FP-NUCA: A Fast NOC Layer for Implementing Large NUCA Caches

Anuj Arora, Mayur Harne, Hameedah Sultan, Akriti Bagaria and Smruti R. Sarangi

Abstract—NUCA caches have traditionally been proposed as a solution for mitigating wire delays, and delays introduced due to complex networks on chip. Traditional approaches have reported significant performance gains with intelligent block placement, location, replication, and migration schemes. In this paper, we propose a novel approach in this space, called FP-NUCA. It differs from conventional approaches, and relies on a novel method of co-designing the last level cache and the network on chip. We artificially constrain the communication pattern in the NUCA cache such that all the messages travel along a few predefined paths (*fast paths*) for each set of banks. We leverage this communication pattern by designing a new type of NOC router called the *Freeze* router, which augments a regular router by adding a layer of circuitry that gates the clock of the regular router when there is a *fast path* message waiting to be transmitted. Messages along the *fast path* do not require buffering, switching, or routing. We incorporate a bank predictor with our novel NOC for reducing the number of messages, and resultant energy consumption. We compare our performance with state of the art protocols, and report speedups of up to 31% (mean: 6.3%), and ED^2 reduction up to 46% (mean: 10.4%) for a suite of Splash and Parsec benchmarks. We implement the *Freeze* router in VHDL and show that the additional fast path logic has minimal area and timing overheads.

Index Terms—NUCA caches, Freeze router, Bank prediction

1 Introduction

Due to continued Moore's law based scaling, the number of cores per chip, and the size of the last level cache (LLC) are doubling roughly every two years. As a direct consequence of this trend, it is becoming increasingly necessary to connect the cores and cache banks with a complex NOC. Due to the increased number of links, buffers, and routers on the NOC, the average time to reach distant cache banks is showing an increasing trend [5]. Along with increasing access latency, the NOC is also becoming a major source of power consumption in modern processors. Consequently, it is necessary to design effective access protocols for the LLC such that we can minimize access latency, and power consumption for manycore processors of the future. In this paper, we focus our attention on effectively co-designing the shared last level L2 cache and the NOC.

Let us start with a historical perspective. Since the beginning of this century, researchers realized that interconnect delay is not scaling with processor speed. Hence, researchers proposed non-uniform cache architectures (NUCA) architectures [6], [29] that essentially prioritize nearby cache banks, and try to move data "closer" to the cores. Some of the early work in this area [6], [10] investigated different methods of data placement, location, and ultimate migration of data towards cores that tend to access them frequently. At that time, this problem was very relevant because processor speed was rapidly increasing [6], and interconnect delay was not able to keep up with it. However, for the last 10 years, processor speed has remained stagnant, and interconnects are gradually getting faster, albeit at a very slow pace. Nevertheless, the problem of designing large multi-banked caches that are conscious of a wide variance of access times depending on the placement of data, has not vanished. It has simply reinvented itself with a different flavor.

Today, large LLCs need to be NOC-aware. Data has to be placed closer to the core such that we can avoid costly traversals on the NOC. In this context, the problem of creating NUCA caches can be divided into three parts: (1) data placement, (2) search, (3) and migration. For placing data, researchers have primarily looked at two sets of approaches. The first approach proposes to partition data statically based on their access patterns. Data can be private to one core, or can be shared. We can either annotate data at the page level [12] or use compiler support. The idea here is to place private data close to the requesting core, and shared data in far away banks. The second approach is to achieve this split dynamically [10], [13], [23]. Based on the access patterns of data, heuristics [12], [13] have been proposed in prior work to either keep them close to cores, or store them in remote cache banks.

For the problem of searching for data, most NUCA proposals associate a set of banks with each block. We

- *Anuj Arora, Akriti Bagaria and Smruti R. Sarangi are with the Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India – 110016. E-mail: {mcs122812,mcs132541, srsarangi}@cse.iitd.ac.in*
- *Hameedah Sultan is a student in the Electrical Engineering Department at the Indian Institute of Technology Delhi, New Delhi, India – 110016. E-mail: jvl132811@ee.iitd.ac.in*
- *Mayur Harne is with NVIDIA Inc., Panchshil Tech. Park, Shivaji Nagar, Pune, India – 411005. E-mail: mharne@nvidia.com*

can start searching from any bank, and then iterate through the entire set. The search policy has an impact on performance. For the problem of migration, most proposals have looked at migrating data towards the core that has requested it. The proposed heuristics typically incorporate a variety of threshold values for making decisions on when to migrate and block, and its target location within the bank set. Another interesting design point in NUCA caches is the selective use of block replication [9], [34]. In this paper, we shall not consider replication of non-read-only blocks because, we need to implement cache coherence for the LLC, which has been deemed to be very expensive in terms of area, power, and complexity in prior work [12], [29].

In this paper, we propose a new set of schemes collectively called FP-NUCA (Freeze-Predict NUCA) for a non uniform cache, which are very different from prior work. The basic aim is to avoid complicating the cache banks, and avoid adding additional software layers to mark pages as shared or private. Our alternative strategy is to restrict the communication pattern in the NUCA cache, and subsequently leverage this communication pattern to design a fast and power efficient NOC. We divide our set of cache banks into bank sets. Each requesting core (or L1 cache) has a designated *home bank* in every bank set. All the requests from the requesting core go to the home bank, and then the home bank searches for the block in the bank set according a pre-defined search policy. We restrict the pattern of messages such that they can only travel on a set of pre-defined paths. We propose the design of the *Freeze* router that augments a traditional 3-5 stage NOC router with additional circuitry. The additional circuitry gates the clock of (freezes) the router, whenever an FP-NUCA message needs to be transmitted. The router transmits the FP-NUCA message, and then resumes processing normal messages. This strategy ensures that FP-NUCA messages reach their destination without any stalls and buffering overheads. Furthermore, we use the *Freeze* routers to create a high speed path to a bank predictor. We demonstrate roughly 20% energy reduction in the NOC and cache subsystems by augmenting our scheme with predictors.

We compared our results with two of the best performing NUCA schemes proposed in the last five years namely R-NUCA [12], and SP-NUCA [29]. The best scheme in FP-NUCA outperforms R-NUCA by 6.3% and SP-NUCA by 5.7%. The maximum speedups can go up to 31%. Furthermore, we obtain on an average 10.4% reduction (maximum: 46%) in ED^2 by employing the best FP-NUCA scheme.

The organization of this paper is as follows. We discuss related work in Section 2. Subsequently, we characterize the behavior of applications in Section 3, and try to find the characteristics of applications that might potentially benefit by using a FP-NUCA cache. Then, we discuss the hardware implementation in Section 4, present the results in Section 5, and finally conclude in Section 6.

2 Background and Related Work

Figure 1 shows a typical tiled architecture of a manycore CMP. In such CMPs, a set of cores and LLC (last level cache) cache banks are logically grouped into a set of *tiles*. The set of LLC cache banks in a tile are known as a *slice*. Additionally, each tile contains the L1 caches, a set of routers for communication on the NOC, and possibly a memory controller. In this paper, we consider the LLC to be the L2 cache that is split into multiple cache banks.

2.1 Bank Based Schemes

One of the earliest proposals by Kim et al. [6] proposed a method to divide a large LLC into bank-sets with variable access latencies. In this scheme, blocks do not migrate between cache banks, and thus it is also known as a static NUCA (S-NUCA) scheme. Here, cache lines are statically mapped to cache banks. The D-NUCA (dynamic NUCA) [10] scheme extends the basic S-NUCA scheme. It associates a set of cache banks (bank-set) with a cache line. While locating a block, it is necessary to search all the banks in the bank-set. Secondly, frequently accessed blocks can dynamically migrate towards the requesting cores to minimize wire delay. Let us now evaluate some major proposals in terms of block placement, location, migration, and replication.

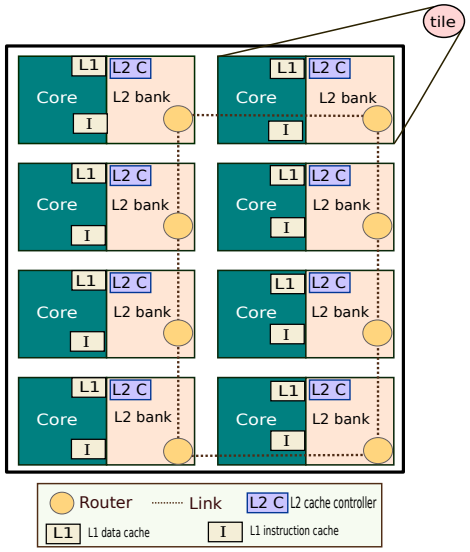

Fig. 1: Layout of a NUCA cache

R-NUCA [12] classifies memory accesses of workloads into three distinct categories – *private*, *read-only*, and *shared*. The classification is done at a page level and requires operating system support. The authors propose to keep the private data in a nearby bank, keep the shared data in a unique remote bank, and distributed the read-only data in overlapping clusters. The scheme prioritizes private data over shared data and makes its access faster. This strategy is useful under two circumstances. The first is when most of the data accessed is private and it fits within one bank, or the data is very heavily shared. For

read-only data such as instructions, this scheme reduces conflict and capacity misses by distributing the data across a set of cache banks.

Huh et al. [16] propose a NUCA based organization that uses a similar classification of accesses and uses a spectrum of cache bank mapping policies with different *sharing degrees*. A private block has a sharing degree of 1, whereas a block accessed by N threads has a sharing degree of N. Their dynamic bank-mapping scheme takes the degree of sharing into account, and tries to minimize hit delay.

2.2 Bank and Set Based Schemes

Merino et al. [29] proposed the SP-NUCA scheme that dynamically partitions each set into private and shared ways. A core first accesses its nearest bank, and then a designated remote shared bank. If a block is not found in both the banks, then it checks the private banks of the rest of the cores, and performs a block migration if necessary. The SP-NUCA proposal mainly focuses on replacement algorithms to reduce on-chip access latency without compromising on the hit-rate. It dynamically changes the ways of each set from private to shared and vice-versa depending upon the memory access pattern of the workload. Merino et al. subsequently extended this idea and proposed the ESP-NUCA [28] scheme that incorporates selective replication, and management of victim blocks. The main drawback of this scheme is the hardware overhead for maintaining coherence among replicas.

PSA-NUCA [14] is a similar proposal that takes into account the asymmetric distribution of memory accesses in workloads. It proposes novel replacement, mapping and replication policies by using pressure information, which is defined as the number of accesses for each cache set. Two recent proposals namely Elastic Cache [13] and Cloud Cache [23] propose methods to partition the address space into multiple sub-address spaces. Each address space is tailored to a specific type of data, and its directories and coherence engines are managed separately. These schemes however were not experimentally found to be conclusively better than R-NUCA. Hence, we choose R-NUCA as a point of comparison.

2.3 Smart Search and Bank Prediction

Kim et al. [19] proposed smart search techniques using partial-tag comparison in order to reduce the hit latency and miss resolution time in a D-NUCA cache. The partial-tag bits are stored in a smart search array located in the cache controller. Bischewski et al. [4] studied the effect of bank-predictors for distributed L1 caches.The study concludes that accurate bank-prediction can significantly reduce the perceived cache access latency. Ricci et al. [31] propose to use a Bloom filter for each set of cache banks. To check if a block exists in a set of cache banks, we simply need to check its associated Bloom filter. A Bloom filter can give a false-positive result, but never a false-negative result. If the output of the Bloom filter is negative, then we need not search the bank cluster; we can move to another bank cluster. We design our bank predictor on the lines of the designs proposed in [4], and integrate it into our design. We were not able to locate prior work that incorporates and evaluates bank predictors at the L2 level for shared caches.

