L2 for each core, and the L3 cache is a shared L3 with 16 2 MB banks. In general, `shared–L2` generates more traffic than `shared–L3`, as in `shared–L3`, private L2s filter out requests out of each tile. For the memory, we use 4 memory controllers at each corner of a chip and a detail memory model as DDR3 shown in Table 1.

For the baseline on-chip network model, we use a 4×4 2D mesh topology. The other detailed configurations are shown in Table 1. The two prefetchers we use in our evaluation are a

TABLE 1
System Configuration

Parameter	Value	
Processor	16 out-of-order cores (4 ways, 128 ROB size, dual-threaded SMT)	
	Shared-L2	**Shared-L3**
L1 cache	32KB, 4 ways, 2 cycles	32KB, 4 ways, 3 cycles
L2 cache	1024KB, 8 ways, 12 cycles	256KB, 8 ways, 11 cycles
L3 cache		2048KB, 8 ways, 39 cycles
Block size	64B block	64B block
Coherence	directory-based coherence, MOSI protocol	
Memory	4 Memory controllers Samsung DDR3-1866 M378B1G73BH0-CF8 [1] (tCL=13.91, tRCD=13.91, tRP=13.91) 933MHz bus cycle, 8B-wide data bus 8 banks/rank, 2 ranks/dimm	
Network	4x4 mesh with 8B channels, 4 VCs 4 cycle router pipeline, 6 buffers/VC dimension-ordered routing(DOR)	
Prefetcher	stream prefetcher with 32 streams (degree of 4, distance of 64) [33] next-line prefetch (next 1 line)	

stream prefetcher with a stream degree of four (stream–4) and a combined prefetcher which adds next-line prefetching to the stream-4 prefetcher (combined). In shared–L2, prefetchers are trained by L1 miss requests, and issue prefetch requests to the shared L2 cache. Prefetched data fill the requesting L1 cache. In shared–L3, prefetched data fill only the requesting private L2 cache, not to corrupt the L1. Due to space constraint, we focus on the results from stream–4, except for the performance results of proposed techniques, but results from combined showed similar results.

We selected our application mixes to represent both various prefetch effects and traffic amounts, as shown in Table 2. For the OoO model, 16 applications are used with four instances of each application in mix-1 to mix-4, with two instances of each application in mix-5 to mix-8, and with one instance of each application in mix-9 and mix-10. For the SMT model, 32 applications are used with a double of total applications employed on a workload in the OoO model. In addition, we pinned two different applications, an odd and an even index applications, on a core for the SMT model.

We denote *prefetch packets* as the request and data packets for prefetch requests and data responses. Packets other than prefetch packets in the on-chip networks are classified as *demand packets*, which include request and data response packets. The weighted speedup metric is used as the performance metric which represents the average speedup from each applications compared to a baseline system with prefetching enabled.

2.3 The Effect of Prefetching on Networks

In this section, we show how much network traffic prefetching generates, and how the limited network bandwidth affects the effectiveness of prefetching. We also show slack exists between the time when prefetch is issued and the time when the subsequent demand access to the prefetched cache-lines occurs.

Prefetch vs. Demand Packets: The prefetch traffic accounts for a significant portion of the overall traffic in on-chip networks. Table 4 shows the ratios of prefetch packets over the total packets. In shared–L2, prefetch packets account for 49% of the total packets with OoO cores. Using SMT cores increases the ratio of prefetch packets further, to 56% with stream–4. In shared–L3, prefetch packets account for 31% and 33% for OoO and SMT cores respectively. These results show that a significant portion of on-chip network traffic comes from the prefetcher and the impact of prefetch traffic on overall performance needs to be properly considered.

TABLE 2
Application Configurations

Workloads	Applications ( ): The number of applications
mix-1	sphinx3(4), soplex(4), xalancbmk(4), GemsFDTD(4)
mix-2	omnetpp(4), xalancbmk(4), GemsFDTD(4), milc(4)
mix-3	mcf(4), libquantum(4), omnetpp(4), lbm(4)
mix-4	sphinx3(4), soplex(4), omnetpp(4), milc(4)
mix-5	sphinx3(2), gromacs(2), lbm(2), hmmer(2), omnetpp(2), cactusADM(2), povray(2), tonto(2)
mix-6	bzip2(2), sphinx3(2), dealII(2), hmmer(2), astar(2), milc(2), cactusADM(2), povray(2)
mix-7	gobak(2), bzip2(2), hmmer(2), astar(2), milc(2), cactusADM(2), tonto(2), lbm(2)
mix-8	libquantum(2), omnetpp(2), mcf(2), namd(2), cactusADM(2), tonto(2), gromacs(2), lbm(2)
mix-9	bzip2, sphinx3, perlbench, libquantum, omnetpp, mcf, povray, hmmer, astar, milc, cactusADM, tonto, dealII, namd, lbm, gromacs
mix-10	gobak, bzip2, namd, gromacs, dealII, omnetpp, mcf, hmmer, perlbench, libquantum, milc, cactusADM, povray, tonto, lbm, sphinx3

TABLE 3
Prefetch Accuracy and Coverage on Shared-L2 with Stream-4 Prefetcher

Application	Prefetch Accuracy (%)	Prefetch Coverage (%)
bzip2	28.39	18.47
sphinx	35.08	28.21
deal	38.69	17.84
hmmer	50.81	28.62
astar	33.67	5.47
milc	73.97	20.24
cactusADM	4.86	3.63
povray	8.42	11.41
gromacs	5.21	2.8
lbm	75.37	40.67
omnetpp	7.21	6.3
tonto	11.43	15.84
libquantum	99.99	60.02
mcf	3.24	2.95
namd	26.47	20.64
gobak	14.28	12.82
perlbench	7.41	4.63
xalancbmk	35.49	15.53
gemsFDTD	89.19	69.21
soplex	43.8	22.19

TABLE 4
The Ratios of Prefetch Packets Over Total Packets

Workloads	shared-L2		shared-L3	
	Out-of-order	SMT	Out-of-order	SMT
mix-1	51.94%	55.49%	35.01%	38.97%
mix-2	48.49%	60.51%	33.13%	37.12%
mix-3	49.21%	57.33%	32.34%	40.42%
mix-4	49.84%	59.15%	34.32%	37.80%
mix-5	55.46%	59.32%	32.57%	30.83%
mix-6	57.28%	54.58%	28.02%	26.17%
mix-7	47.61%	58.11%	29.56%	29.40%
mix-8	43.23%	52.31%	28.21%	35.07%
mix-9	44.90%	51.72%	26.76%	25.81%
mix-10	46.08%	52.68%	31.72%	30.07%
avg.	49.41%	56.12%	31.16%	33.16%

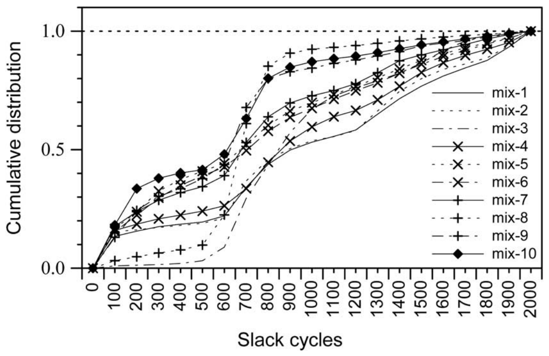


Fig. 2. Cumulative distributions of prefetch slack cycles with out-of-order cores.

The Effect of Limited Network Bandwidth: Since prefetchers generate a significant amount of additional traffic, limited network bandwidth affects the effectiveness of prefetchers. Fig. 1 presents the speedup of using prefetchers with a baseline network and an *ideal* network. An ideal network has the same zero-load latency as the baseline 2D-mesh network but assume a fully-connected topology with sufficient link bandwidth to inject an entire packet in a single cycle. Each bar represents the speedup of an execution with prefetching on ideal and baseline network compared to an execution without prefetching on baseline network.

With the ideal network, prefetching improves overall performance for both OoO and SMT cores. On average, in `shared–L2`, the `stream–4` prefetcher can improve the overall performance by 39% for out-of-order cores, and 45% for SMT cores. However, with the baseline network model, the performance improvements *drop* significantly. With out-of-order cores, the performance improvement from the stream-4 prefetcher is reduced by approximately 30%. With SMT cores, the negative effect of limited network bandwidth is much more severe on the performance improvements by prefetching. The results indicate the effectiveness of prefetchers is not only dependent upon the capability of prefetchers, but also the available network bandwidth which must transfer prefetch packets effectively without delaying critical demand packets.

