Optical Overlay NUCA: A High Speed Substrate for Shared L2 Caches

Eldhose Peter, Anuj Arora, Janibul Bashir, Akriti Bagaria and Smruti R. Sarangi, Indian Institute of Technology Delhi

In this paper, we propose to use optical NoCs to design cache access protocols for large shared L2 caches. We observe that the problem is unique because optical networks have very low latency, and in principle all the cache banks are very close to each other. A naive approach is to broadcast a request to a set of banks that might possibly contain the copy of a block. However, this approach is wasteful in terms of energy and bandwidth. Hence, we propose a set of novel schemes in this paper that create a set of virtual networks ($overlays$) of cache banks over a physical optical NoC. We search for a block inside each overlay using a combination of multicast and unicast messages. We first propose two simple protocols: *TSI* and *Broadcast*. The former uses unicast messages and the latter uses multicast messages. We subsequently, propose an improved scheme, *OP_BCAST*, that combines the best of *TSI* and *Broadcast*, and mainly uses restricted multicast messages. We subsequently propose a set of novel hardware structures for creating and managing overlays, for efficiently locating blocks in the overlay, and for implementing dynamically changing overlays with *OP_BCAST*. The performance of the *TSI* scheme is within 2-3% of a broadcast scheme, and it is faster than traditional schemes with electrical networks by 26%. As compared to the broadcast scheme it reduces the number of accesses, and consequently the dynamic energy of the caches by 6-8%. *OP_BCAST* is 34% faster than the best solutions with copper based NoCs; moreover, it reduces the dynamic energy for cache access by 33% as compared to the *TSI* scheme.

CCS Concepts: •**Computer systems organization** → **Interconnection architectures; Multicore architectures;**•**Hardware** → **Emerging optical and photonic technologies;**

Additional Key Words and Phrases: On chip optical networks, NUCA protocol, overlay network

ACM Reference Format:
Eldhose Peter, Anuj Arora, Janibul Bashir, Akriti Bagaria and Smruti R. Sarangi, 2016. Optical Overlay NUCA: A High Speed Substrate for Shared L2 Caches. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 25 pages.
DOI: 0000001.0000001

1. INTRODUCTION

The [1] last level cache (LLC) in modern processors has been found to be critical to performance. As a result, there is a plethora of research [Changkyu et al. 2003; Hardavellas et al. 2009; Kim et al. 2002; Winkle et al. 2016] in the community to propose efficient algorithms for managing large shared LLCs. Most of these proposals are classified as non-uniform cache architectures (NUCA caches), which are cognizant of the fact that the latency of fetching a line from the LLC is variable and is dependent upon the position of the bank that contains it in a large multibanked cache. Prior work in NUCA

[1]This is an extension of a conference paper: Eldhose Peter et al., Optical Overlay NUCA: A High Speed Substrate for Shared L2 Caches, Eldhose Peter, Anuj Arora, Akriti Bagaria, Smruti R. Sarangi, HiPC 2014 [Peter et al. 2015a].

Author's addresses:Eldhose Peter, Anuj Arora, Janibul Bashir, Akriti Bagaria and Smruti R. Sarangi, Computer Science Department, Department of Computer Science and Engineering, IIT Delhi, Hauz Khas, New Delhi -110016. E-mails: {eldhose,mcs122812,csz158489, mcs132541,srsarangi}@cse.iitd.ac.in
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2010 ACM. 1539-9087/2010/03-ART39 $15.00
DOI: 0000001.0000001

caches contains sophisticated protocols for locating, migrating, and replicating cache lines in a large shared NUCA cache. The problem of effectively managing a NUCA cache is particularly interesting with traditional electrical networks because the on-chip latencies can be fairly large, especially with a complex NoC (network-on-chip). For example, in a 32 core chip, with 32 cache banks it can take up to 42-56 cycles (@3.4 GHz) for a flit to be transmitted between 2 cores.

In this paper, we look at the problem of designing large shared LLCs with optical networks with dynamic NUCA (DNUCA) protocols. To the best of our knowledge there is no other work in this area. Optical networks are very different from electrical networks in the following aspects [Haurylau et al. 2006; Kirman et al. 2006] (see Sections 2 and 3). First, they have very low latencies. It is possible to send a message across the chip in less than 2-3 cycles. Second, they have very high bandwidth owing to the possibility of wavelength division multiplexing (sending multiple wavelengths(bits) along an optical channel), and finally, some variants of optical networks naturally support multicast and broadcast based traffic without any additional messaging overhead. All of these features have been used to design coherence protocols in the past [Cianchetti et al. 2009; Kirman and Martínez 2010; Kurian et al. 2010; Morris et al. 2014].

The crux of our idea is as follows. In a traditional NUCA cache, we create a set of banks called a *bank set*, and have the flexibility of migrating cache lines across banks in a bank set. Due to routing constraints the design of bank sets is constrained (linear, or square shaped). However, with optical networks we can create arbitrary shaped bank sets, and we can use this fact to design extremely efficient LLCs. Moreover, banks in a bank set need not be physically close to each other. We treat such bank sets as virtual networks, and refer to them as *overlays* in this paper. In Section 4, we propose three schemes to create and manage such overlays in this paper. We find the scheme, *OP_BCAST*, to be the best. Additionally, this scheme supports dynamic overlays where the constituent banks in an overlay can be changed at runtime (see Section 5). The main components of our schemes are block location, migration between banks, and managing block evictions when an overlay is dynamically changed.

For a suite of Parsec [Bienia et al. 2008] and Splash2 [Woo et al. 1995] benchmarks we demonstrate a mean speedup of 34% over some of the best electrical NUCA protocols, and a speedup of 7% over an optical network with traditionally designed overlays (organized linearly) [Peter et al. 2015a] in Section 6. For additional details of the design, details of the VHDL based hardware implementations, and additional results, interested readers can refer to the online appendices [Appendices, Peter et al.].

2. BACKGROUND AND RELATED WORK

2.1. Optical Communication

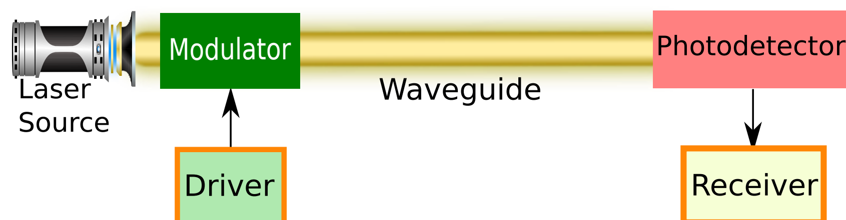

Fig. 1: An optical communication system

The basic components of an optical communication system are shown in Figure 1. Every optical network needs a light source, which can either be an off-chip laser, or an on-chip VCSEL based laser. The optical power is then distributed to each transmitting station via an optical network (typically referred to as the power waveguides). The transmitters and receivers can be connected in various topologies. In this paper, we consider the SWMR (single writer multiple reader) topology [Pan et al. 2009], where each transmitter is connected to all the receivers via a set of waveguides (optical channels). Transmitters modulate the power sourced from the power waveguides, and send it to the receivers, who use photodetectors to infer the strength of the optical signal.

Optical networks such as the SWMR network do not require routers and any internal buffering. As a result, they are very fast and it is possible to send messages between any pair of stations in less than a few cycles. Additionally, we can multiplex 64 different wavelengths [Vantrease et al. 2008] in the same channel to significantly increase the available bandwidth. However, optical networks have a major shortcoming, which is high static power dissipation [Demir and Hardavellas 2014; Morris et al. 2014; Zhou and Kodi 2013]. The laser is always turned on irrespective of the traffic in the network. There are a host of laser modulation schemes proposed in prior work such as ColdBus [Peter et al. 2015b] and Probe [Zhou and Kodi 2013] to reduce the laser output based on predictions of future traffic. Any of these laser modulation schemes can be used with our approach.

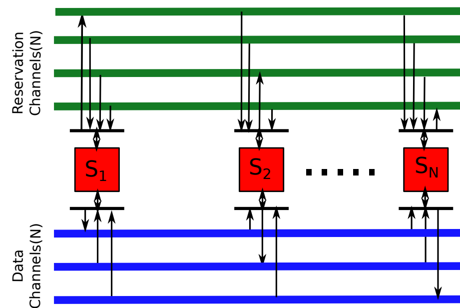

Fig. 2: Reservation assisted SWMR

To reduce the energy associated with receivers that are on all the time, we adopt the R-SWMR scheme originally proposed by Pan et al. [Pan et al. 2009](See Figure 2). Here, we have an additional waveguide called a *reservation waveguide* between a transmitter and its receivers. All the receivers are turned off by default. Before transmitting a message, we send a message along the reservation waveguide that contains the ids of the receivers that need to be turned on. The corresponding receivers turn themselves on(*on denotes ready to accept messages*). Subsequently, we transmit a message on the main waveguide, and all the receivers that have turned on get the message. After receiving the entire message, the receivers turn themselves off.

2.2. Uses of Optical Networks

Let us quickly look at the uses of optical networks proposed in literature.

2.2.1. On-Chip and Off-chip Communication. Optical interconnects are the ideal ingredients for creating a high bandwidth and low latency on-chip communication substrate. There are several important works in this area that propose ultra-high throughput optical interconnects. We list a few in this section due to a lack of space. Corona [Vantrease et al. 2008] is one of the most important proposals in this area

that proposes a ring based interconnect for connecting 256 cores. There are separate waveguides for unicast and broadcast traffic. Firefly [Pan et al. 2009] extends the idea by proposing a partitioned crossbar based architecture. It additionally has optimizations to save energy by turning off receivers. Recently, Koohi et al. [Koohi and Hessabi 2014] have proposed a ring shaped optical waveguide based architecture for on-chip communication using optical data and control planes. This technology uses all optical switches which route the data based on their wavelength. Some more references in this area are as follows: Morris et al. [Morris and Kodi 2010], and Somayyeh et al. [Koohi et al. 2014].