2.4 Specialized NOCs

There is a plethora of prior work on NOCs. In this paper, we focus on ideas that propose a quick(fast) path between sets of nodes. The earliest solutions in this area consisted of circuit switched networks [11] that can be used to setup dedicated paths between pairs of nodes. The transmission time between the nodes was low because there was no additional routing delay. However, the flexibility of this approach is limited, and the time for setting up a path is prohibitive. Hence, the focus has moved to packet switched networks with fast paths between pairs of nodes.

A seminal paper in this area is the work on express virtual channels by Kumar et al. [21]. They propose to partition the set of virtual channels (VCs) into two groups. The first group consists of normal VCs, and the second are express virtual channels (EVCs). We have two kinds of nodes: source/sink nodes and bypass nodes. Packets enter a dedicated EVC in a source node and pass along a pre-specified route to the sink node. There is no additional delay associated in the bypass nodes because the EVCs for such traffic are always preallocated and it is not necessary for packets to use the rest of the router pipeline (VC allocation, route computation, and switch traversal). The authors use different pre-specified EVCs having different lengths. As compared to this, our scheme is more generic; it does not limit the size of the fast path. Moreover, our scheme is a wrapper on an existing router; it does not propose modifications to the core functionality. Krishna et al. [20] extend the work on EVCs by introducing different types of interconnects for different types of traffic: normal or express. They observe that a major limitation of EVCs is the time it takes to reserve buffers along the express path. Hence, they propose to use a fast control plane for setting up this path. In comparison, our scheme does not reserve buffers along the path. The VIP [30] router is designed on similar lines. Instead of having a dedicated express virtual channel, it uses a 1-flit buffer at both the input and the output to store a flit. A VIP path needs to be configured in advance.

Let us now look at a set of schemes that do not rely on prioritized virtual channels. One of the seminal works in this field is ReNoc [32] that proposes to use multiplexers to bypass the router pipeline completely. The idea is to use a multiplexer between the output of a router, and the incoming link. However, these multiplexers are statically configured, and as mentioned in the paper they are meant to be configured very infrequently. Instead

of directly bypassing the router, the SMART scheme (Chen et al. [7]) proposes a bypass path over the input buffers. A crossbar input port can either get its data from the input buffers or from the bypass path. In this case also, the multiplexers need to be configured in advance, and it is possible for two requests to contend for the same output port. This is solved by buffering. LOCO [22] uses the SMART router to send fast unicast and broadcast messages. It is the closest to our scheme. It creates a hierarchical network using SMART links. Cores are grouped into clusters. Each core is connected with its home node (specific to each cluster) with a fast SMART link. It can search for data in its home node. It can also search for data in the home nodes of other clusters by quickly broadcasting data over the virtual network. Moreover, it supports data block migration across clusters, and can adapt the cluster size based on application characteristics.

FP-NUCA is different from prior work in the following ways: it does not require any preconfiguration of paths; it is oblivious to the design of the routers, and it does not modify them; it relies on clock-gating the router, and uses multiplexers at the output ports (other approaches use multiplexers at the input ports, or within the router).

3 Characterization of Applications

In this section, we characterize a suite of applications and kernels from the Parsec [3] and Splash-2 [33] benchmark suites (see Table 1). It is necessary to understand the behavior of these applications from the point of view of the LLC such that we can design effective NUCA protocols. The procedure for collecting the memory accesses of the LLC is as follows. We run the benchmarks till all the threads are spawned. Then we warm up the caches by running 100 million instructions. Subsequently, we run on an average 200 million instructions per thread on an architectural simulator and analyze their behavior. Our simulated system is a 32 core machine, with a private L1 cache (32 KB/core, cache coherent), and shared last level L2 cache with 32 banks (256 KB/bank) (simulation parameters shown in Table 2).

3.1 Block Access Frequency

Figure 2 plots the frequencies of accesses of blocks for each benchmark. For example, in *Blackscholes*, only 2% of the blocks are accessed only once, and 31% of blocks are accessed 2-5 times in our simulation interval. For 7 benchmarks, 70% of the blocks are accessed only 2-5 times. For *Fluidanimate*, 80% of the blocks are accessed just once. For *Lu*, 74% of the blocks have 6-10 accesses during our simulation interval. If we consider a threshold of 100 accesses, then only three benchmarks have a sizeable (>10%) number of blocks in this category. They are *Swaptions*(89%), *Water-sp* (42%) and *Barnes*(56%). We can conclude from this experiment that for a majority of the benchmarks, most of the blocks are accessed less than 10 times. **Hence, we need to have schemes that quickly migrate data to closer nodes.** Let us now study the nature of memory accesses between two consecutive accesses to the same block.

Application	Input size
PARSEC (simlarge)	
Blackscholes	64KB options
Bodytrack	4 cameras, 4 frames, 4,000 particles, 5 annealing layers
Canneal	15,000 swaps per temperature step,2,000 start temperature, 400,000 net list elements
Fluidanimate	300,000 particles, 5 frames
Streamcluster	16KB input points, 16KB points, 128 point dimensions
Swaptions	64 swaptions, 20,000 simulations
Splash-2	
Barnes	16KB particles
Fmm	8KB particles
Lu	512 x 512 matrix (lu contiguous)
Radiosity	batch, largeroom
Water-nsq	512 molecules (water nsquared)
Water-sp	512 molecules (water spatial)

TABLE 1: Configuration of benchmarks

3.2 Stack Distance

Let us characterize accesses to the L2 cache on the basis of *stack distance* [2], which is a standard metric for evaluating temporal locality. We define the stack distance as follows. Let us assume that we maintain a stack of block addresses. Whenever a given block is addressed, we search for its address in the stack. If the block is found at a depth of k, then we record the stack distance of the block to be k. Here, the *depth* is defined as the distance from the top of the stack (which has a depth of 0). Note that the first time a block is added to the stack, we do not record the stack distance. Benchmarks with low stack distances exhibit high temporal locality, because the same blocks tend to get reused quickly. Defining the stack distance for multithreaded benchmarks is slightly tricky because we need to order concurrent requests. This ordering is important when the stack distance is of the same order of magnitude as the number of threads. However, for large values (as we observe in our experiments), the ordering is insignificant.

Figure 3 shows the CDF (cumulative distribution function of the stack distance) for all our simulated benchmarks (across all the blocks accessed). We observe that most of the values are between 100 and 10,000. *Blackscholes* has the smallest average values of stack distance (≈ 250), whereas *Lu* has the largest stack distance value ($\approx 60,000$). The stack distance observed at the L2 cache is a function of the algorithm implemented by the workload, and the operation of the L1 caches. Since the L1 cache contains 256 cache blocks, stack distance values less than 256 are definitely indicative of conflict misses. 6 out of 12 benchmarks have stack distance values less than 256 for a third of their accesses. For 9 out of 12 benchmarks, 90% of the values are less than 25,000. We also observe that there are two points of inflexion, which are in the vicinity of 300 and 1000, for all the

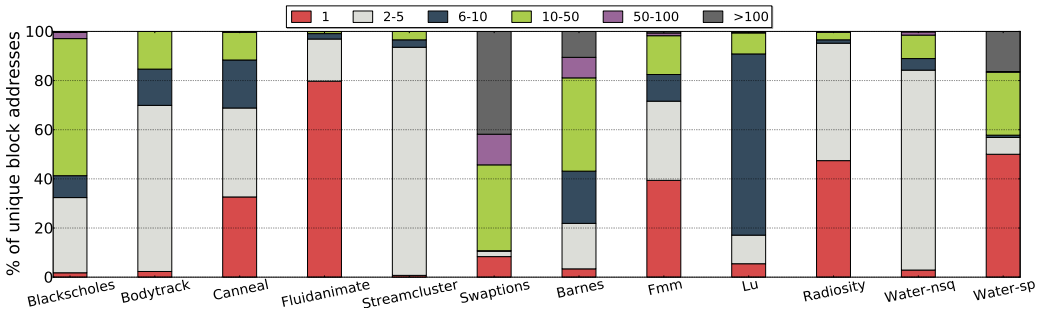

Fig. 2: Block access frequency

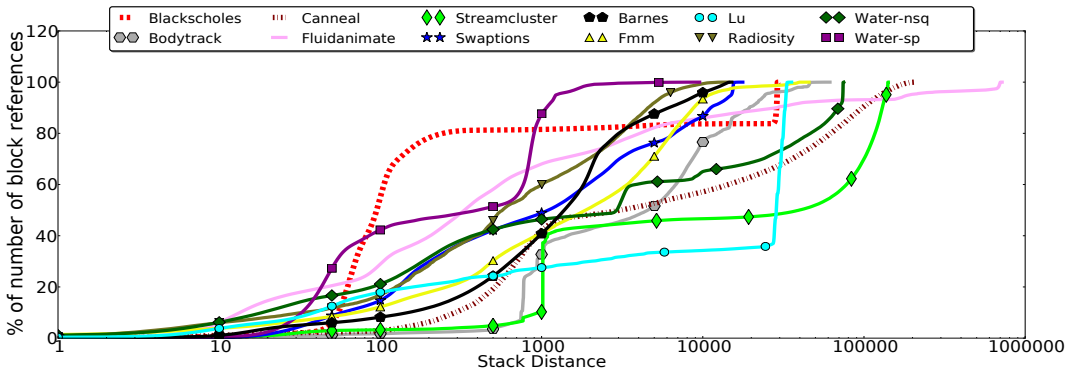

Fig. 3: CDF of the stack distance

benchmarks other than *Water-sp* and *Blackscholes*. The insight that we obtain from this experiment is that stack distance values are typically high (> 500), and even if we assume the worst case where the requests to L2 are equally distributed across bank sets, there are at least 60+ unique requests between two consecutive requests to each bank set (assuming 8 bank sets). **In practice, we will have hundreds of cycles to migrate the block between different banks of each bank set. This is more than sufficient.**

3.3 Shared/Private data

Almost all the prior works on NUCA caches have exploited the sharing patterns in data for designing optimal NUCA protocols [25]. This helps us in designing methods to optimally allocate data between local and remote cache banks, and also manage data migration and replication. Let us thus study the nature of block accesses. Figure 4 shows the distribution of the number of cores that access a given block. We observe that for benchmarks other than *Streamcluster* and *Barnes*, more than 40-90% of the blocks are accessed by a single core. The total number of blocks that are accessed by 2-4 cores is typically between 25-40% for these benchmarks. 90% of the blocks in *Streamcluster* are accessed by 3 cores (because of the nature of the computation). 44% of the blocks in Barnes are accessed by 2-4 cores, and the rest of the blocks are accessed by more than 4 cores. For the rest of the 11 benchmarks, only 4 of them have blocks that are accessed by > 4 cores, 5-20% of the time. **We can thus conclude from this experiment that blocks are accessed rather infrequently, and it would not be wise to bring all of them close to the requesting cores**.