Time Slack for Accessing Prefetched Data: One of the distinct characteristics of prefetch traffic is that packet latency may not be as critical as demand traffic, since prefetched data are not used by the core immediately. The cumulative distribution of the *slack* is shown in Fig. 2, which represents the time between the initiation of a prefetch request and the data access from the core for the prefetched data.

Fig. 2 shows that the majority of prefetches have more than 200 cycle time slack until the prefetch data are used by the core, although each mix may have a different slack distribution. The slack represent how much a prefetch request can be delayed without affecting the overall performance–thus, provide an opportunity to reduce the negative effect of prefetch packets on the overall performance by prioritizing demand packets over prefetch packets.

2.4 Partitioned Networks for Prefetch

Resource partitioning can potentially eliminate possible negative interferences between prefetch and demand traffics. In this section, we explore partitioned networks where prefetch traffic is differentiated from demand traffic. In the baseline architecture, we assume a mesh network where the network resources–mainly buffers (virtual channels (VCs) [12]) and channels–are shared between the prefetch and demand packets.

We evaluate the following prefetch-aware partitioned network architectures in this section:

- **Partitioned VC:** We isolate the buffer resources (or VCs) between the two types of traffic while still sharing the channel resource (Fig. 3(b)). The VCs in each router are dedicated to either prefetch packets or demand packets while the channel bandwidth is shared between the two types of packets.
- **Partitioned Network:** Instead of partitioning the VCs, the channel bandwidth can be partitioned. The two types of traffic are isolated by *channel slicing* [11] or creating two separate networks as shown in Fig. 3(a). Prefetch packets are only allowed to traverse in one network while demand packets utilize the other network. Compared to the baseline network, we maintain the same bisection bandwidth to provide a fair comparison.

In Fig. 4, we compare the performance of partitioned virtual channels and networks against the baseline on-chip network with prefetching enabled. VC($x : y$) in Fig. 4(a) denotes the partitioned virtual channels with x VCs used by demand traffic and y VCs used prefetch traffic. In VC(2:2) and VC(3:1), the total number of VCs is the same as the baseline (4 VCs), but the VCs are partitioned to 2:2 or 3:1 ratio. In VC(x,y) where $x = v$ and v is the total number of VCs, the partitioned VCs are shared buffer organization as y VCs are shared between deand and prefetch traffic while $x - y$ VCs are dedicated for demand traffic. VC(2:2) exhibits a slightly lower performance than the baseline, but VC(3:1) and VC(4:1) result

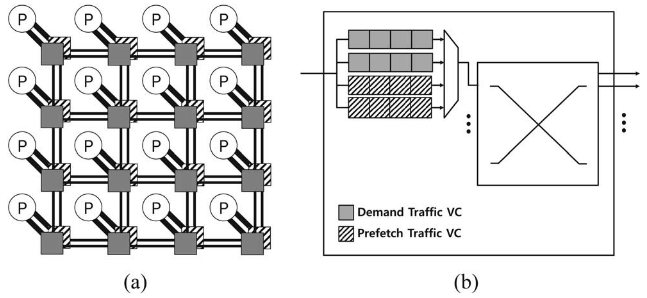
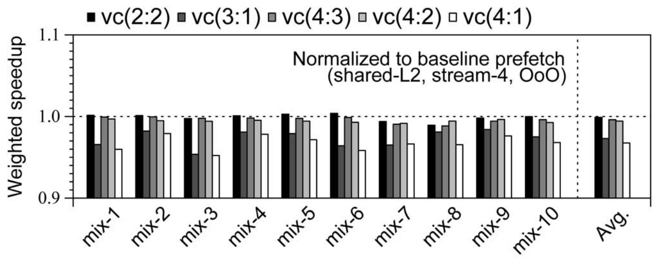
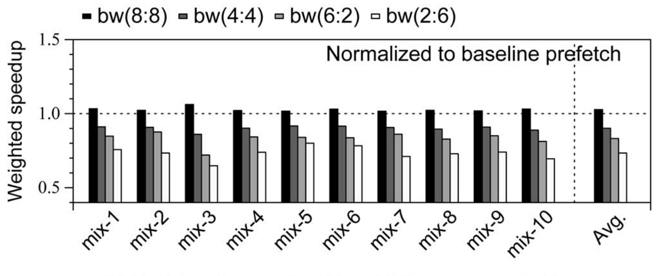


Fig. 3. (a) Partitioned network. (b) Partitioned virtual channel router microarchitecture.

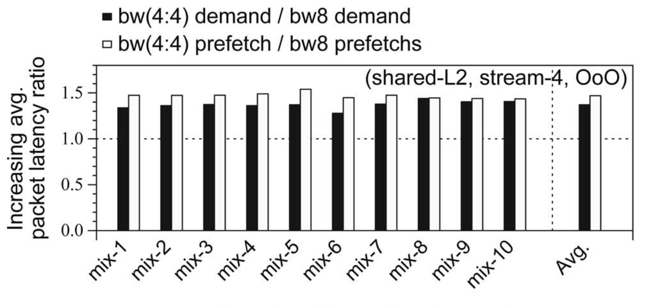


(a) Partitioned virtual channels

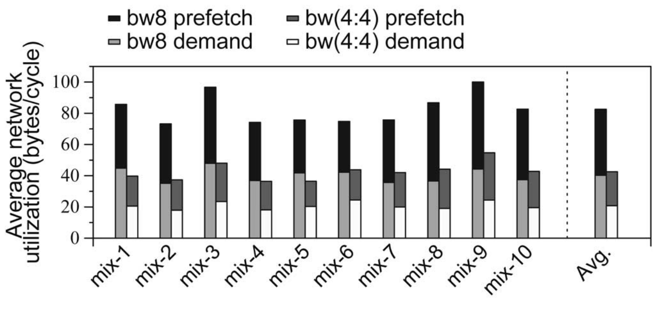


(b) Partitioned networks (shared-L2, stream-4, OoO)

Fig. 4. Performance of partitioned networks.



(a) Avg. latencies with partitioned network



(b) Network utilization with partitioned network

Fig. 5. Avg. latency & util. with partitioned networks.

in performance losses by 3–4% on average. Since the prefetch packets account for one or two thirds of the total packets as shown in Table 4, providing only one virtual channel for prefetch traffic results in non-negligible performance losses. Limiting VCs for prefetch to 3, 2, or 1 channels even if the channels for demand traffic is fixed to 4, also results in performance losses. These results indicate that bursts of traffic from prefetch or demand requests may often do not occur simultaneously, and thus sharing virtual channels and being able to use the entire VCs for bursty prefetch traffic provide better performance than partitioning VCs.

Fig. 4(b) presents the performance comparison of various partitioned network configurations. In the figure, BW($x:y$) stands for partitioned networks with xB channel bandwidth for demand traffic and yB for prefetch traffic. Note that BW(8:8) has twice bandwidth than the baseline network with 8B link (BW8). Partitioning networks for demand and prefetch results in much worse performance than the baseline unified networks with the same link bandwidth, losing 10% performance with BW(4:4), 17% with BW(6:2), and 27% with BW(2:6). As expected, reducing the channel bandwidth for demand traffic further reduces the overall performance. Prior work [37], [32] has partitioned on-chip networks into multiple networks to separate different types of traffic. However, our results show that providing such partitioning across prefetch and non-prefetch packets can significantly degrade overall performance.

In Fig. 5(a), we compare the ratio of prefetch and demand packet latencies for BW(4,4) and the baseline networks (BW8). The results show how both prefetch and demand packet increase in latency with partitioned network (BW(4,4)), compared with BW8, some of which is caused by the additional serialization latency of the narrower network. Fig. 5(b) shows the utilization of networks for BW(4:4) and the baseline (BW8). The figure shows that partitioning networks results in lower utilizations of links than the unified networks.

In this section, we evaluated various fixed partitioning schemes for bandwidth and VCs, but the results showed that such fixed partitioning of networks or VCs across the two types of traffic did not provide any performance benefit compared with the baseline architecture. Based on this observation, we focus on improving the network performance by prioritizing packets over the same unified networks in the next section.