2.2.2. Cache Coherence Protocols. Optical broadcast protocols are ideally suited for implementing cache coherence because of their ability to easily support broadcast traffic. One of the earliest works by Kirman et al. [Kirman et al. 2006] investigates the potential of optical technology for creating a low latency and high bandwidth snoopy coherence protocol. Cianchetti et al. [Cianchetti et al. 2009] propose a hybrid electrical/optical network for large scale cache coherent multiprocessors. They also use a snoopy based protocol. In 2010, Kirman et al. [Kirman and Martínez 2010] proposed an improved all-optical network using wavelength based oblivious routing to implement a MESI-based snoopy coherence protocol. In comparison, ATAC [Kurian et al. 2010] uses a directory based cache coherence protocol, *ACKwise*, which uses a broadcast mechanism to provide high scalability and performance in a 1024 core system. Vantrease et al. [Vantrease et al. 2011] provide the details of a framework for implementing a simple cache coherence protocol that minimizes the number of transient states. They use a novel idea called atomic coherence that uses optical mutexes and proposes a six state protocol for cache coherence, thereby improving performance and reducing complexity. Additionally optical networks have been used to implement barriers [Binkert et al. 2009] and perform arbitration [Vantrease et al. 2009]. However, we are not aware of any proposals that use optical networks to manage the last level cache (LLC). Let us now discuss some of the related work in this area that uses electrical networks.

2.3. Non-uniform Cache Architectures for L2/L3 Caches

For large L2 and L3 caches, it is typically necessary to design a complex protocol for locating and migrating cache lines across banks. Such architectures are referred to as non-uniform cache architectures (NUCA caches) because the latency of searching for a line is not a constant. Since there is no related work in this area that uses optical networks, let us summarize the proposals that use electrical networks.

Kim et al. [Changkyu et al. 2003] in their seminal work proposed to divide a large cache into a number of cache banks. Here lines are statically mapped to banks, and there are no migrations. A later approach, D-NUCA [Kim et al. 2002], extends the idea to create sets of banks, where a cache line can be mapped to any bank in a given *bank set*. Moreover to reduce the access latency, blocks can migrate in a bank set. Subsequent approaches started further specializing this approach to consider different classes of data: shared, private, read-only, and instructions. In particular, the R-NUCA [Hardavellas et al. 2009] scheme leveraged such patterns and placed data in different types of sets based on its class. An alternative approach adopted by Merino et al. [Merino et al. 2010] (SP-NUCA) looked at only two kinds of classes: private and shared. A core first checks its nearest bank that most likely contains private data, and then searches remote banks. There are additional methods to further optimize this process by predicting the bank that contains a line, and smartly searching the set of banks (see [Arora et al. 2015]).

The important point to note is that these solutions cannot be used for optical networks because they are designed for a network where the latencies of banks can differ

by up to 10X. In comparison, all the banks in an optical network are reachable within 2-4 cycles (depending on the technology). As a result traditional smart search, duplication, and migration schemes are not effective. We need to design a new protocol.

3. PROPOSED OPTICAL NETWORK

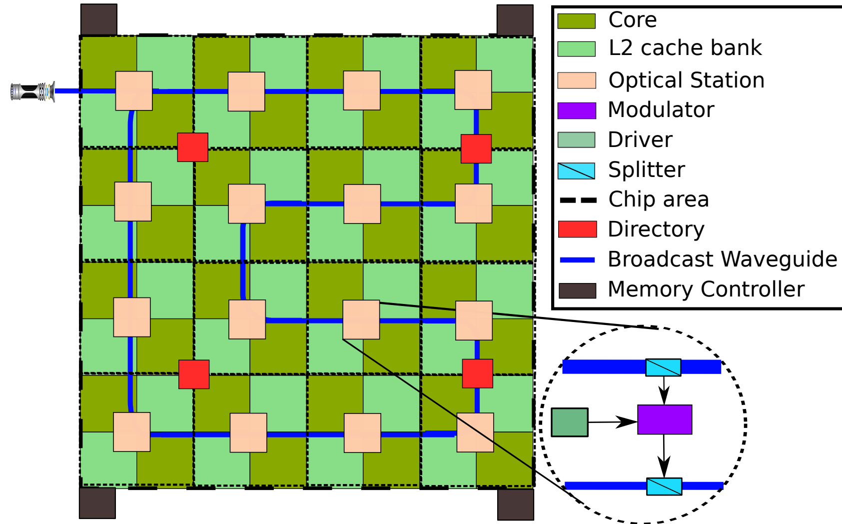

Fig. 3: Overview of the system

The proposed system is shown in Figure 3. We have 32 cores, each with a private L1 cache. The L2 is a shared NUCA cache, with 32 cache banks. All these elements are arranged as a set of tiles, and the cores and cache banks are arranged as a chess board. Each tile consists of 2 cores, and 2 cache banks. We use a distributed directory-based coherence protocol, with the 4 directories (empirically found to be the best choice) placed as shown in Figure 3. We place one optical station (transmitter/receiver) in each tile.

3.1. Optical Communication Infrastructure

3.1.1. Reservation Channel. We have a total of $N = 16$ optical stations and they are connected using a serpentine structured waveguide as shown in Figure 3. Before sending a message, it is necessary to turn on the receivers of the destination stations, and also inform them about the type of the message such that they know for how long they need to keep themselves turned on. To setup the destinations, the transmitter needs to broadcast a bit vector that is N bits long to all the stations. Out of these N bits, we use 1 bit to indicate the type of the message (data or control), and use the rest of the $N-1$ $(= 15)$ bits to indicate the set of receivers that need to be turned on. If a receiving station finds its corresponding bit to be 1, then it turns on the receiver, and also reads the type of the message. We can send 64 different wavelengths using DWDM (dense wavelength division multiplexing) on the same waveguide. Thus, in this case, 4 stations can share the same waveguide and send 16 bits each using different wavelengths to the rest of the stations. Consequently, for implementing the reservation waveguide we need a bundle of 4 waveguides. In comparison, Firefly [Pan et al. 2009] uses unicast messages in its reservation waveguides.

The additional bit, *Type of Message -TM Bit*, denotes whether the upcoming message is a control (1 flit) or data message (5 flits). Depending on this bit, the receiver stays turned on to receive 1 or 5 flits respectively (See Section 4.7). According to the authors of Firefly [Pan et al. 2009], the area and power overheads of these reservation channels are insignificant.

Optical Parameters	
Wavelength (λ)	$1.55\mu m$
Width of waveguide (W_g)	$3\mu m$
Slab height	$1\mu m$
Rib height	$3\mu m$
Refractive Index of $SiO_2(n_r)$	1.46
Refractive Index of Si (n_c)	3.45
Input Driver Power	76 μW
Output Driver Power	166 μW
Insertion Coupling Loss	50%
Output Coupling Loss	13%
Photodetector quantum efficiency	0.8 A/W
Photodetector minimum power	36 μW
Combined transmitter and receiver delay	180-270 ps
Optical propagation delay	7 ps/mm
Electrical propagation delay	35 ps/mm
Bending Loss	1 dB
Waveguide Loss	1 dB
Coupler Loss	1 dB
Photodetector	0.1 dB
Wall Plug Efficiency	20 %
Splitter Loss	0.36 dB
Ring Heating	$26\mu W/ring$
Ring Modulation	$500\mu W/ring$

Table I: Optical Parameters [Kirman et al. 2006; Morris et al. 2014; Reed 2008]

3.1.2. Data Channel. We transfer data using a double pumped clock (transfer data at both the rising and falling edges, similar to Corona [Vantrease et al. 2008]). We can thus transmit (128 bits or 16 bytes) of data per CPU clock cycle. Note that optical channels can support signal frequencies up to 40 GHz, and thus the frequency is not the bottleneck here. We thus require 1 waveguide per station for transmitting data. Since 4 stations share a reservation waveguide, we in effect require 1.25 waveguides per station.

As discussed earlier, there can be two types of messages – *data* and *control*. The cache line width in our architecture is 64 bytes and we can thus send a cache line in 5 flits, which includes the head flit. The total bandwidth of the system is equal to $2 \times 3.4GHz \times 16 \times 64$, which is 0.87 TBPS. Control messages are single flit messages containing 100 bits, which can be sent in a single cycle (128 bits/cycle – double pumped). The optical parameters that we have considered are shown in Table I.

4. OPTICAL OVERLAYS

In a traditional NUCA based electrical network, cache banks are divided into bank sets. The bank sets are typically arranged in columns. A cache line can be in any bank in a bank set. Most NUCA protocols typically assign a *home bank* to a cache line, and if the line is not found in the home bank, then they search other banks in the same bank set. We can consider each bank set to be an *overlay network* (a virtual network over a real on-chip network), which is used for searching, migrating, and replacing cache lines. We were forced to form column based overlays in electrical networks because of

practical constraints such as proximity and locality on the NoC. However, with optical networks where all the nodes are more or less equidistant from each other (within 2-3 cycles), we are free to create any kind of overlay that we wish (see Figure 4). We conducted experiments with traditional column based overlays with optical networks and observed that there is scope for improvement (see Appendix B.2). We thus propose more sophisticated optical overlay structures with a dual aim of improving performance and reducing dynamic cache access energy.

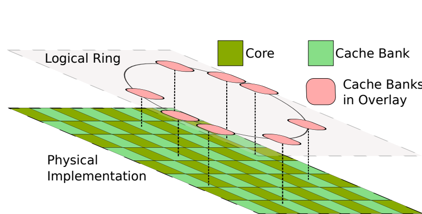

Fig. 4: Logical ring based overlay

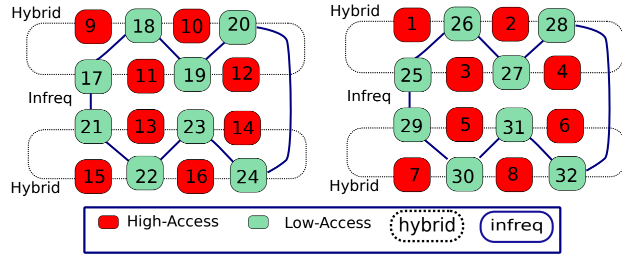

Fig. 5: Structure of the $hybrid$ and $infreq$ overlays

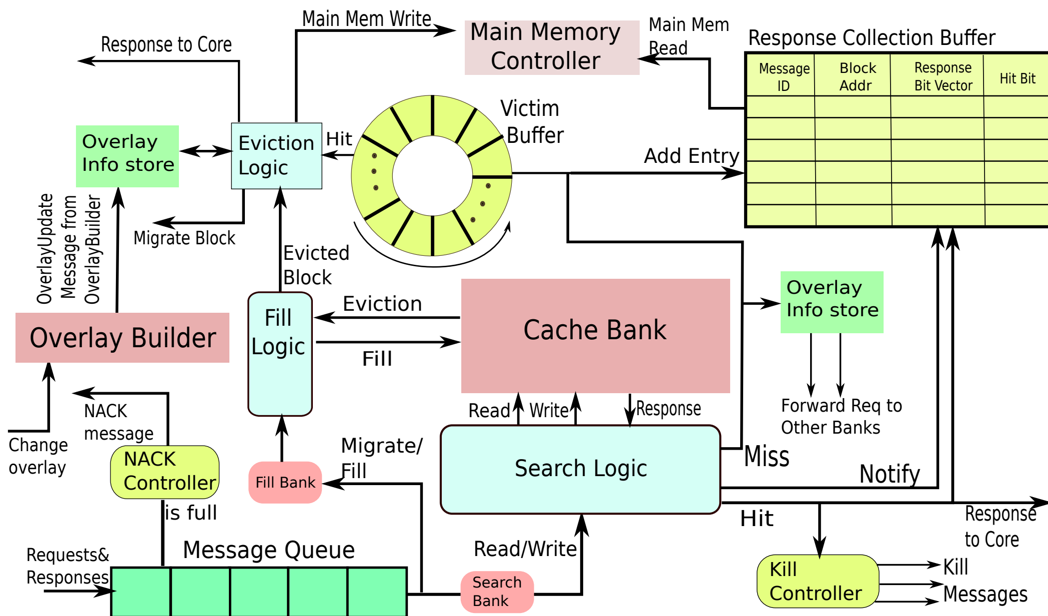

Fig. 6: Home bank controller

4.1. Details of the Overlay Network

Let us conceptually try to lay the basis for a new kind of overlay network that is based on the access frequencies of different cache banks. The heavily accessed banks tend to have lower hit rates because of conflict misses. Let us delve into the reason for varying degrees of contention in different banks.