Let us now further look at the nature of memory accesses. We classify data into three categories – instructions, private data, and shared data. For *Blackscholes*, *Bodytrack*, *Fluidanimate*, *Swaptions*, *Water-nsq*, and *Lu*, the percentage of private accesses is more than 60%. For all the benchmarks, the memory footprint of instructions is very small (<5%). *Canneal*, *Streamcluster*, *Water-sp*, and *Barnes* have more than 80% shared accesses. *Fmm* and *Radiosity* have a balanced profile of private and shared accesses. Given the spectrum of behaviors, **we believe that keeping private data in a bank close to the requesting core might not be a feasible solution because all the data might not fit in that bank**. We will need to spill it to other banks, and for block location, we would consequently require sophisticated search protocols. Secondly, it is also **not wise to use a scheme like S-NUCA, which does not differentiate between shared and private data, because for some benchmarks most of the data is private, and it would be beneficial if a thread can access its private data quickly**.

4 Hardware Implementation

4.1 Design Principles and Policies

From our discussion in Section 3, we have made the following conclusions. (1) It is beneficial to save private

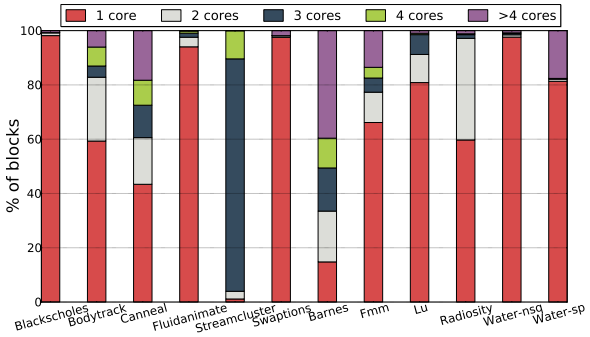

Fig. 4: Degree of sharing across cores

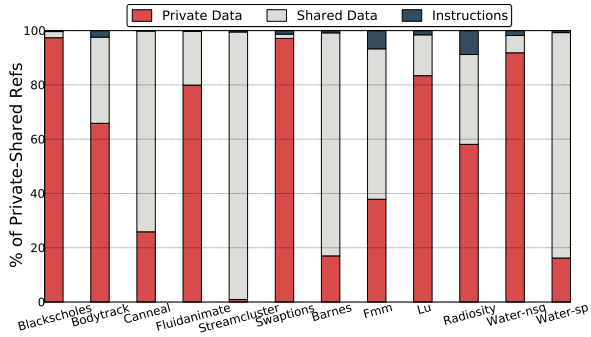

Fig. 5: Percentage of private and shared data references

data closer to the requesting core; (2) however, we cannot save private data in a fixed set of banks, or a fixed number of ways in each set lest there is an overflow (3) It is also necessary to quickly locate and access shared data. (4) For most benchmarks shared blocks have limited sharing, are accessed infrequently, and have a stack distance between 300 and 10,000. Several proposals [12], [29], [28] dedicate nearby banks and some ways in each set for private data. This strategy is not expected to yield benefits because for some benchmarks their entire working set consists of private data. Another set of proposals [13], [23], [6] distribute shared data across a complex network of banks, and have elaborate strategies for location and migration. Since each block is accessed less than or equal to 5 times with a probability of 51%, such complex schemes might not be justified.

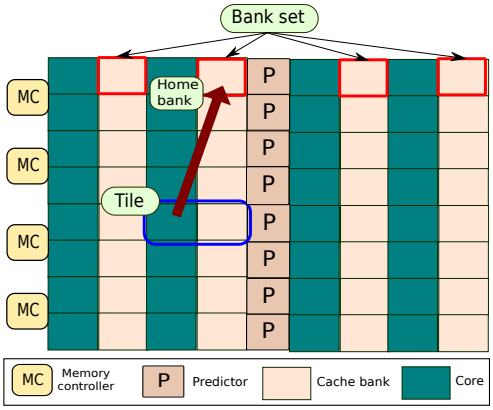

Fig. 6: Layout of cores and cache banks

Hence, in this paper, we design a new scheme that is in accordance with our observations, is simple, and is fairly different from prior work. Figure 6 shows the layout of cores and cache banks for a 32 core machine. Similar to traditional D-NUCA schemes, we divide the banks into *bank sets*. We can define bank sets row wise or column wise. For the purpose of exposition, we assume that we allocate bank sets row wise as shown in Figure 6. Note that a bank might physically consist of many smaller SRAM arrays (traditionally known as a bank). We conducted experiments with different kinds of layouts (1x1 chess board, 2x2 chess board, stripes) and different tile sizes (1-4 banks per tile). However, we did not observe any significant differences(<1-3%) in terms of performance, maximum temperature, and power.

Let us start by defining the concept of a *home bank* (in the LLC), with respect to a given block and core. We first divide the physical address into the block, index, and tag. We use the lower bits of the tag to compute the bank set. Note that an alternative scheme for addressing bank sets in [10] has lower performance. Now, the home bank is the bank in the bank set that is closest to the requesting core/tile (based on Euclidean distance). A controller in the home bank known as the home bank controller (HBC) co-ordinates with HBCs of other banks in the bank set to search for the block according to a given *search policy*. We have four kinds of search policies – *Seq* (sequentially search through all the banks in the bank set), *Two-way* (search in both the directions, left and right, from the home bank simultaneously), *BCast* (search in all the bank sets simultaneously), and *PredBCast* (broadcast with bank prediction).

By default, all newly loaded blocks are placed in the home bank. If it is necessary to evict a block from the home bank, then we move it at random to any of the blocks to the right or left. If we need to evict a block from any other bank, then we move it one block away from its home bank. For block migration, we follow a reverse policy. Whenever, there is a hit in a bank that is not the home bank of the requesting core, we move it to the immediately next bank in the direction of the home bank.

We incorporate a bank predictor in our design. The main insights behind the design of our bank predictor is as follows. The stack distance for 32 cores is limited to 10,000. If we assume 8 bank sets, then the expected stack distance (correlated with measurements) is 10,000/8 = 1250. Hence, if we have one predictor for each bank set that contains 1024 entries, and a LRU replacement scheme, we expect to keep track of most of the frequently used blocks in the bank sets. The protocol for accessing the predictor is as follows. If a block is not found in the home bank, then the home bank forwards the request to the predictor for the respective bank set. The predictor predicts a bank, and subsequently transfers the request

to that bank. If the bank does not contain a copy of the block, then it forwards the requests to the rest of the banks as per the *search policy*. If any bank has the block, then it forwards it to the home bank.

The key observation that needs to be made here is that we have artificially restricted the communication pattern of the FP-NUCA messages. A request is first sent to the home bank, and the home bank then sends messages to the rest of the banks in the bank set (if required). All the responses come back to the home bank, and then the home bank forwards the block to the requesting core.

We design a novel NOC router (called the *Freeze* router) that clock gates the routers of the NOC and forwards the FP-NUCA messages to their destination without using buffering, and by bypassing the pipeline of the router. The *Freeze* router can thus route messages with almost no overhead, which, along with more effective placement and searching, explains our relatively superior performance over competing designs.

4.2 Home Bank Controller (HBC)

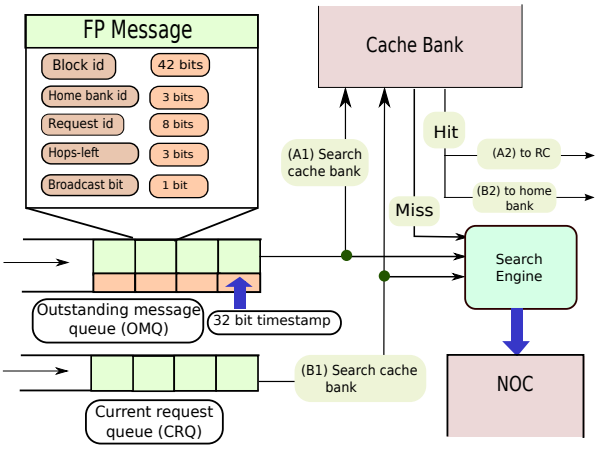

Fig. 7: Structure of the HBC

The structure of the HBC is shown in Figure 7. At the outset, the HBC of the home bank accepts a request from the requesting core (RC), and adds it to the outstanding messages queue (OMQ) with a 32 bit time-stamp (incremented every cycle), and a unique request id (home bank id + 8 bit sequence number). Subsequently, it initiates a search in the local bank, and if there is a hit, then it returns the block to the RC. However, upon a miss, it is necessary to initiate the search algorithm. Implementing the sequential search algorithm is fairly trivial. The HBC sends a request to the HBC of its neighboring bank, which forwards the request to the next bank and so on till there is a hit. If the request reaches the last bank, then the direction of search is reversed (starting from the HBC). The HBC concludes a LLC miss, if the request returns from the last bank in both the directions.

Let us discuss the implementation of the two-way algorithm, which is more involved and requires greater support at the level of the HBC. We begin by sending a request in both the directions (left and right). Each bank adds the request to the CRQ when it is being processed. Once the request has been processed (hit/miss decision made), it is removed from the CRQ. Now, if there is a hit in a bank, then its HBC sends a copy of the block to the HBC of the home bank. The HBC of the home bank then has to cancel the messages sent in the other direction in the interest of reducing power and contention. It computes the ID of the target bank based on an estimate of the time it takes to access each cache bank, and the number of cycles elapsed, and then sends it a message. If there is no entry in the target CRQ (delay due to contention) then the HBC forwards the message in the direction of the home bank. Once, an HBC finds the request in the CRQ, it deletes the request. In our experiments, we assume a 2-port cache bank, which gives equal priority to the requests in the OMQ and CRQ.

Similarly, to implement the broadcast based search algorithm, the HBC sends a request to all the banks simultaneously. Each HBC initiates a search in its local cache bank. If there is a hit, then a message is sent to the HBC of the home bank. Let us now consider the case when there is a miss in all the banks. The HBC of the home bank needs to be made aware of this fact such that it can send a message to the memory controller. Instead of sending a message on every miss, we propose a different solution. Each HBC has a notion of a timeout (B_T). If the local bank is taking more than B_T cycles to search for the block due to contention, then it sends a $BUSY$ packet to the HBC of the home bank. Subsequently, it (and all the banks accessed next) need to inform the home bank regarding the status of the request. The home bank's HBC waits for $B_T \times N$ cycles (N is the number of banks on the critical path) if it hasn't received any $BUSY$ packets. After that it informs the memory controller to fetch the block of memory. Otherwise, it waits till it has gotten hit/miss messages from all the banks that had sent $BUSY$ packets.

4.3 Prediction and Migration

We implement the bank predictor as a 4-way 1024 entry cache (size chosen on the basis of the stack distance profile for bank sets of size 2-8 MB). Each entry contains the ID of the bank that most likely contains the block. If the HBC of the predicted bank realizes that the bank does not have the block, then it initiates the process of search, and all the banks are instructed to send the block back to the home bank. The HBC of the bank that has a hit initiates a process of migration to the neighboring bank (in the direction of the home bank) after the request is serviced. It is not necessary to send another message to the predictor to update itself. Whenever there is a hit in the predictor, the predictor automatically assumes that the block will migrate one step towards the home bank at a later point of time. It thus dynamically updates the ID of the entry.