3 Prefetch-Aware Network

As discussed in Section 2.4, partitioning networks or virtual channels for demand or prefetch traffic leads to performance losses. In this section, instead of partitioning networks which cannot be resilient for bursty traffic of both types, we use prioritization based on packet type, while the networks resources are shared. We first propose a simple source-level prioritization, which can send demand packets first even though older prefetch packets are pending. In addition, we propose traffic-aware prioritized arbitration (TPA), which prioritizes packets in the routers. Finally, we also evaluate two possible further optimizations on TPA, multi-level TPA and dynamic priority boosting.

3.1 Source-Level Prioritization (SP)

The most simple prioritization mechanism for demand requests over prefetch requests is to re-order the requests at the source node. Request packets can be pending at the source node, if the router of the source node can no longer accept a new packet. As soon as the router has a free buffer, source-level prioritization (SP) enforces demand packets to be injected to the router even if older prefetch packets exist. This mechanism does not require any change in the network and requires only a minor change in the request selection logic of the cache controller.

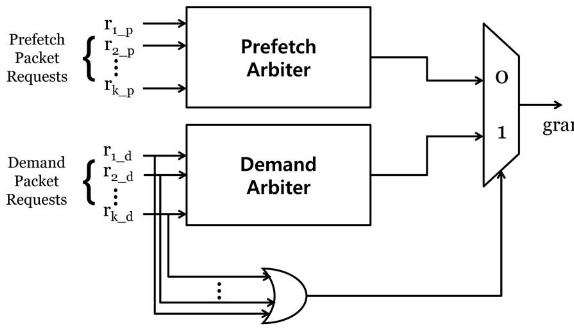


Fig. 6. TPA arbiter microarchitecture.

3.2 Traffic-aware Prioritized Arbiter (TPA)

The second technique prioritizes packets during transmission through the on-chip networks. Instead of traffic-oblivious arbitration such as round-robin arbitration or age-based arbitration, we propose traffic-aware prioritized arbitration (TPA) where demand packets are prioritized over prefetch packets. To provide support for TPA, each packet injected into the network is classified as either a demand packet or a prefetch packet. An additional bit in the packet header is used to differentiate between the two types of packets. For multi-flit[1] packets such as cache line data packets, each flit does not need to be appended with this information but only the head flit. The packet classification can be stored in the state information associated with each virtual channel–similar to how routing information is only maintained by the head flit.

In our baseline on-chip network, two-phase round-robin arbitration is used–arbitration is first done among all the virtual channels at the input and then, arbitration is done at the output port among the input ports, similar to iSLIP [25] algorithm. With TPA, round-robin arbitration is first done among all the demand packets. If there are no demand packets, then round-robin arbitration is done among the prefetch packets. To minimize the impact of arbitration on the critical paths, the arbiter can be duplicated as shown in Fig. 6 with a demand packet arbiter and a prefetch packet arbiter. If there are one or more demand packets, the output of the demand arbiter is used; otherwise, the result of the prefetch arbiter is used. Prior work has shown that the area and power impact of arbiter in on-chip network is small [35], [28]–thus, adding an extra arbiter has minimal impact on the cost of an on-chip network router. To prevent priority inversion (i.e., demand packets being buffered behind prefetch packets), we only allocate a VC to a new packet once the previous packet has departed the downstream router, when a *tail*-credit is received [11].

With fixed prioritized arbitration, fairness can be an issue as demand packet always receives priority over prefetch packets and starve prefetch packets. A techniques can be used to prevent starvation–i.e., create a threshold (t) and if a prefetch packet has not been serviced for t cycles, the prefetch packet is *upgraded* and receive priority. In addition, because of the slack available in the prefetch packets as described earlier in Section 2, the increase in prefetch latency does not negatively impact overall performance.

1. Packets in on-chip networks are often partitioned into one or more flits [11], a unit of flow control.

3.3 Further Optimizations on TPA

In addition to SP and TPA, we evaluate two schemes for further optimizations. However, as shown in Section 3.4, the following two logical extensions to TPA and SP, do not provide performance benefits enough to justify the added complexity. We report the negative results as part of our design space exploration.

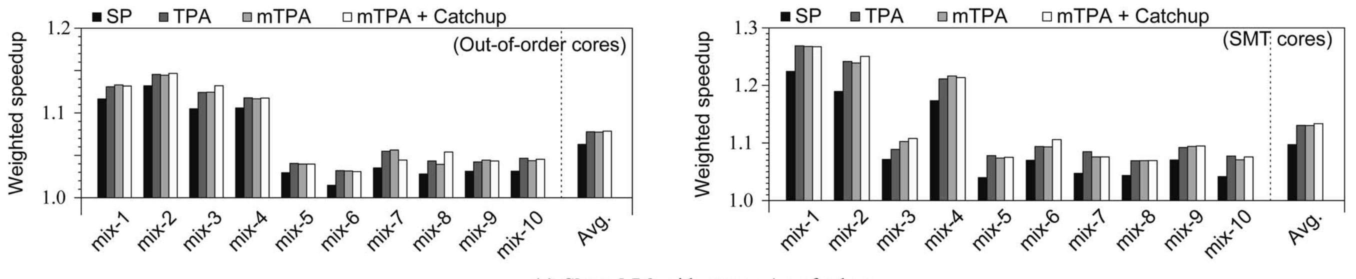
Multi-level TPA (mTPA): The first optimization for TPA is to support multiple priorities for prefetch traffic. Multi-level TPA divides prefetch priority into two levels, urgent prefetch and less urgent prefetch packets. As proposed by Das et al. [14], the different behaviors of applications affect the criticality of network packets for the applications, and thus, some applications can benefit from prioritization at routers. Similarly, for prefetch traffic, to find the best parameter to determine the priority level for prefetch traffic, we have evaluated several parameters, time slack from prefetch to demand, L1 miss per 1K instructions, and L2 miss per 1K instructions. Among the three parameters, using L1 miss per 1K instructions results in the best performance, which is consistent with the finding from Das et al.

Dynamic Priority Boosting (catch-up): The second optimization for TPA is to boost the priority of packets dynamically. When the slack from a prefetch issue to a demand access to the same cacheline is short, the demand access may occur while the prefetch packet is in the networks, or waiting for the data response from the memory. In such cases, the prefetch packets must be returned as soon as possible, with the same priority as demand traffic. We evaluate the possible benefit of such dynamic priority boosting (*catch-up*) with an ideal implementation. In the ideal implementation, as soon as a demand miss occurs on the same cacheline as an outstanding prefetch request, subsequent packets serving the prefetch are upgraded to demand packets.

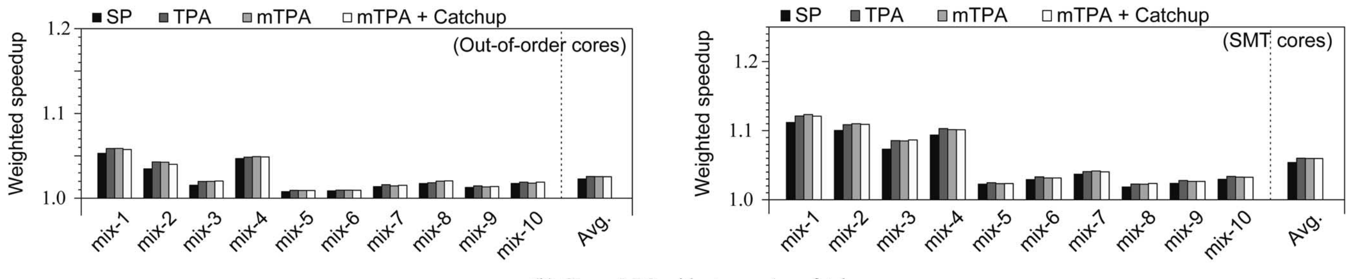
3.4 Results

Fig. 7 presents the weighted speedups normalized to the baseline for four configurations, `SP`, `+TPA`, `+mTPA`, and `+mTPA+catchup`. Each configuration includes the previous configurations. For example, `+TPA` includes `SP`.

