Performance Enhancement of Caches in TCMPs using Near Vicinity Prefetcher

Dipika Deb*, John Jose*, Maurizio Palesi†
* MARS Research Lab, Dept. of Computer Science and Engineering, Indian Institute of Technology Guwahati, India.
† Dept. of Electrical, Electronic and Computer Engineering, University of Catania, Catania, Italy.
{d.dipika, johnjose}@iitg.ac.in, maurizio.palesi@dieei.unict.it

***Abstract*—Prefetching aids in reducing the increasing processor-memory latency gap in a multiprocessor system. In a Tiled Chip MultiProcessor system (TCMP), the cache block reuse pattern of applications vary across the cores. Applications that have a working set larger than the L1 cache size cannot be accommodated in the limited L1 cache space. Such applications suffer from cache space constraints. Prefetching cache blocks of heavy application may cause thrashing by evicting useful cache lines from L1 cache thereby demanding frequent cache block replacements. On the other hand, light applications due to less cache block demands may under-utilize the available L1 cache space. This paper proposes Near Vicinity Prefetching (NVP) where some prefetch blocks are placed in the L1 caches of adjacent tiles running light applications. The prefetching framework works simultaneously with other L1 caches in different tiles to reduce the cache block access time. During cache hits in the neighboring L1 cache, the network transaction required to bring that block reduces from an average of *'n'* hops to just one. NVP helps in better cache space utilization and reduced network transactions at minimal hardware. Experimental analysis on a 64-core TCMP with SPEC CPU 2006 benchmark mixes shows that the average memory access time is reduced by 4.42% using the proposed NVP technique.**

***Index Terms*—Cache Space Management, Neighbor Tile, Miss penalty.**

I. Introduction

In the era of multicore architecture, computing speed of processing cores on a chip has increased but the access speed of the underlying memory is comparatively lower. Multiple levels of caches help in bridging the speed gap between processor and memory. The cache space demand by different applications running on each core in CMPs varies from application to application. A light application has a smaller cache footprint than a heavy application. This variation in cache block usage leads to cache store imbalance.

In recent TCMPs [1], each tile consists of an out-of-order superscalar processor (core), a private L1 cache and a slice of shared L2 cache (inclusive) as shown in figure 1. The tiles are arranged in a mesh topology and interconnected by an underlying Network on Chip (NoC) [2]. The core fetches its instructions and accesses its data from its private L1 cache and an L1 cache miss triggers an L2 cache access. In a TCMP, such L2 cache requests are forwarded as NoC packets destined to other tiles as per the SNUCA block mapping policies [3].

This research is supported in part by Department of Science and Technology (DST), Government of India vide project grant ECR/2016/000212.

Prefetching reduces the long memory access latencies by speculatively bringing cache blocks for future use [4]. A prefetch engine monitors run time activities like cache misses and cache access patterns. Upon identifying a pattern, the prefetch engine generates prefetch requests to fetch the desired cache block. In a TCMP since the L2 cache is distributed among all the tiles, the cache access pattern is also distributed across them. To speculate the cache access pattern, the prefetch engine is trained using L1 cache accesses which is local to each core. Upon a cache miss, the prefetch engine speculates the future accesses and prefetches the desired blocks. Prefetch techniques for shared banked distributed caches [5], [6], [7] also suffer from scalability and adaptivity issues. Most of the prefetching techniques are suitable for non-banked caches and they cannot be implemented on banked distributed cache structures like the one used in TCMPs [8].

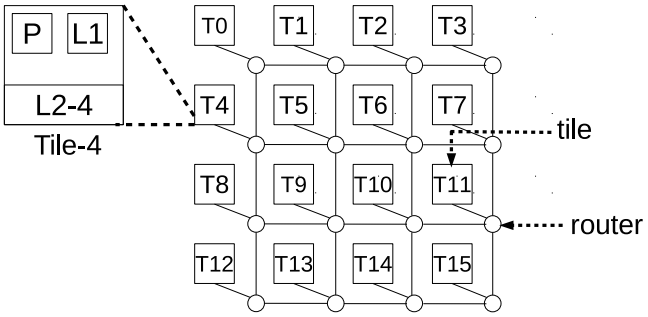

Fig. 1: An example of 8x8 Tiled Chip MultiProcessor (TCMP).

To reduce the memory stall cycles, a prefetcher can fetch more than k blocks where k (≥ 1) is known as the prefetching degree. In next line prefetching ($k = 1$), a cache miss for block B initiates a prefetch request for the next sequential block B+1, if it is not already present in the cache. In this paper, we call the existing technique as Source Core Prefetching (SCP) in order to avoid confusion between next line prefetcher and near vicinity prefetcher. The prefetch block is placed in the same tile where the miss occurs. When prefetching is enabled in a system, the cache contains demand as well as prefetch blocks. Demand blocks are fetched when the core requests for it while prefetch blocks are speculatively fetched ahead of its use by the core. In TCMP, the demand and prefetch blocks

2380-6923/19/$31.00 ©2019 IEEE
DOI 10.1109/VLSID.2019.00021

are brought as NoC packets from the respective L2 tile to the source tile at which the miss occurs.