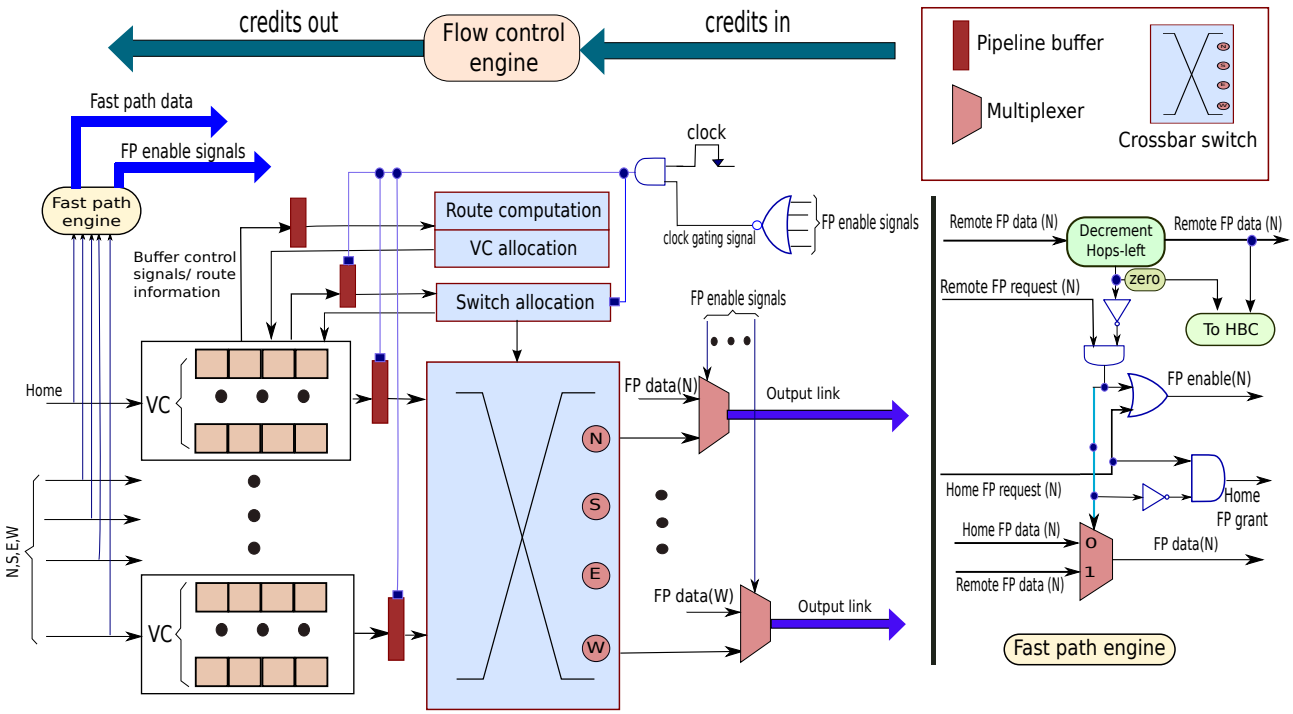

Fig. 8: The design of the *Freeze* router

4.4 Freeze Router

The NOC is used for implementing directory based cache coherence, location/placement/migration of blocks in the LLC (in our case, the L2 cache), for communicating with the memory controller, routing interrupts, and for invalidating blocks that are used by I/O devices (I/O coherence). Let us focus on FP-NUCA messages that are only confined to communication in the LLC. L1 coherence and I/O messages are roughly 3-4 times more than the messages in the L2 layer for a directory hit rate of 50%. Let us now propose to give a higher priority to messages in the LLC layer such that we can take advantage of the message communication pattern in our protocol.

There are two primary kinds of messages in our protocol. Messages and responses travel between the RC to the home bank's HBC along a column. Messages in the bank set travel along a row. We shall prioritize these messages. Note that all the messages from the RC to any HBC have the same priority. They are sent in FIFO order from the RC. All the other messages sent by the RC for coherence and I/O have a lower priority.

Let us now design a router as shown in Figure 8 that works as follows. The router has 5 input ports (1 for each direction, and 1 for the local bank). If the *fast path* bit of a message is set to 1, then it is a *fast path*(FP) message and needs to pass through the router without any buffering. Instead of queuing the message in the virtual channels it is sent to the FP engine.

The FP engine starts by decrementing the *Hops-left* field. If it becomes equal to 0, then it is sent to the HBC of the local bank (the message has reached its destination). Otherwise it is sent in the same direction to the next router. Figure 8 shows a message that is travelling north. Now, it is possible that there is a FP message that needs to be sent, and the local bank is interested in sending a FP message also. In this case, priority is given to the remote request (otherwise, buffering is required). The FP enable signal is asserted if either a local request or a remote request needs to be sent as a FP message. If the *Broadcast bit* is set then the message is delivered to every HBC on the way (not shown in the figure). Note that a FP message never makes a turn; hence, there are no conflicts between remote FP messages.

The rest of the router is organized as a traditional router with virtual channels, lookahead routing, and bypassing (see [17]). Such routers typically have a 3 stage pipeline (buffer write (BW), switch allocation (SA), switch traversal (ST)). However, for the first router in the path of a message, there is an additional pipeline stage (VA) that allocates a virtual channel. In a lookahead router, route computation is done 1 hop in advance. Hence, it is not on the critical path, and can be done in parallel with the BW or SA stages. Let us now focus on the pipeline registers. If a FP message needs to be sent on any link, we propose to temporarily freeze the entire router (all the channels) for 1 cycle. This might sound overkill; however, it leads to a very simple implementation, and the performance penalties are not prohibitive. To freeze the router, we gate the clock of the registers between the BW, SA, and ST stages. This ensures that no message moves in the data path of the

router pipeline and no regular message is sent across any link.

When these stages are clock-gated, we do not want to lose any incoming message or flow control messages. The message buffers can keep on accepting new data. A buffer overflow will not occur since we use credit based flow control, and neighboring routers will not send messages after they run out of credits. We can also perform the route computation and VC allocation during this time. Similar to other state of the art designs we do not have output buffering in our router [17].

For the outgoing link, we need to choose between the high-priority fast path flits, and the low-priority flits that pass through the router's pipeline. To choose between both the types of data, we use a multiplexer at each output port of the router. One of the inputs is connected to the fast path engine, and the other is connected to an egress port of the crossbar. The multiplexer is controlled by the signals generated by the fast path engine. It always prioritizes the fast path. An astute reader might argue that prioritizing some messages might not necessarily be beneficial, because it might delay the other low priority messages (e.g., L1 $\leftrightarrow$ directory messages). Our answer would be that we should only designate those messages as *fast path* messages that are not very frequent (or very infrequent), and often lie in the critical path of the execution. Given our performance results, we would like to believe that FP-NUCA messages in the L2 cache fall in this category. Hence, if we prioritize them, we get performance gains for applications such as Parsec and Splash benchmarks.

4.5 Deadlock/Starvation Freedom

Any NOC protocol is prone to deadlock and starvation unless additional steps are taken. We propose a simple and practical method to avoid starvation. Note that starvation freedom implies deadlock freedom. Additional solutions are proposed in Appendix D. We size our structures (Section 5.1) such that we do not observe any starvation in our simulation runs consisting of billions of instructions. We thus design a starvation recovery solution that is slow, yet practical. It is based on the conjecture that starvation will rarely be seen in practice.

We incorporate two starvation counters with the OMQ and CRQ for each HBC. Each starvation counter keeps a count of the number of cycles the request at the head of the respective queue has not been able to make progress. If the starvation counter reaches a threshold (500 cycles in our design), then we invoke the starvation recovery algorithm. The HBC sends a message to all other HBCs in its bank set to stop accepting any new messages from their RCs, and to clear off their CRQs. Since every message in a CRQ is also maintained in the OMQ of some HBC, no information is lost. Once the network is quiescent (achieved by waiting for 1 more cycle), the HBC that suffered from starvation starts the process of clearing of the OMQs of each node. In this process each request follows the *Seq* protocol and is sent to all the banks. After a hit/miss decision has been made, the next request in its OMQ is made to proceed and so on. In this manner the OMQs of all the nodes in the bank set are cleared. Note that there will be no starvation or deadlock in this process because there is only one request in flight. Once, all the OMQs are cleared, we restart normal execution.

Parameter	Value	Parameter	Value
Cores	32	Technology	22 nm
Frequency	3.6 GHz		
Pipeline			
Retire Width	4	Integer RF (phy)	160
Issue Width	6	Float RF (phy)	160
ROB size	168	Branch Predictor	Tournament
IW size	54		(Pag-Pap)
LSQ size	64	Bmispred penalty	14 cycles
iTLB	128 entry	dTLB	128 entry
Integer ALU	4 units	Int ALU latency	1 cycle
Integer Mul	1 unit	Int Mul latency	2 cycles
Integer Div	1 unit	Int Div latency	4 cycles
Float ALU	2 units	FP ALU latency	2 cycles
Float Mul	1 unit	FP Mul latency	4 cycles
Float Div	1 unit	FP Div latency	8 cycles
L1 i-cache, d-cache			
Write-mode	Write-back	Block size	64 bytes
Associativity	4	Size	32 kB
Latency	2 cycles	MSHRs	32
Directory	fully mapped, MOESI, 4096 entries, 8-way		
Shared L2			
Write-mode	Write-back	Block size	64 bytes
Associativity	8	# banks	32
Latency (per bank)	8 cycles	Bank size	256 KB
Main Memory			
Latency	250 cycles	Mem. controllers	4
NOC			
Topology	2-D Mesh	Routing Alg.	X-Y
Flit size	16 bytes	Hop-latency	1 cycle
Routing delay	2/3 cycles	# Virt. channels	4
(w/wo bypassing)		Buffers/port	8
Freeze Router: (routing + link) delay			1 cycle
Auxiliary structures (size in number of entries)			
OMQ	16	CRQ	10

TABLE 2: Simulation parameters

5 Evaluation

5.1 Experimental Setup

Table 2 shows our simulated processor configuration. The list of benchmarks along with their configurations have already been shown in Table 1. We use the Tejas simulator [27] for architectural simulation. For the NOC, we take our models from the Garnet [1] simulator, and we use Orion 2 [18] to estimate the power and latencies of NOC components. For energy estimation of the core, and dedicated buffers such as the OMQ and CRQ, we use the McPat toolkit [24]. The VHDL design and synthesis report of the *Freeze* router is explained in Appendix C. To size the OMQ and CRQs we measured the maximum size that they ever attain during all our simulation runs (see Appendix A). The maximum OMQ size is 9 entries ($Blackscholes$), and the maximum CRQ size is 8 entries($Bodytrack$). We thus overdesign and size

the OMQ and CRQ to have 16 and 10 entries respectively. We thus never observe a deadlock/starvation situation in our simulations. Lastly, note that whenever we refer to the term "performance" in our evaluation, we refer to a quantity that is inversely proportional to the total simulated execution time. The speedup is always in terms of increase in performance.

5.2 Comparison of FP-NUCA Policies

5.2.1 Benchmark Characterization

Before discussing the performance results, let us characterize the behavior of benchmarks on our simulated system using the *PredBCast* (broadcast with predict) search policy. The hit rates of caches, and the number of requests seen by the L2 cache is more or less (within 1%) independent of the search policy. Now, we are concerned about three vital parameters – number of requests to the L2 cache per instruction, L2 cache hit rate, and L2 cache latency. We are interested in the number of requests to the L2 cache for getting an estimate of the sensitivity of a benchmark to changes in the latency and hit rate of the L2 cache. Given an L2 request, its latency is determined by the number of hops it traverses to reach the home bank, its hit rate at the home bank, and its hit rate in the rest of the banks in the bank set.