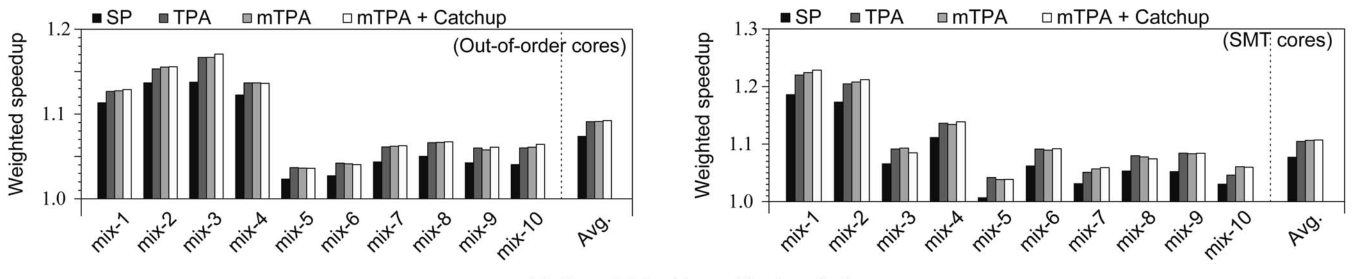
Firstly, source-level prioritization (SP) provides significant performance improvements, even if it requires no change in the networks, but only a minor change in the cache controller. For the out-of-order, SP improves the baseline in performance by 6% and 2% for `shared–L2` and `shared–L3` with *stream-4* prefetcher, and by 7% and 4% with *combined* prefetcher. TPA further improves the performance significantly. In the case of using *stream-4* prefetcher, TPA with SP improves the baseline in performance by 7% for the Out-of-order configuration and 13% for the SMT configuration in `shared–L2`, and by 3% and 6% in `shared–L3`. Also, in the case of using *combined* prefetcher, TPA with SP also improves the baseline by 9%, 10% for Out-of-order and SMT with `shared–L2`, and by 5% and 10% for ones with `shared–L3`. The performance improvements by TPA with SP over the baseline can be as high as 25% for some of mixes (mix-1 and mix-2) in SMT. Generally, the benefit of TPA is higher in the SMT than in the Out-of-order core model, since the prefechers operated by multi-threads in SMT core model generate much more traffic than in Out-of-order core model. Also, TPA can get more benefit of



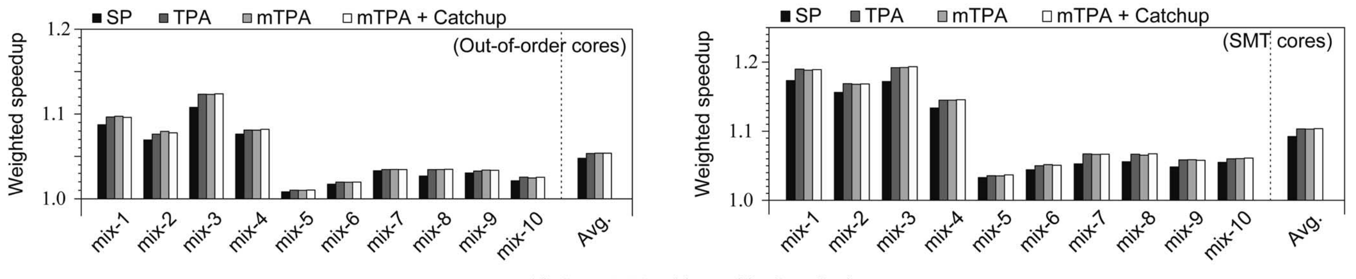
(a) **Shared-L2** with stream-4 prefetchers



(b) **Shared-L3** with stream-4 prefetchers



(c) **Shared-L2** with combined prefetchers



(d) **Shared-L3** with combined prefetchers

Fig. 7. Performance with prefetch-aware networks.

performance in shared–L2 relatively where volume of traffic movements is high, than shared–L3. Especially, mix-1 and mix-2 with using *stream-4* prefetcher have high performance gap between in shared–L2 and in shared–L3.

Multi-level TPA (mTPA) does not provide much of improvement in overall performance. However, the mTPA improves several applications in both of the out-of-order core and the SMT core models with stream-4 and combined prefetcher. Dynamic priority boosting (catchup) also improves the overall performance slightly by 1–2% compared to TPA.

The two optimizations, although they may improve performance for some applications, generally provide relatively low performance benefits. Table 5 shows that there is not much volume of in-flight prefetch packets cores request in on-chip networks. This indicates that the catchup mechanism has little chance to boost prefetch packets, and can not improve performance in spite of our ideal implementation.

Source-level prioritization (SP) improves the overall performance significantly with very low extra hardware costs. TPA with a modest increase of complexity in routers provides further improvements. However, the relatively low performance improvements with mTPA and dynamic priority boosting may not justify extra hardware and design complexity to support them.

4 Network-Aware Prefetchers

In this section, we investigate network-aware prefetch designs, which can adjust the aggressiveness of prefetchers based on the levels of network congestion. We propose traffic-aware prefetch throttling (TPT), which detects network congestion dynamically, and throttles prefetch requests, if the networks become congested. TPT tracks congested paths on networks, and if generated prefetch requests must go through

TABLE 5
Ratio of Catch-Up Packets in Total Packets (Shared-L2, Stream-4)

Workloads	mix-1	mix-2	mix-3	mix-4	mix-5	mix-6	mix-7	mix-8	mix-9	mix-10	Avg.
Out-of-order	1.9%	1.3%	1.9%	1.9%	1.2%	1.1%	1.2%	1.4%	1.3%	1.9%	1.5%
SMT	2.2%	1.1%	2.5%	3.1%	1.7%	3.1%	4.9%	3.4%	1.2%	9.0%	3.2%

one of the congested paths, the requests are throttled. To detect congested paths, the throttling mechanism has a congestion tracking component at each node to trace the network congestion level to every destination node. Section 4.1 describes the congestion detection mechanism used in this paper. Section 4.2 proposes network-aware throttling mechanisms for prefetchers. Section 4.3 presents the experimental results for network-aware prefetchers.

4.1 Estimating Network Congestion

To support traffic-aware prefetch throttling (TPT), each node tracks the congestion status of paths to the other nodes. TPT measures congestion status between a source and destination pair, instead of tracking congestion status at each link. A *congested path* from a source to a destination is a path in networks which has much longer packet latencies than the unloaded latency from the same source to destination pair. Such congested paths can change dynamically as application phases change and even the operating system change the scheduling of applications, migrating a thread to a different physical core.

In this paper, we use a straightforward mechanism to measure the congestion status for each path. Each packet carries a time-stamp which is set when the packet leaves the source node. Checking the time-stamp in packets, the receiving node can determine the transfer latency for each packet. If the latency of the arrived packet is much larger than the unloaded latency from the source, we designate the path as a congested one.

Fig. 8 shows the cumulative distribution of normalized packet latencies against the unloaded latency. For example, 2.0 in the x-axis represents that the packet latency is twice longer than the unloaded latency. For the 10 benchmark mixes, the distributions are very similar. About 60% of the total packets arrive at destinations with a latency less than twice of the unloaded latency. We also use a separate network simulation with random patterns to confirm that the network saturates when the average latency is doubled. Based on the observations, we use a multiplier of two from the unloaded latency as the threshold to determine a congested path.

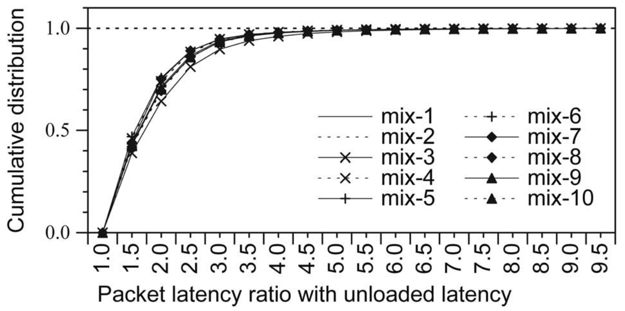


Fig. 8. Packet latency distribution.

Fig. 9 presents average latencies with increasing injection rates, measured from a separate network simulator modeling the same 4x4 mesh networks with a random access pattern. The dotted line is when the average latency is twice of the unloaded latency. The figure shows that the dotted line (2x unloaded latency) marks an injection rate just before the networks are saturated.

4.2 Throttling Prefetch Requests

Fig. 10 depicts the overall architecture of TPT. For each core, there is an N-bit vector (*congestion status vector*) to mark whether each path to another node is congested or not. N is the total number of nodes in the networks. When the prefetcher generates a prefetch, it checks whether the path to the target node is congested. If the path to the destination is congested, the prefetch request is throttled. The congestion status for each destination can be set in various ways, and we evaluated several policies to set the entry.