We select the home bank based on the memory address. We tried using bits from different parts of the address to determine the home bank. We observed (see Appendix B.1) that if we use the least significant bits of the block address to determine the bank that contains it as proposed in [Changkyu et al. 2003], there is still a great degree of imbalance in accesses to different banks. Note that this method of selecting bits to identify the home bank is expected to produce the most homogeneous bank access profile. We observed in Appendix B.1 that the difference in the number of accesses can be as large as 1000X, and is typically 10-20X. The cause of the imbalance is the nature of the code. We observed this to be a trend across all the benchmarks that we studied. This imbalance translates to an imbalance in terms of accesses in banks and bank sets in NUCA caches as well. We additionally conducted studies on which bits in the memory address we should use for determining the id of the home bank. Empirically, we found the best solution is to use the least significant bits of the fragment of the memory address that is left after removing the least significant block and set index bits (for simulation results, refer to Appendix B).

Our main insight is to have more flexible bank sets (overlays) that take into account the nature of accesses by a program such that we can maximize the hit rate and homogenize bank accesses. With optical networks we can afford this flexibility. Our approach is as follows. For the first 100 million instructions of a parallel program (such as Splash or Parsec benchmarks), we use the S-NUCA policy, where each block is mapped to a unique bank called its *home bank* based on its memory address. Meanwhile, we collect the access counts of each bank, and at the end of this period, we divide them into two sets (of 16 banks each): *high-access* and *low-access*. Subsequently, we create the overlays based on the access counts obtained in the profiling phase. We propose three schemes to construct overlays, and they can either be static or dynamic. The standard approach is to first search in the home bank, and then if there is a miss, search for the block in the overlay that the home bank belongs to. Let us first discuss the home bank controller in Section 4.2 and subsequently move to discussing overlays.

4.2. Home Bank Controller

Each bank in the overlay has a home bank controller (HBC) as shown in Figure 6. Each *HBC* has its own Message Queue(*MQ*). All the requests are enqueued in it. Messages are enqueued and dequeued from this structure every 2 cycles (our pipelined cache bank accepts two new requests every 2 cycles). If there is an L1 cache miss, the request for the data block is sent to the HBC of the home bank optically, where it is enqueued in its MQ (takes 1 cycle). If no space is left in the MQ, then a *NACK* message is sent back to the requester. The requester retries after 2 cycles, and follows an exponential backoff scheme if it receives another *NACK* message. The requests in the MQ are processed in FIFO order. A processed request is then sent to the search logic of the HBC (takes 1 cycle). If there is a hit in the home bank, then a *Response* is sent to the core. Otherwise the request is forwarded to the home bank's successors in its overlay (1 more cycle). The details of the searching mechanisms are explained in Section 4.3. The *Eviction logic* along with the *Overlay info store* store the rules for eviction, and the structure of the overlay respectively. The *Overlay Builder* is used to create and reconfigure the overlay. Section 5 explains the procedure for dynamic overlay reconfiguration.

4.3. Simple Overlays – *TSI* and *Broadcast*

Let us first discuss some simple mechanisms that were originally proposed in the conference paper [Peter et al. 2015a].

4.3.1. Topology. Let us create two kinds of overlays: $hybrid$ and $infreq$ (infrequent). As shown in Figure 5, a $hybrid$ overlay consists of 4 high-access and 4 low-access banks

(defined in Section 4.1). In Figure 5, we number the banks in descending order of their access counts (bank 1 is the most frequently accessed, and bank 32 is the least frequently accessed). We have 4 such overlays. We have two more $infreq$ overlays, which consist of 8 low-access banks each. Every low-access bank is shared between a single $hybrid$ and $infreq$ overlay. The insight is to bring uniformity in the access counts of each bank. The hardware invokes a custom software/firmware routine to compute the overlays as shown in Figure 5. Each bank contains a list of all the other banks in its overlay, and also a list of its neighbors. The details of the overlay are saved in the structure – *Overlay Info Store*.

Two-Side Incremental(TSI): If there is a miss in the home bank, the home bank controller finds the overlay that the block is mapped to. The mapping logic is as follows: if the home bank is a high-access bank we select the $hybrid$ overlay that the home bank is a part of, otherwise we choose the $infreq$ overlay that the home bank is a part of. Then the home bank allocates an entry in the *RCB* (see Section 4.6.2) and sends requests to its left and right successors (two branches of the ring). The left and right successors forward the requests (in case of a miss) to their successor banks and so on. Here, we are searching simultaneously on both sides (branches) of the ring shaped overlay. Note that all the messages have the same ID (discussed later). If there is a hit in any of the successor banks, the bank stops forwarding the request, and sends a *Response* message to the requesting core and a set of *Kill* messages in the opposite branch of the ring. A *Kill* message accesses the MQ in the destination cache bank and dynamically invalidates any copies of the original message. This helps us in controlling contention in the rest of the cache banks.

We send the block to the requesting core, and a *Hit* message is sent to the *RCB* of the home bank to free the corresponding entry. If there is a miss, then the request is sent to the successor bank. This process is continued till the last bank is reached on both sides. If a miss occurs at the last bank, a *Miss* message is sent to the *RCB* of the home bank. If the *RCB* of the home bank receives two *Miss* messages, it declares a L2 cache miss and requests the data block from the lower level in the memory hierarchy. The access protocol is shown in Figure 7.

Broadcast: This is a naive search scheme, where the home bank broadcasts a message to the rest of the cache banks in its overlay. If any bank has a cache hit, it sends a *Hit* message to the home bank. The *Kill* messages works in the same manner as the TSI protocol. Note that the insight behind the TSI protocol was to reduce traffic and bank accesses, and the broadcast protocol is in reality a multicast operation because we are sending a message to only the banks in the overlay. Furthermore, to reduce energy we send messages in batches of 3. Finally, note that we rigorously simulated our protocols and did not find any race conditions or protocol errors for simulations with up to 10 billion instructions.

4.4. Optimal Broadcast Protocol(*OP_BCAST*)

This scheme combines the positive aspects of *TSI* and *Broadcast*, which have lesser contention and lower latency respectively. Additionally, it does not suffer from some of the problems that plague the *TSI* scheme. In the *TSI* protocol, evictions from the high-access bank immediately reach the low-access banks. This increases the contention at these banks and was found to negatively influence the queuing delay at the banks as well as the hit rate. The *Broadcast* protocol is plagued with a lot of unnecessary cache accesses.

To ameliorate such problems, we propose a different overlay structure as shown in Figure 8. In the *TSI* protocol an overlay is created by placing the high-access and low-access cache banks alternately. Here, we create two types of overlays namely *hybrid-o* and *infreq-o* as shown in Figure 8. There are six overlays, out of which four are of type

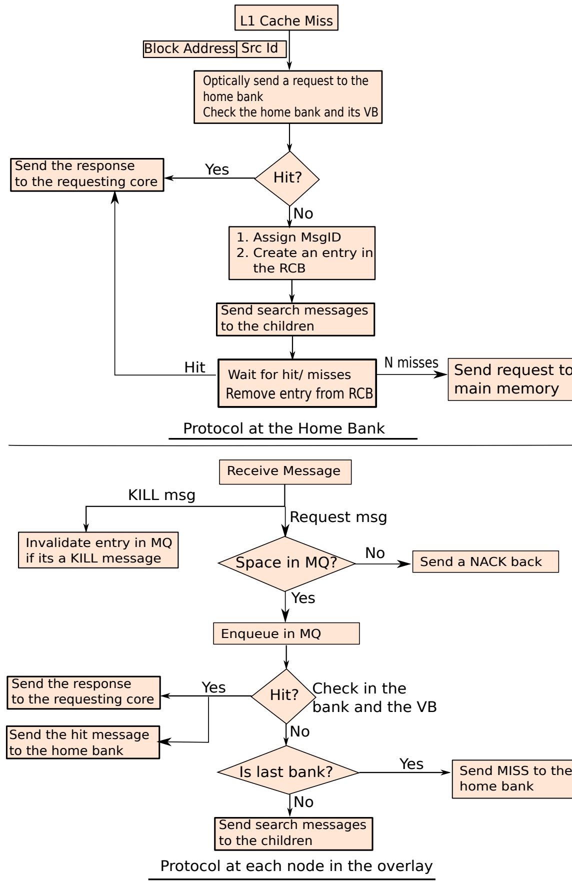

Fig. 7: Protocol

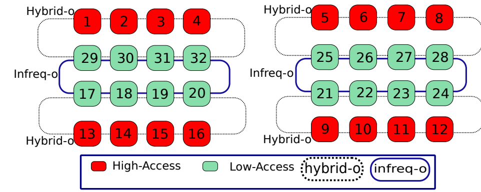

Fig. 8: Optimal Broadcast overlay

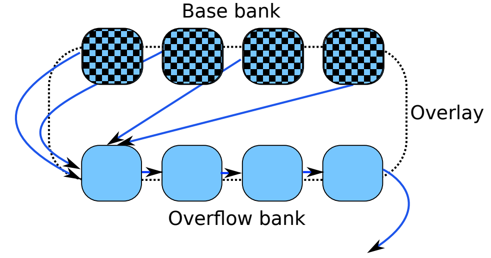

Fig. 9: Base and overflow sets (Eviction logic)

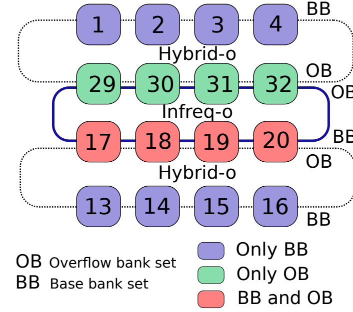

Fig. 10: BB and OB in Detail

hybrid-o and two are of type *infreq-o*. Instead of placing high-access and low-access banks alternately, we place them together. The *infreq-o* bank contains low-access banks only. We further classify the banks as follows. In each overlay, we divide the 8 cache banks into two subsets of 4 cache banks based on their access frequency. The top 4 frequently accessed banks are called *base banks* and the rest of the 4 banks are referred to as *overflow banks*.