Table 3 and Figure 9 show this information. The first column in Table 3 shows the number of instruction cache misses. For all our benchmarks, the code size of the kernel of each benchmark is relatively small, and thus the instruction cache has a very high hit rate ($> 99.8\%$). The L1 data cache hit rates vary from 71% (*Streamcluster*) to 97.6% (*Radiosity*). Since the Splash and Parsec benchmark suites are parallel applications, with a non-trivial amount of data sharing, a sizeable number of L1 cache misses find their data in other L1 caches. We refer to this aspect as *directory hits*. Note that in our directory, if a line has to be evicted in a directory, it is invalidated in all the sharers. The directory hit rate varies from 22.13%(*Fluidanimate*) to 94.44% (*Radiosity*). Typical values range from 40-70%. Readers must note that the L1 cache layer (inclusive of the directory) filters out a significant number of requests. The fifth column lists the number of requests that reach the L2 cache (per 1000 instructions). The numbers vary from 0.22(*Radiosity*) to 31.54(*Canneal*). For 9 out of the 12 benchmarks the number of requests per 1000 instructions that reach the L2 cache are between 1.5 and 11.

The next column shows the hit rate of the L2 cache. For 5 out of the 12 benchmarks the L2 hit rate is between 40-65%, and for the rest of the benchmarks it is between 81.43–98.85%. Thus, there are two classes of benchmarks. The latter class has a relatively higher L2 hit rate than the former. The L2 hit rate is related to the average number of hops that a message needs to traverse (in the case of a L2 hit). For most of the benchmarks with a high L2 hit rate, the hop count is low. This means that the data is mostly available in the home bank. For benchmarks with a low L2 hit rate, we need to traverse more hops to locate the block.

Figure 9 shows this information in some more detail. It classifies the L2 hits into the following categories: home bank hits, predicted bank hits, and broadcast hits. *Blackscholes*, *Swaptions*, *Lu*, and *Water-sp* have more than 80% hits in the home bank. The next category of benchmarks (60-80% hits in the home bank) are *Bodytrack*, *Fluidanimate*, *Barnes*, *Fmm*, and *Water-nsq*. The rest of the benchmarks: *Canneal*, *Streamcluster*, and *Radiosity* have the lowest hit rates in the home bank with *Canneal* being the worst (54%). Predicted bank hits range from 20-40%, and for some benchmarks such as *Radiosity*, the predictor can predict almost all the banks for L2 hits that miss in the home bank.

5.2.2 Results with Normal Routers

Let us start with Figure 10. It shows the performance of different schemes with conventional routers that implement flit bypassing, and lookahead routing. The results are normalized to the configuration with a sequential search policy (*Seq*). We show the results for three other configurations: *Two-way*, *BCast*, and *PredBCast*. We did not evaluate a scheme with bank prediction and the two-way search policy because we expect a large hit rate in the bank predictor, and thus did not want to complicate the design further by introducing a more elaborate search policy. The last entry in the figure shows the geometric mean (G.M) of the speedups for each configuration over the *Seq* configuration, which we assume to be the baseline (since it is very similar to the original D-NUCA scheme [10]). We observe from Figure 10 that the G.M speedup of *Two-way* over *Seq* is 2%, the mean speedup of *BCast* is 3.5% and the mean speedup of *PredBCast* is 4%.

Let us start out by considering benchmarks with a large amount of private data. In this class, we observe the largest speedups in *Fluidanimate*(12%). *Fluidanimate* has a very high percentage (95%) of private data. All of this data does not fit in the home bank; it tends to get distributed to other banks in the bank set. As a result of this pattern, schemes that send more messages in parallel tend to do better than *Seq* . Let us now consider the case of *Blackscholes*, *Lu*, and *Swaptions*. All of them have a high degree of private data; however, they do not show any significant speedups similar to *Fluidanimate*. We observe that for these three benchmarks, the percentage of L2 hits that find their data in the home bank is more than 85%. This means that their working sets are relatively small as compared to *Fluidanimate*, and thus they do not benefit from a more elaborate search policy. *Water-nsq* also has a large amount of private data; however, it shows a speedup in the range of 5-6%. This is because it has a high block access frequency, and it benefits from the fact that blocks evicted from the home bank can still be found in nearby banks in the bank set.

Let us now look at benchmarks that have a large amount of shared data: *Canneal, Barnes, Water-sp, and*

Application	% I-cache hit-rate	% D-cache hit-rate	% Directory hit-rate	L2 Reqs per 1000 instructions	% L2 hit-rate	Average Hop Length
Parsec						
Blackscholes	99.99	96.69	46.89	5.02	94.77	3.79
Bodytrack	99.98	98.34	43.99	2.23	81.43	4.38
Canneal	99.96	77.85	34.54	31.54	54.00	4.09
Fluidanimate	99.98	85.86	22.13	8.14	46.53	4.22
Streamcluster	99.94	71.16	36.17	2.63	48.17	4.52
Swaptions	99.89	82.67	13.85	20.65	86.72	4.24
Splash-2						
Barnes	99.97	92.20	79.13	4.15	97.35	3.27
Fmm	99.93	94.80	56.36	2.38	86.42	3.77
Lu	99.94	94.07	74.76	1.51	88.49	3.78
Radiosity	99.99	97.61	94.44	0.22	64.41	4.36
Water-nsq	99.94	94.24	70.62	2.87	41.18	4.17
Water-sp	99.96	82.51	11.08	10.92	98.85	3.54

TABLE 3: Memory system statistics

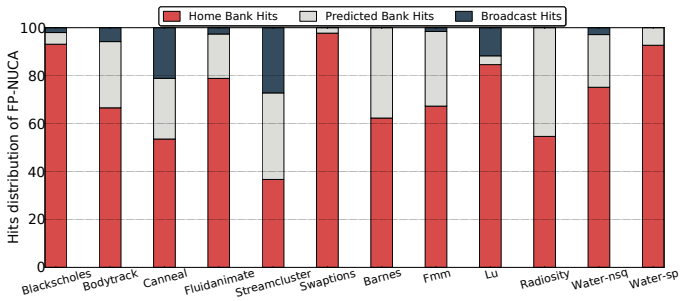

Fig. 9: Distribution of cache hits in FP-NUCA

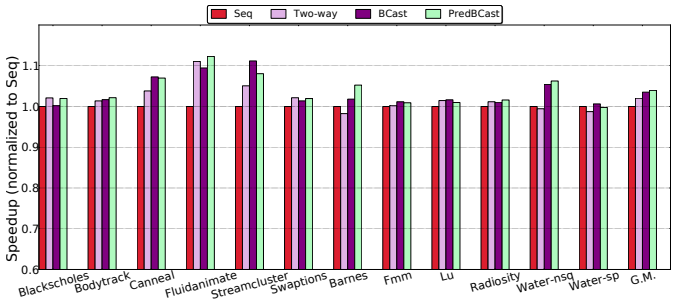

Fig. 10: Performance of different FP-NUCA policies (with normal routers)

Streamcluster. *Streamcluster* has the highest speedup (11%) in this class. This benchmark has a high degree of sharing (more than 85% of blocks are shared by 3 cores), and thus the blocks are uniformly distributed in the bank set (see Figure 9). Hence, it does not benefit by prediction because often messages need to travel to the end of the bank set. In comparison the *BCast* scheme works the best because it is the best scheme for quickly locating uniformly dispersed data. *Canneal*, *Barnes*, and *Water-sp* have much lower levels of sharing, and thus have a higher percentage of home bank hits. Consequently, *Seq* is a relatively better performing scheme.

Benchmarks that fall in the middle of the spectrum are *Bodytrack*, *Fmm* and *Radiosity*. We can discard *Radiosity* from consideration because very few accesses (0.22/1000) reach the L2 cache. For *Bodytrack* and *Fmm*, each block has at the most 2 sharing cores, and thus their blocks tend to bounce between the home banks of different cores. Their speedups are limited to 3% using the *BCast* or *PredBCast* search policies. The reasons for this is that most (roughly 60%) of the L2 hits find their data in the home bank, and the remaining requests are not very widely dispersed in the bank set.

5.2.3 *Results with* Freeze *Routers*

Figure 11 shows the results with *Freeze* routers instead of normal routers. We observe a uniform increase in the speedup by roughly 5%. The only counter-intuitive result is that of the *Seq* configuration for *Swaptions* that shows a slow down with Freeze routers. We believe that this is because more than 97% of the L2 hits find their data in the home bank. The route from the requesting core to the home bank remains busy because of FP-NUCA messages. Since these messages have the highest priority, other messages starve, and this leads to a net slowdown. This is the only example, in which using the Freeze router has led to a more than 1% slowdown.

5.3 Comparison of FP-NUCA with other NUCA Policies

Now, let us compare our schemes with other high performing NUCA policies such as R-NUCA, SP-NUCA, and the baseline S-NUCA scheme. For implementing the R-NUCA scheme we assume the best case scenario, where we are aware of the private and shared pages by performing a post facto analysis of the sharing patterns of the blocks. We use this information to dynamically tag blocks as shared, read-only, or private. R-NUCA places private blocks in the nearest cache bank, places read-only blocks in a cluster of 4 blocks, and distributes shared data across the banks. For SP-NUCA, we augment the directory to dynamically tag blocks as shared or private (we do not assume any timing overhead). The directory executes the protocols and does the job of locating and migrating data across the blocks according to the schemes described by Merino et al. [29]. For the baseline

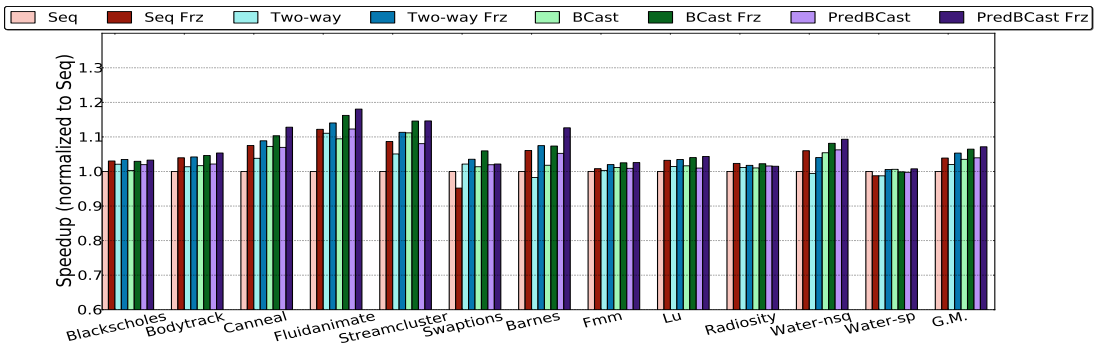

Fig. 11: Performance of different FP-NUCA policies with *Freeze* routers

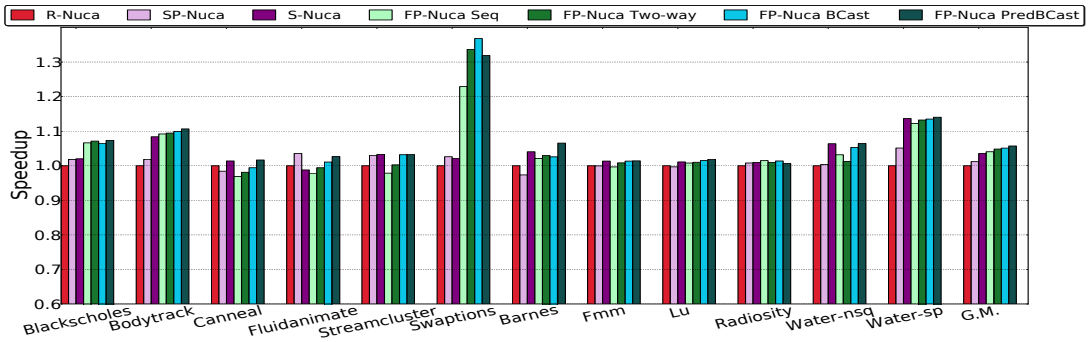

Fig. 12: Performance of different NUCA policies

S-NUCA scheme, we uniformly distribute the blocks across the banks(see [6]). Note that henceforth all our schemes **use *Freeze* routers**.