For a prefetch request, there are four different paths which may be involved during the process of the request, as shown in Fig. 10. (1) The prefetch request is transferred to the corresponding L2 bank node. (2) If the request is an L2 miss, it is transferred to the corresponding memory controller node. (3) The data response is transferred to the L2 bank. (4) The data

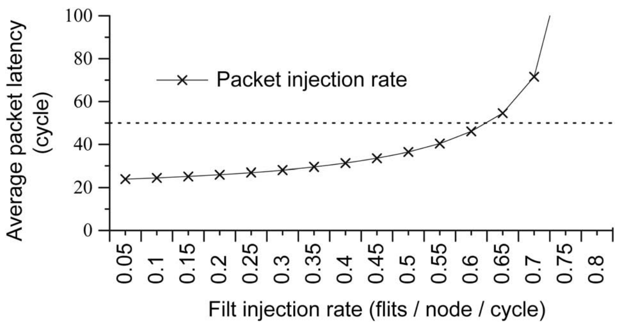


Fig. 9. Packet latency analysis.

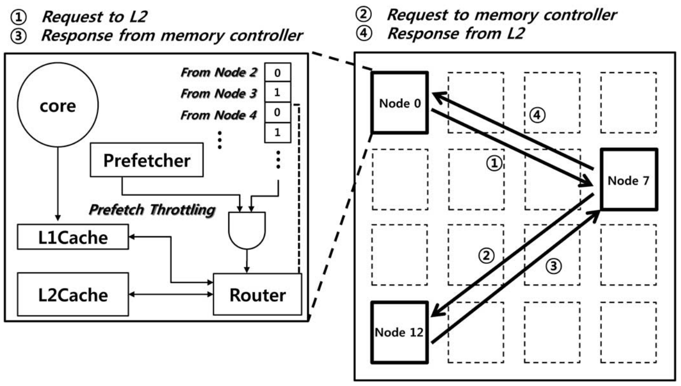
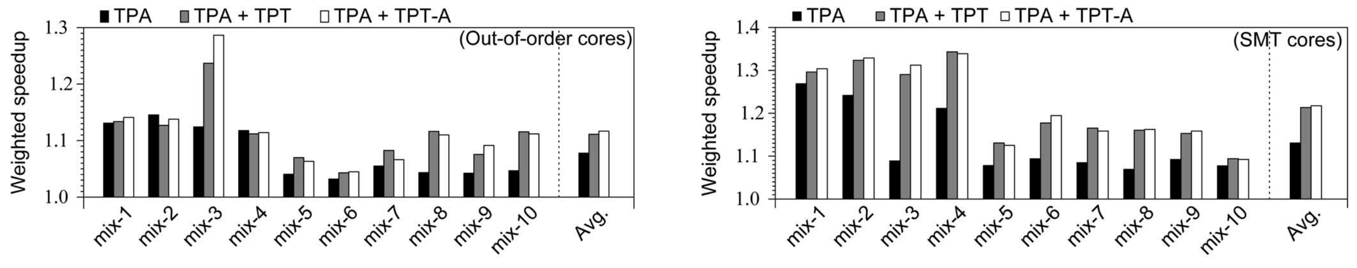
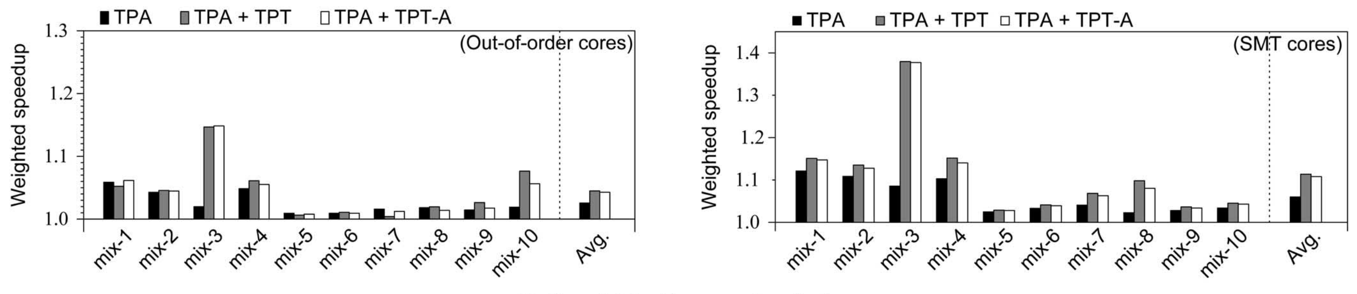


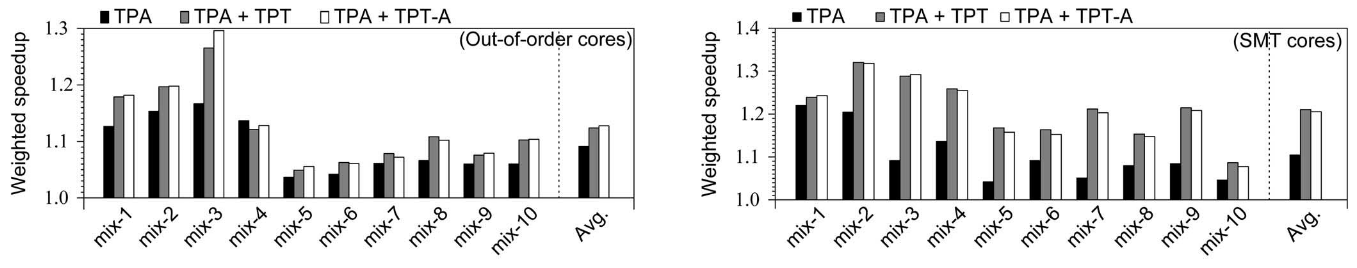
Fig. 10. TPT architecture.



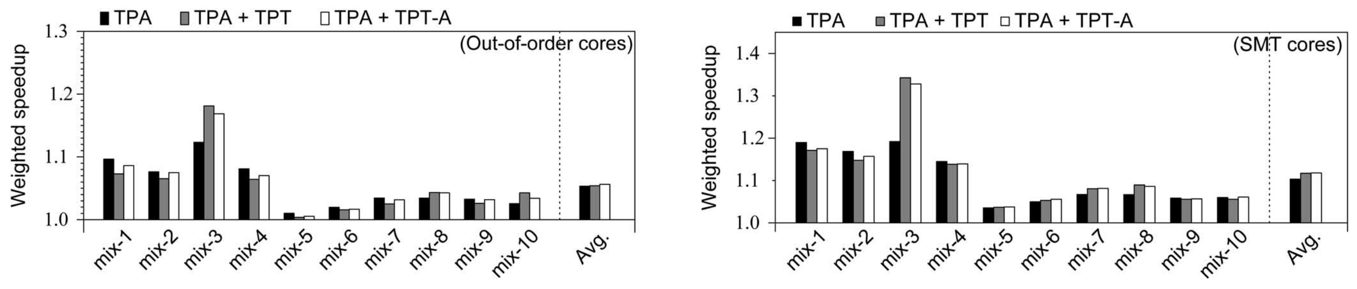
(a) **Shared-L2** with stream-4 prefetchers



(b) **Shared-L3** with stream-4 prefetchers



(c) **Shared-L2** with combined prefetchers



(d) **Shared-L3** with combined prefetchers

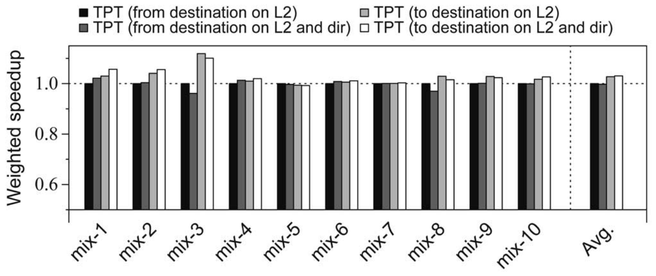
Fig. 11. Performance with network-aware prefetchers.

response is transferred to the requesting core. We have evaluated various combinations of path congestion status to set the congestion status entry for each node.