Here we can make a few minor inferences. In the sorted order of cache banks based on their access frequency, the first 16 cache banks are present only in the base bank set (see Figure 8). The next 8 cache banks are present both in the base bank set of one overlay and in the overflow bank set in another overlay. The least accessed 8 cache banks are present only in overflow bank sets (see Figure 10). In other words, other than the least accessed 8 cache banks, all the other banks are there in one base bank set. We can also say that the cache banks that are not present in *infreq*-o overlays, will be present in the base bank set only.

4.4.1. Searching. Now we move on to the searching mechanism. See Figure 9 where we divide each overlay into two subsets (base and overflow). Now we see that when a request comes to a home bank, there are two cases: (1) the home bank is present in the base bank set, or 2) the home bank is present only in the overflow bank set. If we have

a choice, we always map a bank to an overlay in which it is in the base bank set. For the 8 banks that are only in the overflow bank sets, we choose the *infreq-o* overlay.

Case 1 : When a request comes to the home bank, which is a base bank, the address is searched in the home bank. If it is a hit, then the response is sent to the requesting core. Otherwise, we forward the request to banks in the overflow bank set. The *RCB*, *Hit*, *Miss* and *Kill* messages are used in the same way as the *TSI* protocol. One difference in the operation of the *RCB* is that the removal of the *RCB* entry is after receiving 4 *Miss* messages, i.e., from all the banks in the overflow bank set.

Case 2 : For the least accessed 8 cache banks, when a request comes, the address is searched within the home bank and if it is a miss, the request is forwarded to the lower level. We do not search in any overlay.

4.5. Eviction and Migration Policies

In the case of *TSI* and *Broadcast* when an eviction of a data block occurs from a bank, then instead of evicting it from the L2 cache, we migrate the block to a neighboring bank (clockwise along the ring). In our ring based overlay, each block is given $n - 1$ chances to remain in the L2 cache, where n is the number of banks in the overlay. When the block reaches the last bank, it is finally evicted from the L2 cache and written to the lower level. The eviction and migration procedures take care of keeping a cache line in the same overlay in which its home bank is present. This is important to prevent multiple copies of a cache line. When a block is accessed, we migrate it towards its home bank. Migration is in the opposite direction of eviction. A cache line can be migrated one step on every access.

In the case of *OP_BCAST*, the evicted cache line from a *base_bank* is moved to the first bank of the overflow bank set (see Figure 9). On further evictions, the cache line goes to next bank in the overflow bank set and finally an eviction from last overflow bank sends the cache line to the lower level, which in our case is main memory. Migration happens in the reverse direction. A cache line can migrate from the first bank in the overflow bank set to the home bank in the base bank set.

The associated circuitry is contained in the *Eviction logic* block in Figure 6. The circuitry accesses the *Overlay info store* to determine whether the bank from where the block is evicted is the last bank, and also to collect the id of the successor bank. Our main aim in designing the overlays was to use the under-utilized space in low-access banks for the evicted blocks of high-access banks.

There are some major differences between *TSI* and *OP_BCAST*. In *TSI*, any bank in an overlay uses all the other cache banks to possibly store the line. In other words, in *TSI* any cache line gets a maximum of 7 chances before it gets evicted to the lower level. One advantage of *OP_BCAST* over *TSI* is that in *OP_BCAST* the heavily accessed cache banks are not affected by the evictions from the less accessed cache banks. Another advantage is that all the migrations are in the same direction and all the evictions are in the opposite direction(see Figure 9). Compared to *Broadcast*, in *OP_BCAST* we multicast messages to a subset of banks (overflow bank set) and thus reduce contention. We send messages in batches of 2.

4.6. Additional Structures

4.6.1. Victim Buffer (VB). Note that if additional precautions are not taken, we can have race conditions in our search protocol. All such race conditions have to be avoided.

— **False Miss** - Let us assume that a block is originally present in bank A. The protocol first searches in bank B (home bank), and it does not find the block there. It subsequently, sends the request, req, to the successor of B, which is A. Exactly at that time, bank A decides to evict the block, and put it in bank B. After A has removed

the block, it gets the request, req. It also returns a miss. req travels to the end of the ring, and concludes that the block is not present in the L2 cache, which is false. A request is sent to main memory.

— **Multiple copies problem** - Because of a false miss, a request is sent to main memory, which places the block again at the home bank although the data block was present in some other cache bank.

The main issue is that we do not have a method of querying the contents of messages that are in flight. The first option is to add a mechanism to query the contents of messages in the MQ. However, this feature increases the number of ports in the MQ, and increases its complexity as well. Hence, we introduce a victim buffer (VB) at the node that evicts the block. It has $M + K$ entries, where M is the size of the MQ and K is the worst case network delay (3 cycles). Unless a *NACK* message is received, the evicted block will be written to the successor bank after at the most M cycles. We arrive at M cycles by considering the fact that we dequeue two messages from the MQ every 2 cycles. We are guaranteed to find the block in either bank A or B, or their victim buffers. We rigorously experimentally validated our protocol and did not find any race conditions.

4.6.2. Response-Collection Buffer (RCB). The *RCB* is used in each bank to collect the *Miss* messages coming from the cache banks. When there is a miss at the home bank, it creates an entry for the requested data block in its *RCB*. In the *TSI* protocol, when a miss has been identified in one branch of the ring overlay, the last bank of that branch notifies the home bank by sending a *Miss* message. If the *RCB* receives two such messages, then it implies that there were misses in both the branches. The entry is removed if the *RCB* receives either a *Hit* message or two *Miss* messages. In the case of *Broadcast* and *OP_BCAST* protocols, the condition to remove the *RCB* entry is either a *Hit* message or N *Miss* messages, where N is 8 for *Broadcast* and 4 for *OP_BCAST*. The size of the RCB is also determined experimentally. The maximum size it requires is 128 entries. Each entry contains the *Message ID*, block address, and the *Miss Response Bit Vector (MRBV)*. The *Message ID* is a 32 bit number, which is generated by the home bank. The 5 MSBs indicate the bank number, and the 27 LSBs are a per bank sequence number.

For the *Broadcast* and *OP_BCAST* protocols, the *MRBV* contains a bit vector whose size is equal to the number of banks in the overlay. In the case of *TSI*, it contains two bits (one for each branch of the ring). A bit in this vector is set to 1 whenever a *Miss message* is received from the bank corresponding to this bit. The *RCB* is also responsible for requesting a data block from main memory. When all the bits in the *MRBV* are set to 1, the *RCB* sends the request to main memory. All the communication between banks, buffers and cores happens though the optical network.

4.7. Message Format

We have two message formats: *Data Access* messages (Request, Response) and *Control* messages *(NACK, Kill, Hit, Miss)*, as shown in Figure 11. A 3-bit field is alloted in each message to specify the type. In a control message, the core-id denotes the requesting core that has requested the data block. In our system, each cache bank and core is assigned an ID. The source-id and the destination-id can be either a core-id or a bank-id.

Data messages for a response require 5 flits: 1 for the head flit and 4 for the payload. A cache line (as shown in Figure 11) or data block of 64 bytes is broken into 4 flits (16 bytes each) and then sent over the optical NoC. Each control message is 1 flit. The home bank controller treats data and control messages separately. A data message needs to be added to the message queue and kept there till a port becomes available.

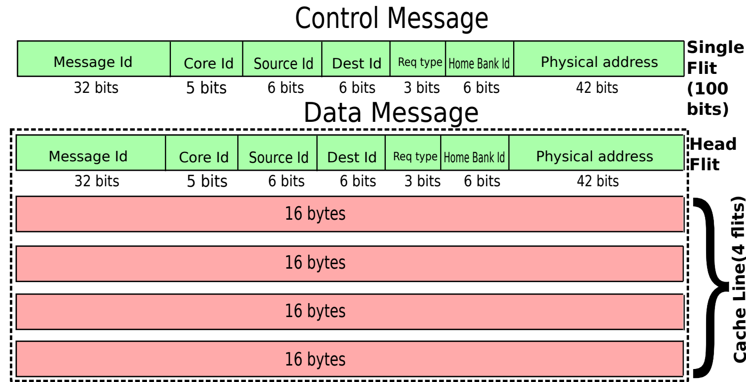

Fig. 11: Message format

A control message is not added to the message queue. Control messages are processed in the home bank controller separately.

5. DYNAMIC OVERLAY

In this section we examine the need for changing the optical overlay at runtime. The need is primarily driven by changes in the access frequency of blocks. For a lot of benchmarks, we have found the access frequencies to remain relatively stable across the execution of the benchmark. However, this is not the case for some benchmarks, and thus changing the overlay proves to be beneficial (see Section 6). Changing the overlay is however not that simple, given that it is necessary to transfer blocks between banks, and this can prove to be fairly expensive in terms of performance. Let us first show the structure of the overlay builder in Section 5.1 that builds new overlays by collecting access counts from banks. Note that before building a new overlay and dismantling the old overlay, it is necessary to temporarily suspend memory accesses in the system, migrate lines between banks if required, and then restart the system with the new overlay. We have shown the algorithm to build the overlay in Appendix A.

5.1. Overlay Builder

The hardware implementation of the overlay generator is shown in Figure 12. The overlay builder has a sorting mechanism, and dedicated logic to build the overlay. The sorting circuit sorts the bank accesses that are stored in the *Bank Access Vector* (BAV). The output of this sorted vector is then supplied to the *overlay-builder logic*. The output of the overlay builder populates the *OSV* (overlay structure vector) and *OBV* (overlay bit vector). Each entry in the *OSV* represents one overlay structure. We save six overlay structures in the *OSV*. Each entry in the *OBV* is 1 bit denoting whether the corresponding bank belongs to the *hybrid*/*hybrid-o* or *infreq*/*infreq-o* overlays.