Figure 12 shows the speedups for our suite of benchmarks. We normalize all the results to R-NUCA. To summarize, we find SP-NUCA to be better than R-NUCA by 2% (G.M. speedup), and S-NUCA to be better than R-NUCA by 3.5%. The reader should note that the original papers on R-NUCA and SP-NUCA found much more positive results for database oriented workloads such as the OLTP benchmarks. However, for scientific benchmarks that have larger working sets, and different sharing patterns the results are not that favorable. All four of the FP-NUCA schemes do better than S-NUCA, SP-NUCA, and R-NUCA. *Swaptions* shows the highest speedups (30% with *PredBCast*). The other benchmarks that perform well are *Water-sp*(14.2%), *Blackscholes* (7%), *Bodytrack* (10%), and *Barnes* (8%).

Let us now try to understand the reasons behind the speedups (see Figures 13 and Figure 14). Since the L1 layer is the same, we need to concentrate on the L2 cache hit rate (Figure 13), and perceived latency (Figure 14). We found the hit rate to be mostly independent of the FP-NUCA scheme. We expect a higher hit rate with FP-NUCA because it avoids replication, and uses all the banks in the bank set to store data (rather than storing data at the set level). We indeed do observe an elevated hit rate for some of our best performing benchmarks (*Swaptions*, *Bodytrack*, and *Water-nsq*). Access latencies show a similar trend. Schemes with broadcast have roughly 20-30% lower latency. The *Two-way* policy is also faster than S-NUCA, SP-NUCA, and R-NUCA for 9 out of 12 benchmarks. There are two contributory factors for the lower latency. The first is the way that we organize data by prioritizing their placement in the home banks, and the second is the use of *Freeze* routers. The latter is the dominant factor.

5.4 Energy-Delay^2

Let us now sum up all our discussion up till now by considering the ED^2 metric. This metric encompasses both the energy consumption of the entire system (including core energy), and the performance benefits. The results are shown in Figure 15. Other than *Canneal*, and *Fluidanimate*, FP-NUCA is conclusively better. On an average *PredBCast* reduces ED^2 by 10.5% as compared to R-NUCA, and 8.7% as compared to SP-NUCA. We see the maximum amount of benefit in *Swaptions* (47%), and *Water-sp* (23%), as compared to R-NUCA. Readers can refer to Appendix B for a detailed breakup of the energy consumed by different subsystems for our suite of benchmarks, and for an explanation of the trends in energy consumption.

5.5 Synthesis Results

We synthesized the *Freeze* router using Cadence tools, and the UMC 90nm library. The results were scaled to 22nm using the results in [15]. The results are shown in Table 16. We can quickly conclude that the area

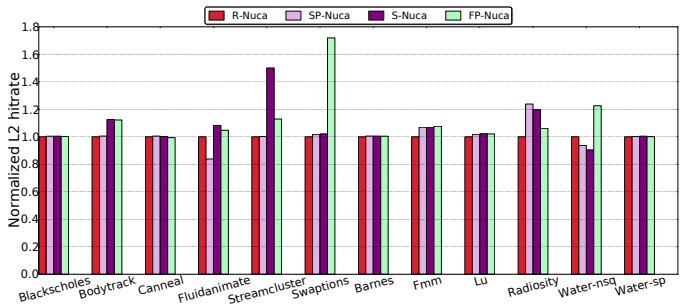

Fig. 13: L2 hit rate (normalized to R-NUCA)

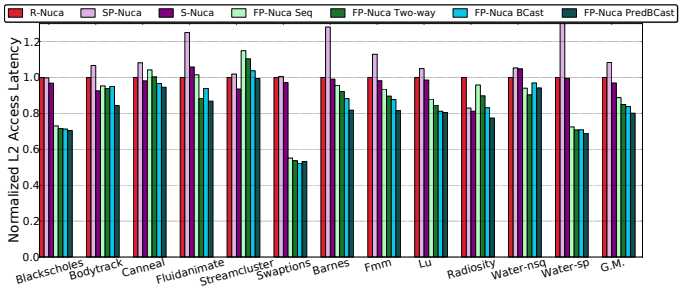

Fig. 14: L2 (NUCA) access latency (network latency + cache latency) (normalized to R-NUCA)

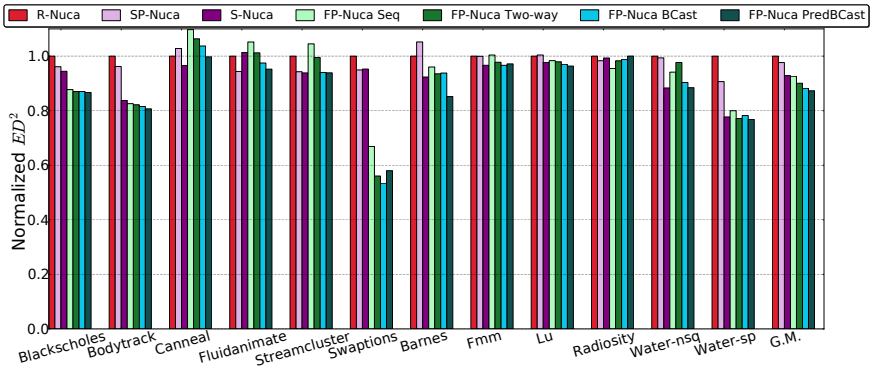

Fig. 15: ED^2 (normalized)

	Normal Router	Fast path engine
Area(μm^2)	59869.5	1643.69
Max. delay (ps) (per stage)	228.7	104.06
Percentage of sequential part	80.2	27.03
Percentage of combinational part	19.8	72.97
Power (mW)	115.01	1.76

Fig. 16: Synthesis report

overhead of the FP engine is 2.5% over a normal router, and the time taken to traverse the fast path (104 ps) is significantly lesser than a clock cycle (277 ps).

6 Conclusion

In this paper, we presented a set of schemes collectively referred to as FP-NUCA (*Freeze-Predict* NUCA). The FP-NUCA scheme is based on a novel placement strategy that prioritizes the cache bank(home bank) that is closest to the requesting core/tile in terms of Euclidean distance. By artificially restricting the communication between the home bank and other banks in the bank set, we were able to design the *Freeze* router that can fast forward L2 cache messages. Secondly, we were able to seamlessly integrate a bank predictor in our design, which can reduce the total number of messages and resultant energy consumption. The net effect of our optimizations is that the *PredBCast* policy is 6.3% faster than R-NUCA, and 5.7% faster than SP-NUCA. In terms of the ED^2 metric, it is efficient by 10.4% as compared to the R-NUCA scheme.

References

[1] N. Agarwal, T. Krishna, L. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, 2009, pp. 33–42.

[2] G. Almási, C. Caşcaval, and D. A. Padua, "Calculating stack distances efficiently," *SIGPLAN Not.*, vol. 38, no. 2 supplement, pp. 37–43, jun 2002.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT*, 2008, pp. 72–81.

[4] S. Bieschewski, J.-M. Parcerisa, and A. Gonzalez, "Memory bank predictors," in *ICCD*, 2005, pp. 666–668.

[5] L. Carloni, P. Pande, and Y. Xie, "Networks-on-chip in emerging interconnect paradigms: Advantages and challenges," in *NOCS*, May 2009, pp. 93–102.

[6] K. Changkyu, D. Burger, and S. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *Micro, IEEE*, vol. 23, pp. 99–107, 2003.

[7] C.-H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L.-S. Peh, "Smart: A single-cycle reconfigurable noc for soc applications," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 338–343.

[8] L. Chen, R. Wang, and T. Pinkston, "Critical bubble scheme: An efficient implementation of globally aware network flow control," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 592–603.

[9] Z. Chishti, M. D. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *ISCA*, 2005, pp. 357–368.

[10] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *MICRO*, 2003, pp. 55–66.

[11] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[12] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009, pp. 184–195.

[13] E. Herrero, J. González, and R. Canal, "Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors," in *ISCA*, 2010, pp. 419–428.

[14] A. Huang, J. Gao, W. Guo, W. Shi, M. Zhang, and J. Jiang, "PSA-NUCA: A pressure self-adapting dynamic non-uniform cache architecture," in *NAS*, 2012, pp. 181–188.

[15] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, vol. 31, no. 4, pp. 16–29, 2011.

[16] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A nuca substrate for flexible cmp cache sharing," in *ICS*, 2005.

[17] N. E. Jerger and L.-S. Peh, "On-chip networks," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–141, 2009.

[18] A. B. Kahng, B. Li, L. Peh, and K. Samadi, "Orion 2.0: a fast and accurate noc power and area model for early-stage design space exploration," in *DATE*, 2009, pp. 423–428.

[19] C. Kim, D. Burger, and S. W. J. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002, pp. 211–222.

[20] T. Krishna, A. Kumar, P. Chiang, M. Erez, and L.-S. Peh, "Noc with near-ideal express virtual channels using global-line communication," in *High Performance Interconnects, 2008. HOTI'08. 16th IEEE Symposium on*. IEEE, 2008, pp. 11–20.

[21] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express virtual channels: towards the ideal interconnection fabric," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 150–161.

[22] W.-C. Kwon, T. Krishna, and L.-S. Peh, "Locality-oblivious cache organization leveraging single-cycle multi-hop nocs," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014, pp. 715–728.

[23] H. Lee, S. Cho, and B. R. Childers, "Cloudcache: Expanding and shrinking private caches," in *HPCA*, 2011, pp. 219–230.

[24] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009, pp. 469–480.

[25] H. Luo, X. Xiang, and C. Ding, "Characterizing active data sharing in threaded applications using shared footprint," in *Proceedings of the The 11th International Workshop on Dynamic Analysis*, 2013.

[26] S. Ma, Z. Wang, Z. Liu, and N. Enright Jerger, "Leaving one slot empty: Flit bubble flow control for torus cache-coherent nocs," *Transactions on Computers (preprint)*, 2013.

[27] G. Malhotra, P. Aggarwal, A. Sagar, and S. R. Sarangi, "ParTejas: A parallel simulator for multicore processors," in *ISPASS*, 2014.

[28] J. Merino, V. Puente, and J. Gregorio, "ESP-NUCA: A low-cost adaptive non-uniform cache architecture." in *HPCA*, 2010, pp. 1–10.

[29] J. Merino, V. Puente, P. Prieto, and J. A. Gregorio, "SP-NUCA: a cost effective dynamic non-uniform cache architecture," *SIGARCH Comput. Archit. News*, vol. 36, pp. 64–71, May 2008.

[30] M. Modarressi, A. Tavakkol, and H. Sarbazi-Azad, "Virtual point-to-point connections for nocs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 6, pp. 855–868, June 2010.

[31] R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian, "Leveraging bloom filters for smart search within nuca caches," in *7th Workshop on Complexity-Effective Design (WCED)*, 2006.