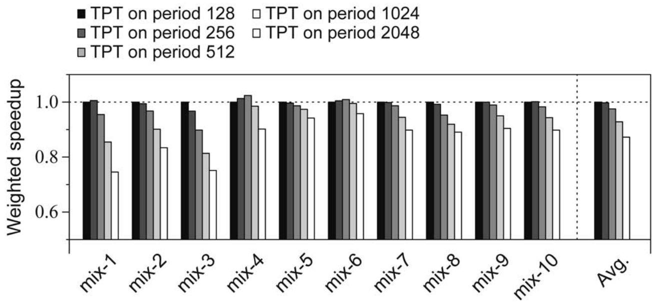
Fig. 12(a) shows the performance of TPT with four different policies. To find the best policy, for the experiments, we assume that the core node is ideally aware of the actual packet latencies for all paths, without any extra overhead. Also, the core node is assumed to know whether a request will be a hit or miss in the L2. The first policy sets the congestion entry to an L2 bank node, if the path from the L2 bank node to the core node is congested (path (4)). The second policy sets the entry to an L2 bank, if both paths, from the L2 bank node to the core node, and from the memory controller node to the L2 node(if the recent request is a L2 miss), are congested (path (3) and (4)). The third policy sets the entry, if the path from the core to the L2 node is congested (path (1)). The fourth policy sets the entry, if both of the paths from L2 node to the memory node, and the core to the L2 node are congested (path (1) and (2)).

The results shows the performance is not very sensitive to the different policies, except for mix-3, but using the both of paths from the core to the L2 node and from the L2 node to the memory controller node (path(3) and path(4)) result in the best performance on average. Even though considering the both paths has the best performance, we choose the first policy (path(4)) for detecting congestion path becuase collecting that path information requires the least overhead that the core node just needs to check the latencies of packets arriving from each L2 bank node. In the rest of paper, each node updates its congestion status vector by checking the latencies of packets arriving from the other nodes.

The network congestion status for each path will change dynamically. To use the latest congestion information,

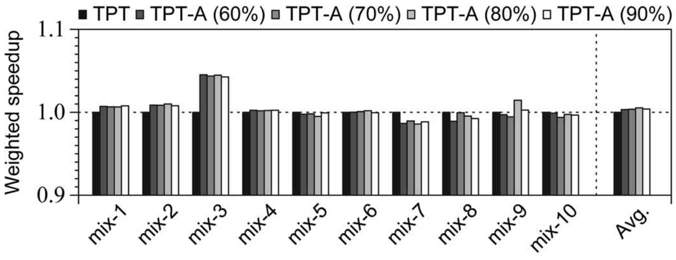


(a) Path estimation (shared-L2, stream-4, OoO core)

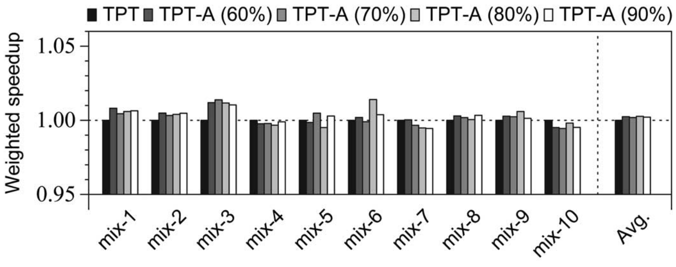


(b) Reset period (shared-L2, stream-4, OoO core)

Fig. 12. Selecting TPT parameters.



(a) Out-of-order cores (shared-L2, stream-4)



(b) SMT cores (shared-L2, stream-4)

Fig. 13. TPT-A with various accuracy thresholds.

TPT resets the congestion status vector in each core periodically. We have evaluated various reset periods from 128 cycles to 2048 cycles. Fig. 12(b) shows the performance of TPT with various reset periods. As shown in the figure, the shortest 128 cycles results in the best performance. This result indicates that congestion status changes in relatively short periods, and TPT must track such short-term fluctuations of network congestion. Using the best reset period, we reset the congestion status vector every 128 cycles, for the experiments in the rest of this paper. During the reset period, the status entry for a destination node is set only when at least a packet arrives from the node, and the latency is twice longer than the unloaded latency of the path.

Traffic-aware prefetch throttling (TPT) stops issuing a prefetch request when the prefetch request should be sent to a destination, to which the path is currently marked as a congested path. The rationale behind TPT is that speculative prefetches may degrade the overall performance by injecting extra traffic when the networks are already saturated for the paths. If a path becomes congested, prefetch requests, which are less critical than demand packets, are throttled by the prefetchers.

However, the base TPT mechanism considers only the network status, but does not distinguish how useful prefetchers are. TPT with accuracy (TPT-A) may potentially enhance TPT by determining the prefetch throttling decision not only by the network congestion, but also with the accuracy of prefetchers. To implement TPT-A, we assume an extra bit in the tag to mark a prefetched block, and if the data are used by the core, the bit is cleared. By tracking the number of prefetches issued, and the number of the first use of prefetched blocks, we approximately measure the current accuracy of prefetchers. With TPT-A, if the prefetch accuracy of the core node is higher than a threshold, the prefetch request will be sent to the L2 bank node, even if the path is congested.

4.3 Results

In this section, we evaluate the throttling mechanisms combined with TPA. Fig. 11 presents the weighted speedups normalized to the baseline. The figure shows three bars, TPA only, TPA+TPT, and TPA+TPT-A configurations. For the TPT-A configuration, we have evaluated various accuracy thresholds, and used the best one.

TPT improves TPA significantly for both out-of-order cores and SMT cores. For out-of-order cores, on average, TPT improves TPA by 11% and 4% for `shared–L2` and `shared–L3` with *stream-4* prefetcher, and by 12% and 5% for ones with *combined* prefetcher respectively. In essence, TPT becomes much more effective with SMT cores, as SMT cores generate significantly more prefetch traffic than out-of-order cores. In SMT cores, TPT improves TPA by 21% and 11% for `shared–L2` and `shared–L3` with *stream-4* prefetcher, and also by 21% and 11% for ones with *combined* prefetcher.

TPT-A exhibits relatively low performance improvements over TPT. Fig. 13 presents the performance of TPT-A with various accuracy thresholds.

In Fig. 13(a), based on Out-of-order cores with the stream-4 prefetchers, TPT-A improves TPT only for a small subset of mixes (mix-1, mix-2, mix-3 and mix-9), and slightly degrades TPT on the others (mix-7, mix-10). The result is consistent with the result in Fig. 11. Similar to in Out-of-order cores, TPT-A also shows slight performance improvements for several mixes (mix-1, mix-2, mix-3 and mix-6) in SMT cores, as shown in Fig. 13(b). In addition, TPT-A can complement TPT when TPT has worse performance than TPA in a certain case, due to a conservative threshold to throttle prefetch requests, such as the case of mix-2 in `shared–L2` and Out-of-order core with stream-4 prefetcher. Such large improvements for some mixes (mix-3 and mix-9 in `shared–L2` and Out-of-order core with stream-4 prefetcher) show the potential benefits of fine-tuning TPT by considering other factors, such as accuracy and congestion levels. Optimizations of TPT with other parameters will be our future work.

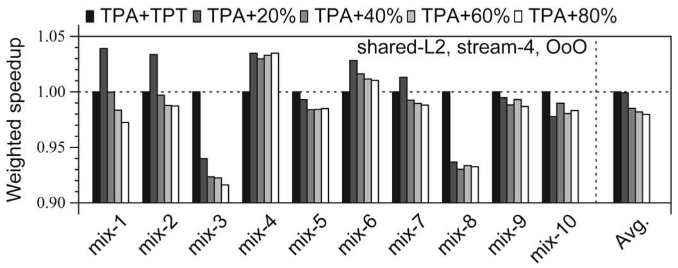


Fig. 14. TPT vs. accuracy-only throttling.

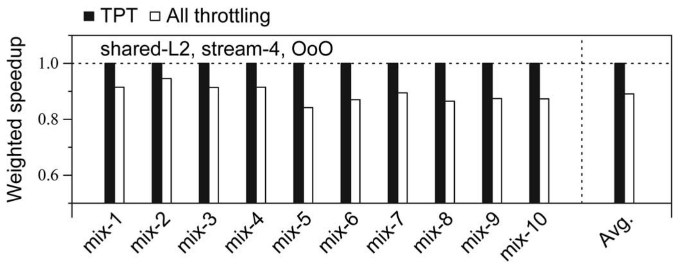


Fig. 15. TPT vs. All request throttling.