5.2. Actions During Reconfiguration

Let us now discuss the complexities of changing the overlay at runtime. In the *OP_BCAST* protocol, for each overlay we define two types of banks: overflow banks and base banks. As discussed in Section 4.1, the overlays along with the type of each bank are determined by the access frequency of each bank. This depends upon the nature of the program. Let us consider the simplest possible situation where the access frequencies change yet the overlays remain the same and banks continue to be assigned to the same type of bank sets (base and overflow). In this case, nothing needs

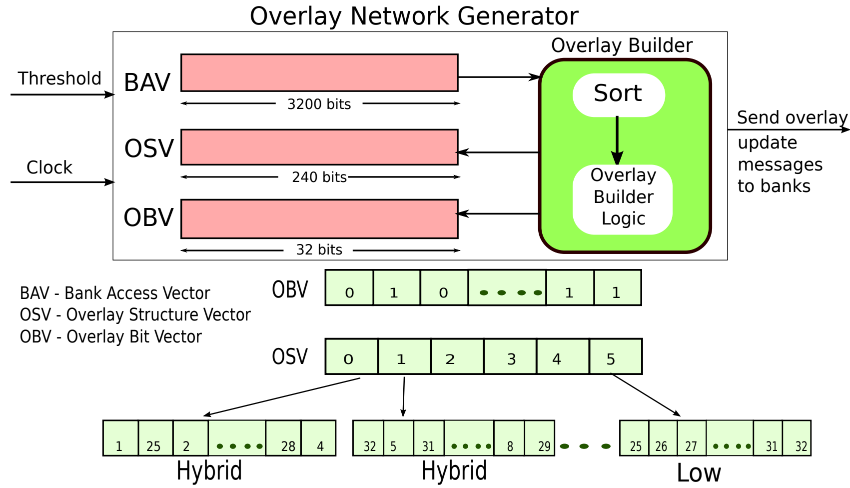

Fig. 12: Overlay Builder

to be done. Base banks will remain as base banks, and overflow banks will remain as overflow banks. Let us now consider more complex cases.

(1) A bank moves out of the base bank set→ It is possible that some of the cache lines belonging to this bank are present in overflow banks. These cache lines need to be removed from the overflow banks of this overlay.
(2) A bank moves out of the overflow bank set → We need to remove all the lines in this bank that originally belong to other banks (referred to as *foreign lines*).

5.3. Hardware Implementation

We add 3 bits to every cache line for identifying its home bank (2 bits called *position bits*), and 1 bit to indicate if it is foreign or not (*foreign bit*). Now for each overlay, we maintain two maps called *bank maps*: one for base banks and one for overflow banks. Each map contains a mapping between a bank id, and a unique integer between 0 and 3 (referred to as its *position*). We can maintain such maps in the overlay controller. Now, when we move a line from its home bank to an overflow bank or between overflow banks, we set its position bits to the position of the home bank in the bank map (number between [0-3]). Additionally, we also set its foreign bit to 1. This technique helps in saving space and leads to an efficient hardware implementation for locating lines originally belonging to some home bank. Now, let us assume that a bank moves out of an overlay's base or overflow bank set, and a new bank comes in its place. We need not change the entries in the corresponding bank map for other cache banks. We can assign the new bank the position of the old bank that was moved out.

Let us now consider the two cases mentioned in Section 5.2 when a bank moves out of the base bank set (case 1), and overflow bank set (case 2). For case 1, we have to check overflow banks for cache lines belonging to the particular bank that is moving out of the base bank set. We need to evict such lines. For case 2 we need to evict all the lines in that bank, whose foreign bit is set to 1.

5.3.1. Structure. The structure of the reconfiguration controller(*reconf_controller*) is shown in Figure 13. We have 4 components: *SRAM*, *trigger_circuit*, *logic_circuit* and *Control unit*. The *SRAM* is a memory unit, which stores 3 bits corresponding to each cache line. There are 2048 cache lines in a cache bank, therefore the total size of the

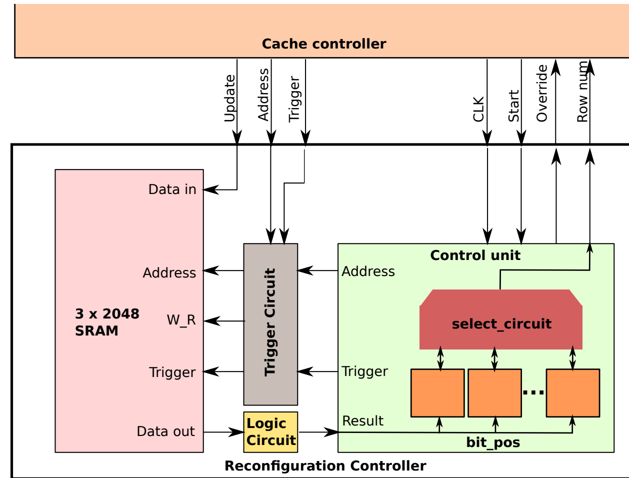

Fig. 13: Reconfiguration controller

memory in the *SRAM* is 2048×3 bits. The *SRAM* memory is implemented as a matrix of SRAM cells. Each row contains 24 bytes. The *trigger_circuit* generates enable signals for the *SRAM* to operate it. It can be initiated both from outside the controller and also from the *Control unit*. The *logic_circuit* processes the output from the *SRAM* and decides whether a cache line should be invalidated or not. The *Control unit* reads data from the *logic_circuit*, and processes it.

5.3.2. Operation. The *reconf_controller* supports two operations: write and reconfiguration. A write operation sets the value of the 3 bits in the SRAM (2 position bits and 1 foreign bit). However, the more complex operation is reconfiguration where we primarily have two kinds of sub-operations. We need to either find those lines whose foreign bit is set to 1, or we need to find those lines whose position bits have a particular value. These lines need to be later selectively removed from the bank.

Each bank in our system has 2048 lines (# entries in the SRAM). Instead of reading each line one by one, we optimize this circuit for speed. Instead of reading 3 bits for each line at a time, we read 24 bytes(one row in the SRAM) for 64 entries in one go. We have a *counter* in the Control unit that counts from 0 to 2047 in increments of 64 such that we can read and process all the elements in the SRAM. The value of this count and a *trigger* signal are passed to the *SRAM* through the *trigger_circuit*. These bits from the *SRAM* are then processed in parallel by the *logic_circuit*.

In the 24-byte sized read packet we have 3-bit entries for 64 lines. We process each entry with a small circuit that checks if the predicate holds. The predicate can check whether a line is foreign and/or its position bits have a certain value. It then yields a 1 bit output. At the end of this stage, we have 64 bits. Now, we cannot afford checking each bit of this 64 bit set one by one. We thus divide the process of finding the position of the bits that are set to 1 into two levels. In the first level, we use a set of circuits called *bit_pos* circuits. Each *bit_pos* circuit can process four bits at a time. We have 16 such circuits that work in parallel to process the entire 64 bit set. This circuit starts from the least significant bit and scans the 4-bit field towards the most significant bit. Whenever it finds a bit to be 1, it reports the relative position of the bit (a number from 0 to 63). After it receives an ACK signal from the second level circuit, it resets the bit, and moves to the next position in the 4-bit field in the next cycle. This means that if the 4-bit field given to *bit_pos* is 0011, we require at least two cycles, and if it is 1111, we

need at least 4 cycles. Note that this circuit is optimized to skip all the entries marked 0.

In the second level we have the *select_circuit*. It works as a multiplexer for the *bit_pos* outputs, and chooses one entry per cycle. It subsequently sends an ACK signal back to the corresponding *bit_pos* circuit such that it can process the next bit. Simultaneously, it adds the output of the *bit_pos* circuit with the value of the counter and obtains the position of the line to be evicted in the bank. It sends this position to the bank for eviction. Once all the *bit_pos* circuits report that all the remaining bits are 0s, we can start processing the round of 64 bits.

This circuit was synthesized with Cadence tools using the UMC 130 nm technology library and then scaled to 18nm using the rules in [Huang et al. 2011]. The designed circuit is optimized based on two design goals: (1) report one cache line to be evicted in each cycle, and (2) process maximum number of inputs in one round. The parameters for the 64 bit set, 4 input *bit_pos* circuit and 16 bit input *select_circuit* were selected such that the speed of the synthesized circuit is maximized. The circuit synthesis result is as follows. The circuit uses a 3.4 GHz clock. If we consider an extreme case (lower bound) in which all the input bits are zero(implies no lines to be invalidated), each round of processing 64 bits takes 4 cycles to complete (includes times for reading, and processing). To process 2048 lines(size of a cache bank), the circuit takes a minimum of $32 \times 4 = 128$ cycles. The area of the circuit is 1033 μm^2 in every cache bank, which is minimal. To ensure that the invalidated lines (that need to be written back to main memory), do not clog the memory controllers and DIMM channels, we add 4 dedicated victim cache banks (32 KB each) associated with each of the 4 memory controllers. At the time of overlay reconfiguration all the lines that are invalidated by the *reconf_controller* as well as modified are written to these banks. Subsequently, we resume operation, and simultaneously write modified data from these banks to main memory (at a lower priority). Requests to main memory first check for data in these victim banks before accessing main memory. The area overhead of the victim bank is roughly 1%.