[32] M. Stensgaard and J. Sparso, "Renoc: A network-on-chip architecture with reconfigurable topology," in *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, April 2008, pp. 55–64.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[34] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 336–345, May 2005.

Anuj Arora is a Master's student at the Department of Computer Science & Engg, Indian Institute of Technology, Delhi. He obtained his Bachelor's degree in Computer Science from Guru Gobind Singh Indraprastha University in 2012. His research interests include non-uniform cache designs for manycore architectures.

Mayur Harne received his Master's degree in Computer Science from the Indian Institute of Technology, New Delhi in 2013. After his graduation, he has been working at NVIDIA Graphics Pvt. Ltd in the Android video driver team as a System Software Engineer.

Hameedah Sultan is pursuing her Master's degree in VLSI Design from IIT Delhi. She has completed her Bachelor's degree in Electronics Engineering from Aligarh Muslim University in 2013. Currently she is working in the area of thermal and noise simulation.

Akriti Bagaria is a Master's student at the Department of Computer Science & Engg, Indian Institute of Technology, Delhi. She obtained her Bachelor's degree in Computer Science from Guru Gobind Singh Indraprastha University in 2012.

Smruti R. Sarangi is an Assistant Professor in the Department of Computer Science and Engineering, IIT Delhi, India. He has spent four years in industry working in IBM India Research Labs, and Synopsys. He graduated with a M.S and Ph.D in computer architecture from the University of Illinois at Urbana-Champaign in 2007, and a B.Tech in computer science from IIT Kharagpur, India, in 2002. He works in the areas of computer architecture, parallel and distributed systems. Prof. Sarangi is a member of the IEEE and ACM.

APPENDIX A
OPTIMALLY SIZING THE OMQ AND CRQ

Benchmark	Max. OMQ Size	Max. CRQ Size
Blackscholes	9	7
Bodytrack	8	8
Canneal	6	5
Fluidanimate	7	7
Streamcluster	6	4
Swaptions	9	7
Barnes	6	4
Fmm	7	6
Lu	8	6
Radiosity	4	3
Water-nsq	7	5
Water-sp	6	4

TABLE 4: Maximum sizes of the OMQ and CRQ

Table 4 shows the maximum occupancy of the OMQ and CRQ observed during our simulations for each benchmark. We observe that the maximum occupancy of the OMQ never exceeds 9, and the maximum occupancy of the CRQ never exceeds 8. Hence, if we set the size of the OMQ to 16 entries, and the size of the CRQ to 10 entries, then we will never have a situation in which these queues fill up, and there is a deadlock/starvation situation in our simulations. However, it is possible to have pathological situations where these queues fill up and there is starvation in the system because of unfulfilled requests. We shall discuss methods to handle such issues in Appendix D.

APPENDIX B
ENERGY CONSUMPTION

Benchmark	NOC + L2 cache Energy (%)	Benchmark	NOC + L2 cache Energy (%)
Parsec		Splash	
Blackscholes	2.60	Barnes	2.70
Bodytrack	4.37	Fmm	3.29
Canneal	48.29	Lu	1.91
Fluidanimate	20.86	Radiosity	0.52
Streamcluster	9.72	Water-nsq	7.10
Swaptions	24.08	Water-sp	3.55

TABLE 5: Percentage of (NOC + L2 cache energy)

Let us now compare the FP-NUCA schemes with other schemes in terms of energy consumption. Our contributions are mainly in reducing the number of NOC requests, and the L2 cache accesses. Hence, let us first focus on the combined energy of these two subsystems. Table 5 shows the relative percentage of energy in the NOC and L2 cache as a function of the energy consumption of the entire system (processor + NOC + cache + clock). The Parsec benchmarks have a much higher value of energy consumption. *Canneal* is an outlier (48.29%). This is because it has the lowest L1 and directory hit rates among all the benchmarks. Hence, it has the maximum number of requests reaching the L2 cache (see Table 3). The (NOC+L2 Cache) account for 10-20% of the total energy in *Fluidanimate*, *Streamcluster*, and *Swaptions*. *Blackscholes* and *Bodytrack* have much lower NOC activity (and energy also). In comparison, the energy consumption of these subsystems in Splash benchmarks is much lower. Here, *Radiosity* is an outlier (0.52%). This is because it has the least number of messages reaching the L2 cache. For the rest of the benchmarks, the percentage varies from 1.91 to 7.10 %.

Let us now evaluate the energy consumption for all the FP-NUCA schemes (with *Freeze* routers) and compare them with R-NUCA, SP-NUCA, and S-NUCA (see Figure 17). Here, again we use R-NUCA as the baseline, and show normalized values. We can draw a quick conclusion that the (L2 cache + NOC) in SP-NUCA is 12.1% more energy consuming. SP-NUCA consumes more energy because it occasionally resorts to broadcasting a message to all the banks. S-NUCA is the best scheme because it has the simplest access protocol. FP-NUCA schemes increase the energy dissipation of these two subsystems by 30.69-61.41% on an average. The only benchmarks that show a reduction in (NOC+L2 Cache) energy for FP-NUCA schemes are *Blackscholes*, *Swaptions*, and *Water-sp*. These are exactly those benchmarks that have a very high percentage of hits in the home bank $> 90\%$. Since additional messages are not sent, the energy consumption is lower than SP-NUCA and R-NUCA. Let us now compare the FP-NUCA schemes among themselves. Predictably, *BCast* is the worst (17.16% more than *Seq*). *Two-way* and *PredBCast* have roughly similar energy consumption, and both of them are better than *Seq*. Hence, we can conclude that we are justified in creating the novel schemes from the point of view of energy consumption.

APPENDIX C
HARDWARE IMPLEMENTATION

C.1 Hardware details

In order to determine the hardware requirements of our proposed router, we implemented it in VHDL and synthesized the design.

We add a *fast path* bit in every message. If it is 1, then it is a fast path message, otherwise it is a regular message. Now, if an incoming message is a *fast path* message (determined by the *fast path* bit), then it is sent to the fast path engine in the respective direction (north, south, east, west and home).

If it is a regular message, it is stored in a flit input buffer, which was synthesized using flip-flops. We create designs with two different flit widths: 16 bytes and 32 bytes. From the input buffer, the messages are sent to virtual channels. There are 8 virtual channels per physical port in each direction, each of which has a 4-flit deep input buffer. Since we use lookahead routing, the virtual channels are allocated in the previous stage. Consequently, the id of the virtual channel is already

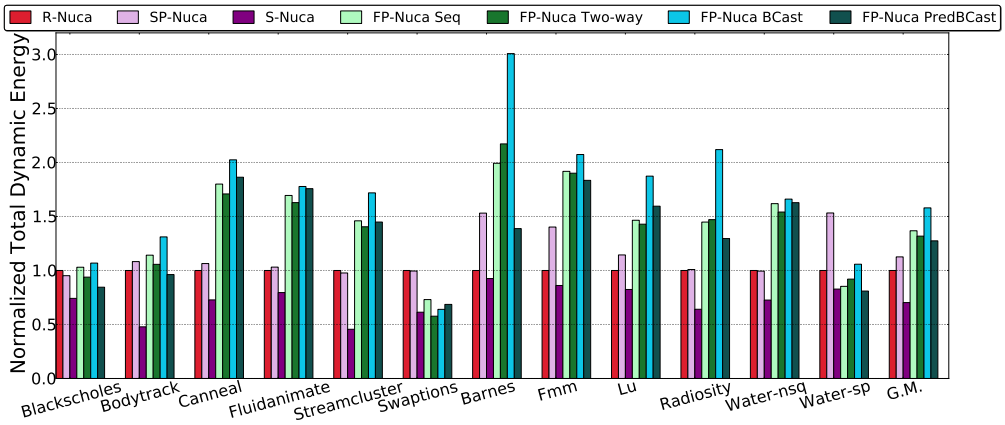

Fig. 17: Network + L2 cache energy (normalized)

a part of the message and this id is used to route the message to the respective virtual channel.

Simultaneously, we need to compute the route and virtual channel id for the next hop. We send the destination (6 bits) to the route computation engine. The route computation engine implements dimension ordered X-Y routing. It consists of a comparator that compares the column id of the router with the column id of the destination. If the columns are different, then it routes the message to the left or the right based on the sign bit of the result of the comparison. If the columns are the same ('zero' flag of the comparator is set), then we compare the row number of the router with the row number of the destination. Based on the sign of the result, we compute the route (up or down).

We implement each virtual channel as a queue. Dedicated head and tail registers maintain the start and end indices. Each queue also has a register called *size* that maintains its current size. If the virtual channel is empty then flits can bypass the queue, and directly proceed to the switch allocation stage. They are buffered in pipeline latches (also referred to as flit buffers). Messages subsequently pass through flit buffers and then a 5×5 crossbar switch. Allocation of switch ports is based on a round-robin policy.

Fast path messages do not traverse the normal router's pipeline, and pass only through the fast path engine (which is a mostly combinational block, with the clock signal going only to the *'decrement hops left'* block). Hence, *fast path* data can be obtained at the output of the FP engine very quickly. The FP engine also sets *FP enable* signals, which gate the clock of the normal router, to freeze it for a cycle when a fast path message has to be transmitted. The input and credit based flow control buffers of the normal router are not controlled by the gated clock and hence they can keep accepting new data on every clock cycle. Finally the *FP enable* signals generated by the FP engine act as select signals for the output multiplexers in each direction, to choose between regular data and *fast path* data.

C.2 Synthesis results

Simulation results show that the functionality of the router obtained matches the expected functionality. We synthesized the circuit in UMC $90nm$ technology with the 90 nm standard cell library using the Cadence Encounter RTL Compiler, and performed timing, area and power analysis. The clock frequency was set to 450 ps. This was sufficient to satisfy the timing requirements. We found out that around 20% of the area is due to the combinational logic and the remaining 80% area is occupied by the sequential part of the circuit. The critical path lies inside the regular router, and the freeze path circuitry does not adversely affect the operating frequency of the circuit.

Next, we scaled the area and power consumption to 22 nm technology using the ITRS scaling method outlined in [15]. For scaling the delay, we employ the methods used by the McPat simulator [24]. Taking into effect the tremendous increase in chip power density for high frequencies, they suggest increasing the clock frequency by 15% every generation. We use the same scaling factor for the delay. Hence, we scaled the area by 50%, delay by 15% and power by 0.527 for each technology node. The area, delay and power consumption obtained for flit sizes of 16 $bytes$ and 32 $bytes$ are listed in Table 6.

We use a frequency of 3.6 GHz. The clock period is thus 278 ps. From Table 6, we observe that for both the 16 byte and 32 byte flit routers, the critical path in any pipeline stage fits within a clock cycle. We can additionally afford a 10% timing margin. The fast path engine is more than twice as fast as the regular path. The high fanouts of control signals such as *FP enable* did cause a problem. However, we were able to reduce the timing overhead by using a tree of inverters. The area overhead of the FP engine is minimal (roughly 2-3% of the router area). As expected the regular router mostly has sequential elements, and the FP engine mostly has combinational elements. Lastly, the power consumption of the FP engine is also negligible (roughly 2% of the regular router).