Comparing TPT to network-unaware throttling: Prefetchers can throttle prefetch requests only by the current accuracy without considering the network status. In such network-unaware throttling, prefetchers are throttled only by locally measured accuracy information. Fig. 14 presents the performance with the accuracy-only based throttling with various accuracy thresholds. The figure indicates that network-aware throttling by TPT significantly out-performs throttling based only on the accuracy of prefetching for most of the mixes. The result indicates that the aggressiveness of prefetchers must be sensitive to traffic on the networks. Throttling traffic without distinguishing demand and prefetch packets has a significant negative impact. Fig. 15 compares TPT with a prefetch-unaware throttling. The prefetch-unaware throttling uses the same threshold as TPT, but throttle both demand and prefetch requests. As shown in the Fig. 15, throttling both requests solely based on network status can degrade performance.

5 Summary

Performance: In summary, Fig. 16 presents the average speedups with the techniques discussed in this paper. SP provides consistent improvements for both of the out-of-order and SMT cores. TPA is also effective for both cores. The effectiveness of TPT is much higher in SMT cores than out-of-order cores, as networks tend to be congested more often in the SMT configuration. In case of using stream-4 prefetcher, a combination of SP, TPA, and TPT can improve the baseline in performance by 10% in the Out-of-order model, and 21% in the SMT model for `shared–L2`, and by 4% and 11% in each core model for `shared–L3`. Also, the combination can improve the baseline in performance 11% in the Out-of-order model, and 21% in the SMT model for `shared–L2`, and by 5% and 11% in each core model for `shared–L3` in the case of using combined prefetcher.

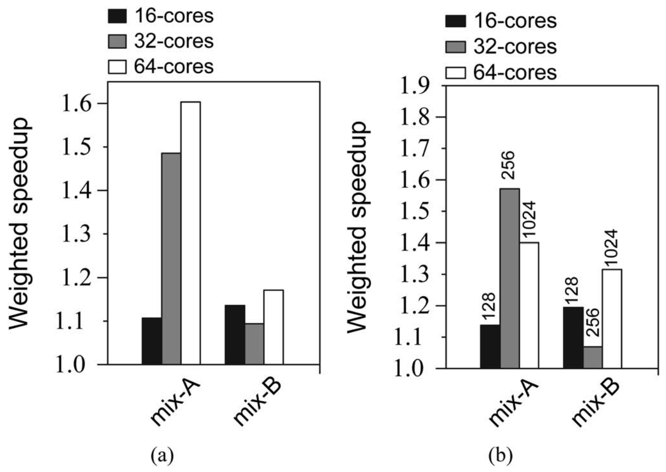


Fig. 17. (a) TPA Performance in each core scale. (b) A change amount of performance according to the period for measuing congesion paths in TPT.

Scalability: In addition, we evaluate the scalability of the techniques discussed in this paper. For the scalability evaluation, we had extended the number of cores to 32 and 64 cores, and presented the TPA's performance with the weighted speedup normalized to the baseline performance in Fig. 17. Fig. 17(a) presents the TPA's performance with 16, 32, and 64 cores, showing the trend that TPA can improve performance with more cores. TPA in larger scale on-chip networks have a better chance to improve performance, due to more routers with a prioritized arbitration which packets should pass through.

Fig. 17(b) presents performance changes with TPT by 16, 32, and 64 cores. Each bar shows the performance with the best period of measuring the congestion path for each core count. Even with 32 and 64 cores, TPT can effectively throttle prefetches, although the best period may change depending on the size of networks. The number on top of each bar shows the best period for each core count. In general, with larger networks, the best period tends to increase, although each mix may have a slightly different optimal period. We expect that

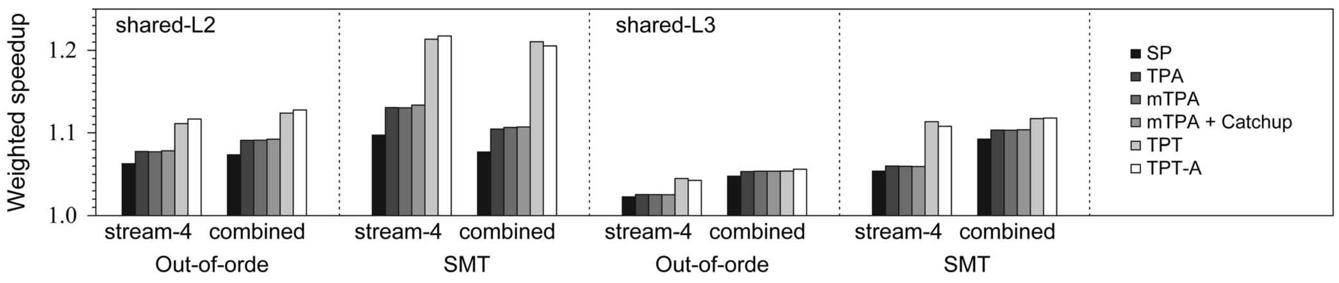


Fig. 16. The performance summary of mutually aware prefetchers and networks.

TABLE 6
Hardware Cost Estimation

TPA	Components	Cost(μm^2)	TPT	Components	Cost(μm^2)
Arbiter	Arbiter for prefetch:1 (from Orion 2.0 45nm technology)	56258.4	Store	1-bit 16 entry 1D flip flop register file:1 AND gate for grant:1	112 1
Control	AND gate, NOT gate for control signal	300	Compare	6-bit 16 entry 1D flip flop register file:1 16-bit subtractor for latency:1 6-bit comparator:1, Decoder:1	229 40 60
	Total	56558.4		**Total**	442

Per router, total chip area is $140\ \mathrm{mm}^2$: means the name of component and the number of component.

the threshold cycles must be adjusted, depending on the size of networks, which will be our future work.

Hardware cost: Table 6 summarizes hardware area cost for our proposed mechanism. TPA needs additional arbiter for prefetch arbitration, and some extra logic gates for control signal. On the other hands, TPT needs two register files for decision of throttling prefetch request and storing saturation threshold. In addition, TPT needs a comparator and a subtractor for comparing packet latency to saturation threshold, and a decoder for decoding source information at packet header. Based on our estimation, the area overhead of TPA and TPT are approximately 0.040% and 0.0003%, over a node and when compared with the total chip area, the area overhead is very negligible. For this estimation, we use Orion [36] for synthesis of TPA arbiter, and compute area overhead based on 2009 ITRS roadmap [2].

6 Related Work

As one of the most important techniques to hide long memory latencies, numerous studies proposed and evaluated prefetch techniques [20], [30], [27], [33], [5], [6], [29].

Recent studies discussed the design of prefetchers considering the limited memory bandwidth and interference in multi-core architecture. Lee et al. propose a scheduling mechanism for usefulness of prefetch request in DRAM controller [21]. Srinath et al. explore the effect of memory bandwidth contention on the effectiveness of prefetching, and propose adaptive prefetcher designs, which adjust the aggressiveness of prefetchers [31]. To reduce the negative impact of interferences among prefetchers in CMPs, Ebrahimi et al. propose a global coordination mechanism to adjust the aggressiveness of multiple prefetchers for the limited memory bandwidth [17]. These prior work have recognized the importance of prefetching and the impact on shared resource but have not considered the impact of the on-chip network shared resource.

Various prioritized arbitration in on-chip networks have been proposed. For example, Cheng et al. improve the performance and power efficiency of cache coherence protocols considering characteristics by using multiple networks [10]. Bolotin et al. prioritize different types of coherence messages to send critical messages before less critical ones [9]. Other studies have investigated the criticality difference among packets by observing application behaviors. Das et al. exploit the different behaviors of applications sharing on-chip networks in multi-cores, and propose the prioritization of packets based on the time criticality of memory requests from applications with different parallelisms [14], [15]. Li et al. proposed to provide priority to more critical packets in the network based on protocol information [22]. In addition, there has been other prioritized arbiters proposed [11], [3]. Unfortunately, they proposed new on-chip networks design for the distinct characteristics of requests, but they dose not directly consider a charateristics of prefetch on on-chip networks.

Avoiding network congestion through injection throttling based on global or local knowledge has been studied. Thottethodi et al. proposed throttling packets based on global network congestion [34] while simple implementation to throttle based only on using local informations [8] has also be proposed. Other injection throttling include injection control mechanism through prediction of congestion [26] and regional congestion awareness (RCA) for load-balancing [18]. These prior work commonly use injection throttling to prevent networks congestion, but did not consider the impact of different traffic type in making the throttling decision. they did not show the benefit to throttle less useful prefetch packets comparing to all requests, which includes demand and prefetch packets. In addition, our proposed congestion detection mechanism will be complementray to congestion detection mechanism proposed by these previous works. The metric used to throttle or obtain congestion information is also different from our work as we obtain congestion information for each source-destination pair. However, our network-aware prefetcher can be modifed to use these metrics–which can result in different cost-performance trade-off.