6. EVALUATION

6.1. Experimental Setup

Table II shows the experimental setup of our simulations. We use the cycle accurate Tejas architecture simulator [Sarangi et al. 2015] for simulating our benchmarks (both power and performance). We use 32 out-of-order cores, 32 cache banks, and a Torus based electrical NoC for comparison with electrical NUCA protocols: S-NUCA, D-NUCA and R-NUCA. We used a suite of Parsec [Bienia et al. 2008] and Splash-2 [Woo et al. 1995] benchmarks for our experiments. In Table III we have shown the benchmarks that we used in the main paper. The main parameters specific to each benchmarks have been shown in the table. Note that we use a subset of benchmarks from each benchmark suite (due to ease of presentation). We have shown results for the entire suite of benchmarks (both Splash-2 and Parsec in Appendix C. Our subset of benchmarks has a mean performance, which is roughly equal to the mean performance of the entire suite. We thus consider them representative. Interested readers can always look at Appendix C. Lastly, note that we use the optical network for all the inter-tile traffic in the experiments with optical NoCs. Tejas has support for modeling both optical networks, and for computing the optical power we use standard approaches. We calculate the energy consumed (on a per cycle basis). We assume that each photodetector consumes $36\mu W$ per cycle. We then calculate backwards, factor in all the losses, take ring heating power into account (see Table I), and account for the

Parameter	Value	Parameter	Value
Cores	32	Technology	18 nm
Frequency	3.4 GHz		
Pipeline			
Retire Width	4	Integer RF (phy)	160
Issue Width	4	Float RF (phy)	144
ROB size	168	Branch Predictor	TAGE
IW size	54		
LSQ size	64	Bmispred penalty	8 cycles
iTLB	128 entry	dTLB	128 entry
Integer ALU	2 units	Int ALU latency	1 cycle
Integer Mul	1 unit	Int Mul latency	3 cycles
Integer Div	1 unit	Int Div latency	21 cycles
Float ALU	2 units	FP ALU latency	3 cycles
Float Mul	1 unit	FP Mul latency	5 cycles
Float Div	1 unit	FP Div latency	24 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, 4-banked, distributed MOESI, total 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	128 KB
Main Memory			
Latency	250 cycles	Mem. controllers	4
Electrical NoC			
Topology	2-D Torus	Routing Alg.	X-Y
Flit size	16 bytes	Hop-latency	1 cycle
Routing delay (w/wo bypassing)	2/3 cycles	# Virt. channels	4
		Buffers/port	8
Auxiliary structures (size in number of entries)			
RCB	128	VB	20
MQ	16	Victim bank	$4 \times 32KB$

Table II: Simulation parameters

Application	Input size
PARSEC (simlarge)	
blackscholes	64KB options
fluid	300,000 particles, 5 frames (fluidanimate)
streamcluster	16KB input points, 16KB points, 128 point dimensions
swaptions	64 swaptions, 20,000 simulations
Splash-2	
barnes	16KB particles
fmm	8KB particles
lu-cont	512 x 512 matrix (lu contiguous)
radiosity	batch, largeroom
water-nsq	512 molecules (water nsquared)
water-sp	512 molecules (water spatial)

Table III: Configuration of benchmarks

Application	% I-cache hit-rate	% D-cache hit-rate	% Directory hit-rate	L2 reqs per 1000 instructions
Parsec				
blackscholes	99.99	97.8	54.03	3.75
fluid	99.98	90.71	26.17	5.80
streamcluster	99.96	64.62	17.41	12.26
swaptions	99.95	82.5	10.89	33.05
Splash-2				
barnes	99.98	74.3	46.28	31.43
fmm	99.96	92.4	46.15	3.41
lu-cont	99.98	38.25	20.87	13.20
radiosity	99.98	96.65	91.86	0.44
water-nsq	99.96	92.12	64.6	4.21
water-sp	99.95	89.92	18.72	7.73

Table IV: Memory system statistics

energy consumed in O/E and E/O conversion circuits to get the total optical energy consumed per cycle.

6.2. Benchmark Characterization

Before discussing the results, let us characterize the behavior of our benchmarks (see Table IV). We mainly focus on four parameters: I-cache hit rate, D-cache hit rate, directory hit-rate, and L2 requests per 1000 instructions. In Table IV we can see that the I-cache hit rate is almost always close to 100%, and thus need not be considered in our further discussion.

The D-cache hit rate varies from 38.2% (*lu-cont*) to 97.8% (*blackscholes*). For 7 out of 10 benchmarks, it is between 80-97%. *Lu-cont* and *streamcluster* have low hit rates in the L1 D-cache because of low locality in their accesses. Next, let us focus on the directory hit rate. It shows an even wider variation 10% (*swaptions*) to 91.8% (*radiosity*). Much of this variation owes its reasons to the data sharing patterns in the code. The combined effect of the L1 and directory hit rates can be seen in the last column (L2 requests/ 1000 instructions). For benchmarks with low hit rates in the L1 and directory, we find a high number of accesses to the L2 cache: 12.26 (*streamcluster*), 33.0 (*swaptions*), and 31.4 (*barnes*). Benchmarks such as *blackscholes*, *fmm*, and *radiosity*, which find most of their data in the L1 level (locally or remotely via cache coherence) have a fairly small number of requests going to the L2 cache. The numbers are 3.75 for *blackscholes*, 3.41 for *fmm*, and 0.44 for *radiosity*.

6.3. Performance

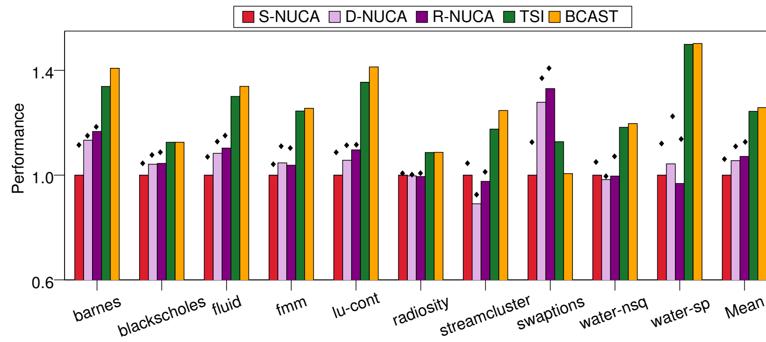

Fig. 14: Performance (electrical vs optical)

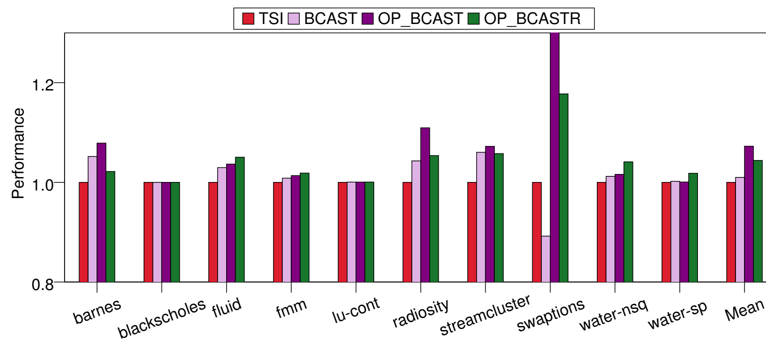

Fig. 15: Performance (optical only)

Let us now discuss the performance of our schemes. Here, **performance** is defined as a quantity that is inversely proportional to the simulated execution time. As a base-

line comparison, let us first compare three of the widely used schemes with electrical networks (S-NUCA, D-NUCA, and R-NUCA) with two of our basic schemes (*TSI* and *Broadcast*). The results are shown in Figure 14. The results are normalized with respect to S-NUCA. The solid bars for the electrical schemes indicate a scenario where we use an electrical NoC for both the L1 level (with cache coherence) and L2 level. The 4th and 5th bars represent the basic optical schemes: *TSI* and *Broadcast*. We find R-NUCA to be the best scheme with an electrical NoC (5% faster than S-NUCA). The performance of D-NUCA is roughly similar (4.5% faster than S-NUCA). In comparison *TSI* and *Broadcast* are 26% and 27% faster than S-NUCA respectively. If we analyze the trends per benchmark, we find low speedups with optical networks in *blackscholes* and *radiosity*. As discussed in Section 6.2 this is because the number of requests that reach the L2 cache are very low. In comparison *streamcluster*, *swaptions* and *barnes* had a relatively much higher fraction of requests going to the L2 cache. Other than *swaptions* the rest of the two benchmarks show good speedups. The number of L2 requests in *swaptions* is high compared to other benchmarks and most of these requests go to 1 or 2 banks, which creates high contention(see Appendix B.1).

The broad trends are not surprising given the fact that we are using a faster communication substrate, i.e., an optical network. Let us now consider a slightly different situation where the L1 level (all coherence messages) is implemented with an optical NoC. The L2 level is however implemented with a traditional electrical NoC (for S-NUCA, D-NUCA and R-NUCA). The results are shown with dots in Figure 14. We observe that the speedups over a baseline S-NUCA implementation with electrical networks is 6%, 10% and 11% for our three electrical schemes (S/D/R-NUCA) respectively when we use the optical NoC for the L2 level. We can quickly conclude that a third to a quarter of the additional speedup comes because of the usage of the optical NoC in the L1 level, and the rest comes due to the optical NoC in the L2 level. Again, the improvements at the L2 level can either be due to a faster communication network, or because our mapping schemes have improved the L2 hit rate and reduced bank contention. Before answering this question, let us compare our overlay based schemes in Figure 15.

Figure 15 shows the performance of all our overlay based schemes with optical networks (normalized to *TSI*). We term the *OP_BCAST* configuration with dynamic reconfiguration as *OP_BCASTR*. *Broadcast* is 1% faster. However, *OP_BCAST* and *OP_BCASTR* are 7% and 4% faster respectively. The speedups for *OP_BCAST* vary from 1-10% in 9/10 benchmarks, with *swaptions* being an outlier (speedup of 39%). *OP_BCASTR* outperforms *OP_BCAST* in 4/10 benchmarks, and performs roughly the same in two more benchmarks (*blackscholes* and *lu-cont*). However, on an average *OP_BCAST* is still better by 2.8%.

6.4. ED^2 Comparison

In Figure 16, we have shown the ED^2 comparison of different configurations. Here the energy of the entire system is considered. **Note that we simulate the energy usage of all our optical components in great detail. Our simulation methodology is in line with the methods adopted by highly cited references [Peter et al. 2015b; Zhou and Kodi 2013]**. We can see that the optical configurations have better results (lower ED^2) as compared to their electrical counterparts. This is because the optical configurations run faster. Among the optical protocols, *OP_BCAST* is the best, because of its low execution time. It also has a lesser number of broadcast requests as compared to *Broadcast*. Moreover, the number of cache bank accesses is also lower (explained in Section 6.5). The mean ED^2 shows a 68% improvement for *OP_BCAST* over SNUCA while *OP_BCASTR* has a 67% improvement. The improvement in the case of

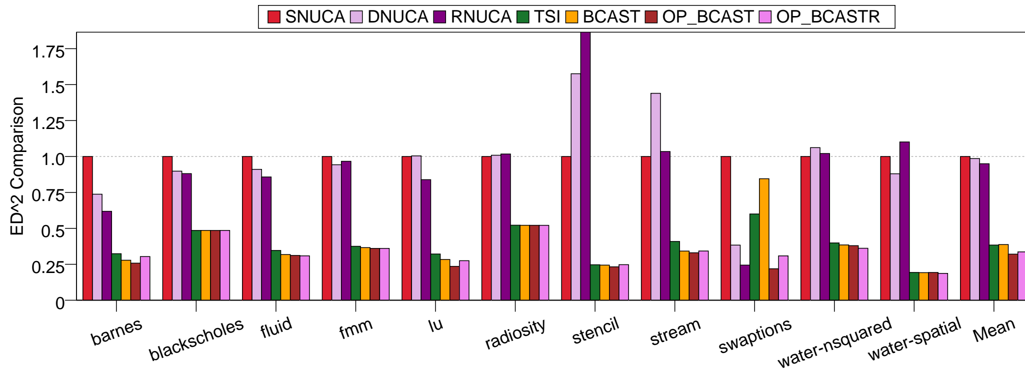

Fig. 16: ED^2 comparison

TSI and *Broadcast* is around 62%. In every benchmark, *OP_BCAST* outperforms *TSI* and *Broadcast*.