	Flit size = 16 $bytes$		Flit size = 32 $bytes$	
	Normal Router	Fast path engine	Normal Router	Fast path engine
Area(μm^2)	59869.5	1643.69	119358.34	3338.88
Max. delay (ps) (per stage)	228.7	104.06	257.29	117.21
Percentage of sequential part	80.2	27.03	80.0	25.77
Percentage of combinational part	19.8	72.97	20.0	74.23
Power (mW)	115.01	1.76	196.19	3.01

TABLE 6: Power consumption, delay and area of the fast path circuitry and ordinary router for flit sizes of 16 $bytes$ and 32 $bytes$ (scaled to 22 nm)

APPENDIX D
ADDITIONAL SOLUTIONS FOR HANDLING DEADLOCK AND STARVATION

D.1 Deadlock

Deadlock might occur in the *Two-way* and the *Seq* protocols if two or more CRQs are full and there is a circular wait. In the *BCast* protocol, deadlock cannot occur as there is no inter-CRQ communication. To prevent deadlocks, we need to augment our existing protocol. Note that in the *Two-way* and the *Seq* protocols, a cycle always consists of two nodes, because in these protocols a request is forwarded only to the adjacent nodes. We take the size of a CRQ to be slightly more than the largest CRQ occupancy observed (which in our case was 9). Therefore there is a miniscule chance that there will actually be a deadlock (we can still have one though).

We propose two additional solutions for handling a deadlock. The first is a conventional solution, and the latter is our novel solution. We augment the latter solution to handle starvation also in Section D.2.

D.1.1 Conventional Solution: Solution 1

The first solution is to allow the simulation to enter into a deadlock, detect it and then recover from it. The key consideration in this design is that we want to move requests quickly when there is no deadlock and avoid the overhead of deadlock handling and detection as much as possible. We employ a deadlock detector, which tracks the allocation and state of CRQs. It uses the standard method of viewing the various CRQs as nodes in a directed graph. We envision a bank set specific deadlock detector. Each CRQ is a node, and there can be edges between nodes. There is an edge from node A to node B if A waits for B to free an entry. There might be a deadlock if the graph contains a cycle.

The mechanism is as follows. If a request (either in the OMQ or CRQ) cannot be allocated an entry in the CRQ of a neighboring bank, then it waits for the other CRQ to free an entry for a time period equal to $DeadlockTimeout$ (100 cycles is a representative value). If the period, $DeadlockTimeout$, expires, then the HBC invokes the deadlock detector. The deadlock detector queries all the CRQs to find dependences. It adds an arrow between two CRQs if one CRQ is waiting for the other. If there is a cycle, then there might be a deadlock. We can have some race conditions here. It is possible that when a cycle is created, one of the edges ceases to exist. We might incorrectly infer a deadlock (false positives are possible). However, this will not cause any correctness issues. It will just add some extra timing overhead because the deadlock recovery protocol will be invoked.

This protocol assumes that deadlocks are rare (which is the case if the CRQ and OMQ are sized appropriately). The deadlock detector broadcasts a message to clear all the CRQs, and kill all the messages in flight. Subsequently, it locks all the OMQs, and does not allow any new request to enter them. During this time, if any core sends a message to a locked OMQ, then it gets a NACK message as a reply. It retries later. The deadlock recovery circuit drains all the OMQs one by one in a sequential fashion. During this period, only one request in the entire bank set can be processed. All the other requests wait. Once all the OMQs are empty, normal operation can resume.

D.1.2 Our Novel Solution: Solution 2

The other alternative is to not allow the system to enter into a deadlock. To ensure this, we introduce an *Exchange* message. Assume that a request needs to go to its neighboring CRQ, and it cannot go because the CRQ is full. For example, if a message in node A's CRQ needs to go to node B's CRQ, and B's CRQ is full, then the message at the head of A's CRQ needs to wait. Here, a deadlock situation is possible, if B is also trying to send a message to A's CRQ, and A's CRQ is full. In fact, any deadlock in our protocol will always involve a cycle between neighboring CRQs because we only send messages to neighboring nodes. If we analyze this situation carefully, we can observe that we can solve this deadlock situation if we allow A and B to exchange messages at the head of their queues. The occupancy of the queues will remain the same. However, there will be no deadlock. We thus use the *Exchange* message here. Whenever a message is stuck in A's CRQ because it is not able to enter into B's CRQ, we send an *Exchange*

message to B. If B is also waiting on A, then it replies with its data to A. A then sends the message at the head of its CRQ to B, and the deadlock is resolved.

Since any deadlock involves a cycle in the dependence graph between neighboring nodes in our protocol, this mechanism guarantees that we will not have any deadlocks during the operation of the algorithm. This strategy does not require a dedicated deadlock detection and recovery engine.

D.2 Starvation

Let us now consider the issue of starvation. Starvation is defined as any single request not being able to make progress. It is perfectly possible for requests in other OMQs or CRQs to make progress. Note that starvation freedom implies deadlock freedom. However, the converse is not true. Let us assume a system that avoids deadlocks using the *Exchange* message (as proposed in Section D.1.2). We shall slightly augment this system to recover from starvation related issues.

Let us now look at the some of the scenarios that can cause starvation. CRQs can lead to permanent starvation of OMQs in the *Two-way* and the *Seq* protocols. Here, a request in the OMQ can starve as the CRQ may be busy accepting requests from its neighbors indefinitely. Similarly, a request in a CRQ can starve because the neighboring CRQ is full and busy processing other requests. Starvation is also possible in the broadcast protocol. Here, a request in the OMQ can be denied an entry in the CRQ indefinitely because the CRQ may keep accepting new requests from other OMQs.

To prevent starvation, we incorporate a starvation counter. The purpose of the starvation counter is to check if a request is near starvation. If a request has been stalled for a certain number of clock cycles (sct or starvation count threshold, 100 cycles in our design), then the HBC sends a special message to the HBCs of other banks. This messages instructs all the OMQs to freeze, allow all outstanding messages in the CRQs of the bank set to reach their destinations and then resume. We have already proved in Section D.1.2 that by using an *Exchange* message this process is guaranteed to terminate.

Now, if the size of each CRQ is n and the number of banks in a bank set is b, then we claim that all the requests present in the CRQs can be serviced in at most $O(nb^2)$ cycles in the *Seq* and *Two-way* protocols, and in $O(n)$ cycles in the *BCast* protocol.

Proof: The maximum number of requests present in the CRQs can at the most be nb. In the broadcast protocol, in one cycle b requests are serviced (if we assume that the bank is pipelined). Hence, in $O(n)$ cycles all nb requests will be serviced.

In the *Two-way* and *Seq* protocols, at least one request is guaranteed to move (either to some adjacent CRQ or to the HBC for local processing in case of a hit) in an interval of T cycles. Here T is the time it takes to perform an *Exchange* operation between CRQs. This is because we cannot have a circular wait between the CRQs since they perform an *Exchange* operation whenever there is a cyclic dependence between adjacent CRQs. Now, each message can at the most move $b-1$ hops. Thus, the total number of hops that all the messages in all the CRQs can traverse is $nb \times (b-1)$. Since there is at least one movement every T cycles, the total number of cycles it will take for all the requests is $O(nb^2)$.

APPENDIX E
COMPARISON OF IPC FOR DIFFERENT FLIT SIZES

In this section, we study the relationship between the flit size and the gain in IPC for the *Bcast* configuration with *Freeze* routers. We compare the *BCast* configuration with the R-NUCA configuration for the same flit size. Figure 18 plots the relative speedup (vs. R-NUCA) for three different flit sizes: 8, 16, and 32 bytes. Let us first comment on the decrease in gains. The mean speedup with 32 bytes was 7%. This number reduced to 6.3% for a 16 byte flit, and 4.7% for a 8 byte flit. The trends for other configurations are the same.

There are two reasons for a reduction in the speedup as compared to R-NUCA, which also uses flits of the same size. The first reason is that it takes longer for receiving the entire packet, if there are additional flits. However, this was not a very significant issue in our design because the additional 2-4 cycle latency was insignificant in a large NOC. Another additional factor is that the Tejas simulator by default supports the critical word fetch, and early-restart schemes. These schemes propose to send the requested words first to the L1 cache and processor such that it can resume its operation. The rest of the words in the cache line subsequently follow. Here, also the order of flits is important. The simulator tries to leverage spatial locality by sending the adjacent words first. Since the critical data is the first to be received, the RC (requesting core) was found to be relatively immune to the number of flits in the response message.

The second and most important reason for a reduction in the speedup is that a message with more flits delays other colliding slow path messages. For example, with 32 byte flits, we require 3 flits to transmit a cache line (including the head flit), and with 16 byte flits we require 5 flits. Another message colliding with a FP message will be delayed by 3 or 5 cycles for both the cases respectively. The additional delay of 2 cycles with 16 byte flits can negatively impact performance.

We observed one interesting effect that occasionally worked in our favor. For most of the benchmarks (*blackscholes, bodytrack, canneal, fluidanimate, swaptions, barnes, fmm, water-nsquared, water-spatial*), the *BCast* configuration gets better than R-NUCA with an increase in

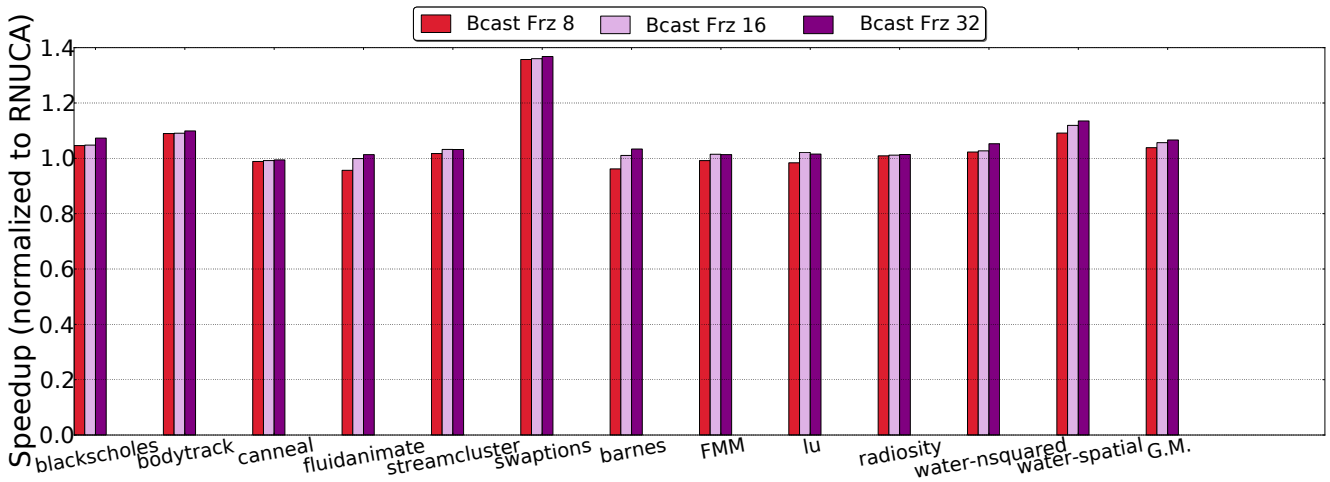

Fig. 18: Relative speedups for flit sizes of 8, 16 and 32 bytes

flit size. However, for some benchmarks (*lu*, *srtreamcluster*) the trends are reverse. Additionally, the expected slowdown due to the delaying of slow-path flits was less than our expectation. This is because, we observed that by introducing short delays in *slow path* flit streams, we were occasionally able to avoid contention: situations where a router runs out of credits, and can thus not send messages upstream. Sometimes a small delay in the flit stream avoids such situations. A similar effect has been observed by Chen et al. [8], and Sheng et al. [26].