Design of some on-chip networks have included separate networks (or partitioned network). Some of the on-chip network designs have used separate or partitioned networks, similar to what we evaluated in Section 2.4. For example, Tilera's Tile64 [37] have five separate networks for different traffic classes. Balfour and Dally [7] explored the benefit of network duplication. However, our results show that partitioned network resources does not provide any performance benefit when we consider the impact of prefetch traffic.

7 Conclusion

In this paper, we investigated the mutual effects of hardware prefetchers and on-chip networks, and proposed the improvements for prefetcher and on-chip network designs. Just partitioning network resources for two types of traffic to isolate them from each other lowered the overall performance. Instead, using unified networks, but prioritizing demand packets over prefetch packets at the source node or routers resulted in significant performance improvements. This paper also showed that for prefetcher designs, it is critical to observe congestion on networks, and adjust the aggressiveness of prefetchers.

Our experimental results shows that the combined benefits of prefetch-aware networks and network-aware prefetchers is about 11–12% performance improvements for Out-of-order cores, and 21–22% for SMT cores on average, up to 37% improvement for some of the workloads and prefetchers.

We believe the potential for this mutual awareness will grow, as more aggressive prefetchers are used and memory and on-chip cache latencies increase and the number of nodes and thus the number of hops in networks increase.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No.2012R1A1A1014586), and the SW Computing R&D Program of KEIT(2011-10041313, UX-oriented Mobile SW Platform) funded by the Ministry of Knowledge Economy. John Kim was supported by WCU (World Class University) program under the National Research Foundation of Korea and funded by the Ministry of Eduation, Science and Technology of Korea (Project No: R31-30007)

References

[1] 240pin Unbuffered DIMM based on 4Gb B-die 8GB(1Gx64) Module M378B1G73BH0, *Samsung Electronics Data Sheet* [online]. Available: http://www.samsung.com/global/business/semiconductor/file/product/ds_ddr3_4gb_b-die_based_udimm_rev13-0.pdf

[2] *The International Technology Roadmap for Semiconductors (ITRS), System Drivers*, 2009 [online]. Available: http://www.itrs.net/

[3] D. Abts and D. Weisser, "Age-based packet arbitration in large-radix k-ary n-cubes," in *Proc. ACM/IEEE Conf. Supercomput.*, Reno, NV, 2007, pp. 1–11.

[4] N. Agarwal, et al., "*Garnet: A detailed interconnection network model inside a full-system simulation framework*," Dept. Elect. Eng., Princeton Univ., Technical Report CE-P08-001, 2008.

[5] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. ACM/IEEE Conf. Supercomput.*, New York, NY, USA, 1991, pp. 176–186.

[6] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, May 1995.

[7] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *Proc. 20th Annu. Int. Conf. Supercomput.*, New York, NY, USA, 2006, pp. 187–198

[8] E. Baydal, P. Lopez, and J. Duato, "A family of mechanisms for congestion control in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 9, pp. 772–784, Sep. 2005.

[9] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny, "The power of priority: NoC based distributed cache coherency," in *Proc. 1st Int. Symp. NoC*, Washington, DC, USA, 2007, pp. 117–126.

[10] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter, "Interconnect-aware coherence protocols for chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 34, pp. 339–351, May 2006.

[11] W. Dally, and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[12] W. J. Dally, "Virtual-channel Flow Control," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 194–205, 1992.

[13] W. J. Dally, and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. Design Autom. Conf.*, Las Vegas, NV, June 2001, pp. 684–689.

[14] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, New York, NY, USA, 2009, pp. 280–291.

[15] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Aérgia: Exploiting packet latency slack in on-chip networks," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 106–116.

[16] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 141–152.

[17] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitect.*, New York, NY, USA, 2009, pp. 316–326

[18] P. Gratz, B. Grot, and S. W. Keckler, "Regional congestion awareness for load balance in networks-on-chip," in *Proc. 14th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2008, pp. 203–215.

[19] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger, "Implementation and evaluation of on-chip network architectures," in *Proc. IEEE Int. Conf. Comput. Des. (ICCD)*, 2006, pp. 477–484.

[20] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Symp. Comput. Archit.*, 1990, pp. 364–373.

[21] C. J. Lee, O. Mutlu, V. Narasiman, and Y. Patt, "Prefetch-aware DRAM controllers," in *Proc. 41st IEEE/ACM Int. Symp. Microarchitect.*, Nov. 2008, pp. 200–209.

[22] Z. Li, J. Wu, L. Shang, R. Dick, and Y. Sun, "Latency criticality aware on-chip communication," in *Proc. Des. Autom. Test Eur. Conf. Exhib. (DATE)*, 2009, pp. 1052–1057.

[23] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[24] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[25] N. McKeown, "The islip scheduling algorithm for input-queued switches," *IEEE/ACM Trans. Netw.*, vol. 7, pp. 188–201, Apr. 1999.

[26] U. Y. Ogras and R. Marculescu, "Prediction-based flow control for network-on-chip traffic," in *Proc. 43rd ACM/IEEE Des. Autom. Conf. (DAC)*, 2006, pp. 839–844.

[27] S. Palacharla, and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proc. 21st Annu. Int. Symp. Comput. Archit. (ISCA)*, Los Alamitos, CA, USA, 1994, pp. 24–33.

[28] L.-S. Peh, and W. Dally, "A delay model and speculative architecture for pipelined routers," in *Proc. 7th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2001, pp. 255–266.

[29] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.

[30] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *Proc. 11th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Washington, DC, USA, 2005, pp. 225–236

[31] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. 13th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Washington, DC, USA, 2007, pp. 63–74.

[32] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar./Apr. 2002.

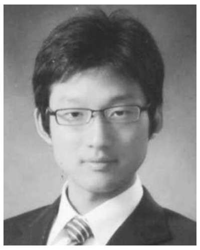
[33] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM J. Res. Dev.*, vol. 46, no. 1, pp. 5–25, 2002.

[34] M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee, "Self-tuned congestion control for multiprocessor networks," *Proc. 7th Int. Symp. High-Perform. Comput. Archit. (HPCA)*,2001, pp. 107–118.

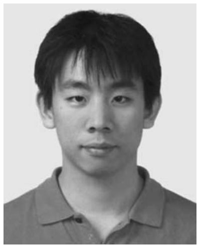
[35] H. Wang, L.-S. Peh, and S. Malik, "Power-driven design of router microarchitectures in on-chip networks," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchitect*, Washington, DC, USA, 2003, p. 105.

[36] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: A power-performance simulator for interconnection networks," *Proc. 35th Annu. IEEE/ACM Int. Symp. Microarchitect*, 2002, pp. 294–305.

[37] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.



Junghoon Lee received the BS degree in computer science from Sungkyunkwan University, Seoul, South Korea, and the MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. He is a PhD candidate in computer science at Korea Advanced Institute of Science and Technology (KAIST). His research interests include computer architecture, on-chip interconnection network, and system security.



Hanjoon Kim received the BS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. He is a Ph.D. candidate in computer science at Korea Advanced Institute of Science and Technology (KAIST). His research interests include computer architecture, parallel computing, and on-chip interconnection network.

Minjeong Shin received the BS and MS degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. She is an associate researcher in the Creative Innovation Center at LG Electronics. Her research interests include OS kernel architecture and low-level hardware/software interface design.

John Kim received the BS and MEng degrees in electrical engineering from Cornell University, New York, in 1997 and 1998, respectively and the PhD degree in electrical engineering from Stanford University, in 2008. He is currently an Assistant Professor in the Department of Computer Science at KAIST with joint appointment in the Web Science & Technology Division at KAIST. He spent several years working on the design of different microprocessors at Motorola and Intel. His research interests include multicore architecture, interconnection networks, and datacenter architecture. He is a member of IEEE and ACM.

Jaehyuk Huh received the BS degree in computer science from Seoul National University, South Korea, and the MS and PhD degrees in computer science from the University of Texas at Austin. He is an Associate Professor of Computer Science at KAIST. His research interests include in computer architecture, parallel computing, virtualization and system security.