Note on energy consumption: An astute reader may argue that optical interconnects are not suitable for broadcast based traffic because of their high power requirements. This is why, we use restricted multicast messages to send requests to a limited number of stations (limited to 3). Multicasting a message to at the most 3 senders is reasonable in our opinion because this is being done only at the L2 level (or beyond) where requests are relatively infrequent as compared to the L1 level. For the L1 level we use a directory and avoid multicast traffic altogether (in line with related work). Moreover, we also leverage the R-SWMR paradigm, which saves energy by sending a message to just the required set of destinations [Pan et al. 2009](by converting a broadcast into a unicast/multicast). As a result we did not find the additional optical energy consumption to be prohibitive. The power requirements for the optical communication substrate were always limited to 10W on chip (including ring trimming power), which is in line with previous work [Peter et al. 2015b; Zhou and Kodi 2013]. Most of the time 6-10W is dissipated inside the chip, which is very reasonable considering the fact that electrical networks also dissipate a similar amount of power and the chip power budget(for a server) is typically 80-100W. We can thus conclude that our schemes are not restricted by energy consumption. Just in case there is a severe need of reducing the energy consumption further, we can always draw on techniques proposed in highly cited recent work [Demir and Hardavellas 2014; Le Beux et al. 2011; Pan et al. 2010] to reduce the energy consumption further.

6.5. Bank Access Patterns

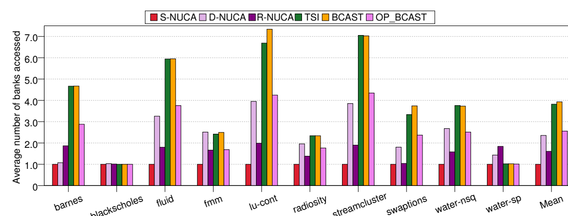

Fig. 17: Average number of banks accessed per L2 request

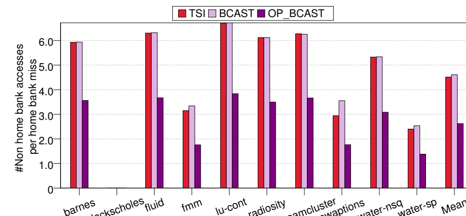

Fig. 18: #Non-home bank accesses per home bank miss

Our main aim of introducing the *TSI* protocol was to reduce the dynamic energy consumption of the L2 cache. Since the dynamic energy is proportional to the number of accesses, we focused on reducing the number of accesses in the *TSI* protocol. Figure 17 shows the number of accesses for all the configurations. We can see that the performance improvement in *Broadcast* is not more than 2-3% as compared to *TSI*, but the dynamic energy savings in *TSI* is approximately 6-8% across all the benchmarks (not significant). All our optical schemes perform poorly as compared to S-NUCA. On an average, *TSI*, *Broadcast* and *OP_BCAST* have 3.62, 3.75 and 2.45 times more accesses than the S-NUCA. R-NUCA and D-NUCA have 2.44 and 1.63 more access respectively (normalized to S-NUCA). One of our main objectives in creating the *OP_BCAST* protocol was to reduce the number of accesses to the banks (reduce both energy and contention). As we can see in Figure 17, we have been successful in this effort. *OP_BCAST* has 33% less access than *TSI* (on an average) and has roughly the same number of accesses as D-NUCA. The two noteworthy exceptions are *blackscholes* and *water-sp*. Let us now discuss the reason for the anomalous behavior of these benchmarks. *blackscholes* has very few accesses to the L2 cache (see Table IV), which is also visible in Figure 17.

Figure 18 shows the number of accesses to other banks, when we have a miss in the home bank. Here, we can also see that *blackscholes* and *water-sp* are exceptions. For *blackscholes* the number is near zero mainly because it hardly ever accesses the L2 cache. In *water-sp* most of the hits are found in nearby banks and because of our mechanism of sending *Kill* messages we can avoid a lot of bank accesses (also see Section 6.8). Additionally, we can also conclude from Figure 18 that accesses to non-home banks are reduced significantly in *OP_BCAST* (by 42%) as compared to other configurations. The reasons were already explained in Section 4.4.

We have not shown results for the *OP_BCASTR* configuration, because for most configurations the numbers were almost the same (or in some cases better by up to 25%).

6.6. L2 Hit Rates

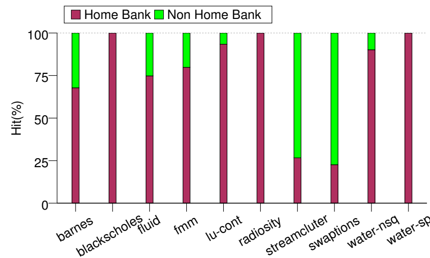

Fig. 19: Fraction of home bank hits

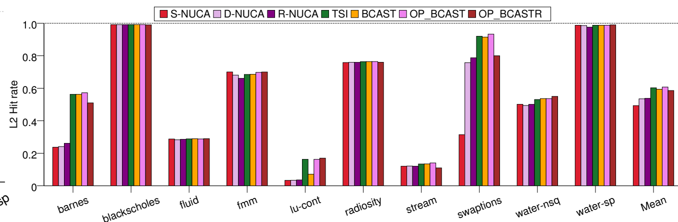

Fig. 20: L2 hit rate

Figure 19 shows the percentage of hits in the home bank for the *TSI* protocol (trends are representative). We can see that *blackscholes*, *lu-cont*, and *water-sp* are special. Almost all their accesses are home bank hits, and thus the number of messages that they send to other banks is on the lower side (confirmed in Figure 17). Hence, for them *TSI* and *Broadcast* perform similarly.

In comparison *barnes*, *fluid* and *streamcluster* have a larger fraction of messages that go to other banks. As we see in Figure 18, the number of banks accessed is 5-6 for these benchmarks. The average latency for *TSI* is thus roughly 3 bank accesses, whereas for *Broadcast* it is always 1 bank access. As a result *Broadcast* is much faster. This effect is however not present in *fmm* and *swaptions*, which also have a lot of

accesses that go to non-home banks. This is because the number of non-home bank accesses is 2-3X lower. As a result, the hit latency of *TSI* is similar to that of *Broadcast*.

Figure 20 shows the L2 hit rates. For 4/10 benchmarks (*barnes*, *lu*, *swaptions*, and *water-nsq*) the L2 hit rates with the optical overlay based configurations are measurably better. There is a marginal improvement in *streamcluster* and *fluid*. On an average, *OP_BCAST* has a 14% better L2 hit rate than D-NUCA. If we think about it, D-NUCA is conceptually the same as *OP_BCAST*. In both of the configurations, we have an overlay containing 8 banks, where a block can be present in any bank. The only difference is that in D-NUCA these banks need to be neighboring banks (along a column), and in *OP_BCAST* they can be anywhere on the chip. Now, the reason that the L2 hit rate is changing in 4/10 benchmarks is precisely because of this feature. Our bank-overlay mapping has been able to reduce the number of conflict misses. Since we can selectively target low-access cache banks and bring them into overlays which have a lot of misses, we can effectively reduce the miss rate for some benchmarks. We plot the L2 hit rates for *OP_BCASTR* as well. Other than *barnes* and *swaptions* the rest of the benchmarks have similar hit rates as compared to *OP_BCAST* (within 3-4%).

6.7. Migrations and Evictions

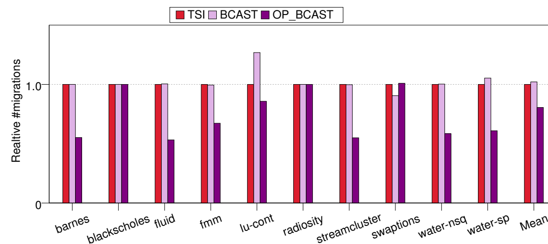

Fig. 21: Migrations (normalized to *TSI*)

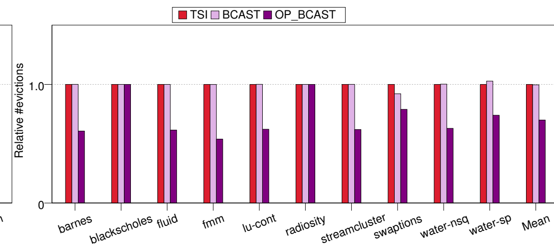

Fig. 22: Evictions (normalized to *TSI*)

Figure 21 shows the number of bank migrations relative to *TSI*. A *migration* is defined as the movement of a block towards the home bank when there is a hit. *OP_BCAST* is much better (#migrations lower by 27%) primarily because the migration scheme is much more constrained. This has a favorable impact on the traffic, and contention at banks. Similarly, Figure 22 shows the number of evictions, where an eviction is defined as a block moving out of a bank because some other block needs to be placed in the same set, and there are no free lines in the set. The block conceptually moves away from the home bank. We see that *OP_BCAST* has significantly lower evictions (lower by 23%). The reason is the same (reduced opportunities for migration/eviction).

6.8. Effective Kills

Let us define an *effective kill* as an event where an entry is removed from the message queue of some bank. This will happen if there is a hit in a non-home bank, and kill messages are sent. If a kill message finds a request for its corresponding block, it removes it from the message queue of that bank. In Figure 23 we show the number of effective kills per hit (in a non-home bank). We can see that the number of effective kills is almost zero in the case of *blackscholes* and *radiosity* due to very few L2 requests. In Figure 18, the number of non-home bank accesses in *water-sp* (for *OP_BCAST*) is low because the number of effective kills is high. In *streamcluster* the number of effective kills is low and therefore there are more accesses to banks other than the home bank. On an average *TSI*, *Broadcast* and *OP_BCAST* have 0.65, 2.73 and 1.64 effective kill

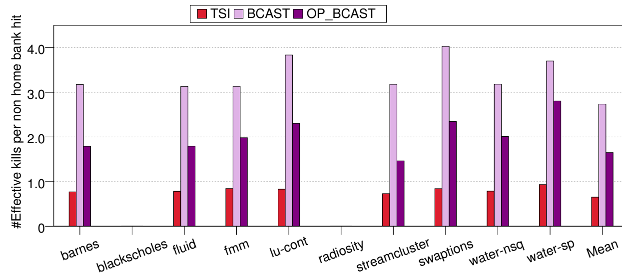

Fig. 23: #Effective kills per non-home bank hit

Benchmark	#cache lines invalidated per 10^6 insts.	Avg. #banks accessed
barnes	1450	3.95
blackscholes	0	1
fluid	718.2	4.77
fmm	292.3	1.78
lu-cont	383.4	4.87
radiosity	20	1.99
streamcluster	848.7	4.91
swaptions	808.4	2.75
water-nsq	19.5	2.86
water-sp	0.5	1.02
Mean	454.19	2.99

Table V: *OP_BCASTR* statistics

messages respectively per hit (non-home bank). Note that the maximum number of effective kills in the *TSI* protocol is limited to 1 (one kill along the opposite branch of the ring), and in *OP_BCAST* it is limited to 3 (rest of the overflow banks). From these results, we can conclude that kill messages play an important rule in reducing bank accesses and are an essential feature in our protocols.

6.9. Discussion on *OP_BCASTR*

Benchmarks	Number of hops		Home Bank Hits		Total Bank Accesses		Main memory requests	
	S	D	S	D	S	D	S	D
barnes	59.02	62.16	14.91	15.02	90.43	93.60	13.44	15.13
blackscholes	0	0	3.73	3.73	3.75	3.75	0.03	0.03
fluid	15.93	21.72	1.46	1.51	21.72	22.30	4.12	4.19
fmm	2.36	2.65	2.14	2.27	5.80	6.07	1.05	1.15
lu	42.86	49.32	1.51	1.64	56.07	61.16	11.06	11.16
radiosity	0.33	0.43	0.34	0.34	0.78	0.88	0.11	0.11
stencil	9.00	20.09	15.64	16.27	28.52	32.26	0.61	2.98
stream	41.64	51.46	0.94	0.98	54.09	75.10	10.72	11.27
swaptions	45.78	50.86	7.99	8.60	79.22	79.84	2.24	7.58
water-nsquared	6.47	6.78	2.19	2.20	10.77	12.15	2.01	2.04
water-spatial	0.09	0.10	7.68	7.78	7.82	7.87	0.1	0.1
Mean	20.32	24.14	5.32	5.49	32.63	35.91	4.13	5.07
S $\rightarrow$ *OP_BCAST*, D $\rightarrow$ *OP_BCASTR*								

Table VI: Statistics for *OP_BCAST* and *OP_BCASTR*

Table V shows the number of cache lines invalidated(due to a change in the overlay) per million instructions and the average number of banks accessed in *OP_BCASTR*. The L2 hit rate has been shown in Figure 20, and in Table VI, we compare the average number of home bank hits, total bank accesses, and main memory requests between *OP_BCAST* and *OP_BCASTR* for 1000 instructions. We observe that these numbers vary significantly for different benchmarks. Let us correlate these numbers with performance numbers shown in Figure 15. In *barnes*, *radiosity*, *streamcluster*, and *swaptions*, the static scheme, *OP_BCAST*, is significantly better. As we see from Tables V and VI, for *barnes* the reason is the higher requests to main memory, for *streamcluster* it is the higher number of bank and memory accesses, and the reason for *swaptions* are similar to those for *streamcluster*. The dynamic scheme does better in *fluid*, *fmm*, *water-nsq*, and *water-sp*, where we don't see a great increase in the parameters shown in Table VI. The benefit basically comes from the increased L2 hit rate, and higher number of home bank hits. In addition for *water-nsq* and *water-sp* the number of invalidations due to changes in the overlay are few. This also reduces the cost of the dynamic (*OP_BCASTR*) scheme.

7. CONCLUSION

In this paper, we proposed a novel scheme based on overlays for accessing the last level cache. We built on our two baseline schemes, *Broadcast* and *TSI*, and proposed a novel scheme *OP_BCAST* that combines the desirable properties of both of them. Additionally, we proposed a variant of *OP_BCAST* that supports dynamic overlays. We obtained a mean speedup of 34% over the static NUCA (S-NUCA) scheme using our approaches. The performance difference between the *Broadcast* and *TSI* schemes is minimal (2-3%). However, *OP_BCAST* performs 7% better than *TSI*, and reduces the number of accesses (proportional to the dynamic energy consumption of the cache) by roughly 33%. *OP_BCASTR* outperforms *OP_BCAST* in 5/10 benchmarks.

REFERENCES

Anuj Arora, Mayur Harne, Hameedah Sultan, Akriti Bagaria, and Smruti R Sarangi. 2015. FP-NUCA: A Fast NOC Layer for Implementing Large NUCA Caches. *Parallel and Distributed Systems, IEEE Transactions on* (2015).

C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*.

Nathan Binkert, Al Davis, Mikko Lipasti, Robert Schreiber, and Dana Vantrease. 2009. Nanophotonic Barriers. In *Workshop on Photonic Interconnects & Computer Architecture (in conjunction with MICRO 41)*.

K. Changkyu, D. Burger, and S.W. Keckler. 2003. Nonuniform cache architectures for wire-delay dominated on-chip caches. *Micro, IEEE* (2003).

Mark J Cianchetti, Joseph C Kerekes, and David H Albonesi. 2009. Phastlane: a rapid transit optical routing network. In *ACM SIGARCH Computer Architecture News*.

Yigit Demir and Nikos Hardavellas. 2014. EcoLaser: an adaptive laser control for energy-efficient on-chip photonic interconnects. In *ISLPED*.

N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. 2009. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*.

Mikhail Haurylau, Guoqing Chen, Hui Chen, Jidong Zhang, Nicholas A Nelson, David H Albonesi, Eby G Friedman, and Philippe M Fauchet. 2006. On-chip optical interconnect roadmap: challenges and critical directions. *Selected Topics in Quantum Electronics, IEEE Journal of* 12, 6 (2006), 1699–1705.

Wei Huang, Karthick Rajamani, Mircea R Stan, and Kevin Skadron. 2011. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro* 31, 4 (2011), 16–29.

C. Kim, D. Burger, and S. W. J. Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*.

Nevin Kirman, Meyrem Kirman, Rajeev K. Dokania, Jose F. Martinez, Alyssa B. Apsel, Matthew A. Watkins, and David H. Albonesi. 2006. Leveraging Optical Technology in Future Bus-based Chip Multiprocessors. In *MICRO*.

Nevin Kirman and José F. Martínez. 2010. A power-efficient All-optical On-chip Interconnect Using Wavelength-based Oblivious Routing. In *ASPLOS*.

S. Koohi and S. Hessabi. 2014. All-Optical Wavelength-Routed Architecture for a Power-Efficient Network on Chip. *Computers, IEEE Transactions on* 63, 3 (March 2014).

Somayyeh Koohi, Yawei Yin, Shaahin Hessabi, and SJ Yoo. 2014. Towards a scalable, low-power all-optical architecture for networks-on-chip. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 3s (2014), 101.

George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. 2010. ATAC: A 1000-Core Cache-Coherent Processor with On-chip Optical Network. In *PACT*.

Sébastien Le Beux, Jelena Trajkovic, Ian O'Connor, Gabriela Nicolescu, Guy Bois, and Pierre Paulin. 2011. Optical ring network-on-chip (ORNoC): Architecture and design methodology. In *DATE*.

Javier Merino, Valentin Puente, and Jose A Gregorio. 2010. ESP-NUCA: A low-cost adaptive non-uniform cache architecture. In *HPCA*.

R. Morris, E. Jolley, and A.K. Kodi. 2014. Extending the Performance and Energy-Efficiency of Shared Memory Multicores with Nanophotonic Technology. *Parallel and Distributed Systems, IEEE Transactions on* (2014).

Randy Morris and Avinash Karanth Kodi. 2010. Exploring the design of 64-and 256-core power efficient nanophotonic interconnect. *Selected Topics in Quantum Electronics, IEEE Journal of* 16, 5 (2010), 1386–1393.

Yan Pan, John Kim, and Gokhan Memik. 2010. Flexishare: Channel sharing for an energy-efficient nanophotonic crossbar. In *HPCA*.

Yan Pan, Prabhat Kumar, John Kim, Gokhan Memik, Yu Zhang, and Alok Choudhary. 2009. Firefly: Illuminating Future Network-on-Chip with Nanophotonics. In *ISCA*.

Eldhose Peter, Anuj Arora, Akriti Bagaria, and Smruti R. Sarangi. Online Appendices. http://www.cse.iitd.ac.in/~srsarangi/tr/opticalnuca_appendix.pdf

Eldhose Peter, Anuj Arora, Akriti Bagaria, and Smruti R Sarangi. 2015a. Optical Overlay NUCA: A High Speed Substrate for Shared L2 Caches. In *HiPC*.

Eldhose Peter, Arun Thomas, Anuj Dhawan, and Smruti R Sarangi. 2015b. ColdBus: A Near-Optimal Power Efficient Optical Bus. In *HiPC*.

Graham T. Reed. 2008. *Silicon Photonics: The State of the Art*. John Wiley & Sons.

S. R. Sarangi, Kalayappan Rajshekar, Kallurkar Prathmesh, Goel Seep, and Peter Eldhose. 2015. Tejas: A Java based Versatile Micro-architectural Simulator,. In *PATMOS*.

Dana Vantrease, Nathan Binkert, Robert Schreiber, and Mikko H Lipasti. 2009. Light speed arbitration and flow control for nanophotonic interconnects. In *MICRO*.

D. Vantrease, M.H. Lipasti, and N. Binkert. 2011. Atomic Coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *HPCA*.

Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P. Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G. Beausoleil, and Jung Ho Ahn. 2008. Corona: System Implications of Emerging Nanophotonic Technology. In *ISCA*.

S. Van Winkle, D. Ditomaso, M. Kennedy, and A. Kodi. 2016. Energy-efficient optical Network-on-Chip architecture for heterogeneous multicores. In *2016 IEEE Optical Interconnects Conference (OI)*.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*.

Li Zhou and A.K. Kodi. 2013. PROBE: Prediction-based optical bandwidth scaling for energy-efficient NoCs. In *NOCS*.