Prefetching can be done both at L1 cache as well as L2 cache level. In TCMP, the latency gap between L1 and L2 cache is controlled by the NoC. Substantial performance gain can be achieved if the prefetcher can prefetch aggressively as in L2 prefetching along with faster access time similar to L1 prefetching. This paper proposes Near Vicinity Prefetching where the prefetch blocks are placed in adjacent tiles located within one hop distance of source tile. If the set index assigned to the block in adjacent tiles already contains another demand block, we store the block in a buffer called Prefetch Buffer Pool (PBP). PBP in each tile can accommodate a few (eight) cache blocks. Thus, our proposed technique increases the cache size virtually by storing the prefetch blocks in unused L1 cache space of neighboring tiles. This may also reduce the average memory access time during an L1 cache miss.

The rest of the paper is organized as follows. Section II presents the related work in prefetching. Section III provides the motivation behind this work. Section IV describes the proposed technique followed by experimental analysis in Section V. Section VI analyses hardware aspect of the proposed technique and the paper is finally concluded in Section VII.

II. Related Work

Existing works in prefetching focused on reducing the cache pollution and timeliness of prefetchers [9], [10]. In a distributed banked cache like TCMP, prefetching is still an active research area. Since this paper focuses on distributed banked cache architecture, therefore we summarize related work for such cache structures only.

Albericio et al. [5] proposed an adaptive controller, ABS to prefetch blocks in the last level cache. The controller controls the aggressiveness of prefetchers running at each L2 cache bank separately. Maria et al. [7] proposed a prefetch system that uses a server and client per core. Upon encountering a cache miss, the client feeds this information to a finite state machine that uses time series prediction to predict the next request generated by the core. The server prefetches cache block based on the prediction to increase the timeliness of the prefetcher. Andre et al. [6] proposed a balanced prefetch controller to prefetch blocks in the L1 cache by controlling the aggressiveness of prefetchers. The author uses a cache sniffer and a directory sniffer at each tile to reduce the miss penalties in a core.

III. Motivation

We consider SCP as the baseline prefetching technique and the term *target location* is used to indicate the set where a block is mapped. In SCP, a block is mapped to a cache using the conventional method. For a set associative cache, if a block maps to set *I*, then the adjacent block is mapped to set *I+1* with the same tag address. Therefore, in SCP the prefetch blocks are mapped in the adjacent set as that of the demand block. We model a 64 core TCMP on gem5 [11] with application workloads consisting of SPEC CPU 2006 benchmark mixes [12]. We have chosen benchmarks with a different range of cache footprints to create the workload.

Observation 1: Figure 2 shows the tile wise distribution of L1 cache misses in SCP. From the figure, it is clear that the L1 miss count varies across tiles (applications). This leads to variable rates of L2 cache block requests. Cache usage is dependent on the application running in the core. A low number of cache misses at certain cores (core 4, 7, 13, 16 ...) in the figure indicates that either the application running on the core is a light application or the L1 cache size is sufficient enough to handle the working set size of the application. A higher number of misses (core 1, 11, 42, 59) indicate that a heavy application is running on the core. For such applications, the working set size may be larger than the cache size and hence frequently required cache blocks cannot be fully accommodated in its L1 cache. As a result of this, the demand for cache space increases. This results in frequent cache block replacements and an increase in NoC traffic affecting the overall system performance.

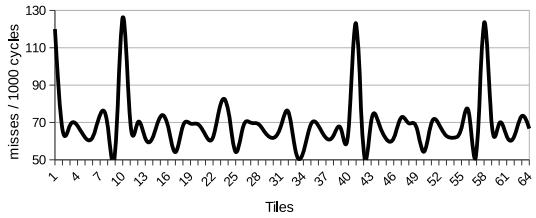

Fig. 2: L1 cache miss distribution per tile.

Observation 2: We model three variants of SCP based on the target location in order to study the impact of cache replacements caused due to prefetchers. As shown in Figure 3, these variants are categorized based on the replacement of existing blocks in the target location by an incoming prefetch block. Note that the figure indicates prefetch hit and not total cache hits. *Variant I:* Prefetch block can be stored if the target location is empty (invalid). *Variant II:* Prefetch block can only replace an existing prefetch block (if present) in its target location. *Variant III:* Prefetch blocks can replace any block already present in its target location. Most of the existing prefetcher uses *Variant III* during prefetching.

From the figure, we can observe that in *Variant I & II* the number of misses is less than that of *Variant III*. In *Variant III*, the prefetch hit is more but the total cache miss and network latency are compartively larger than the other two variants. This indicates that the replacement of existing blocks by a prefetch block may have evicted some useful cache blocks leading to cache pollution. Such cache blocks are written back to L2 and eventually requested again thereby increasing the number of cache misses. Since all cache misses and replies generate network packets, this may result in increasing the average packet latency of the network. Aforesaid behavior may have a detrimental effect in miss penalty on cores and may result in increasing the average memory access time (AMAT) in TCMP. Moreover, the prefetcher fetches such blocks repeatedly thereby increasing the prefetch hit count.

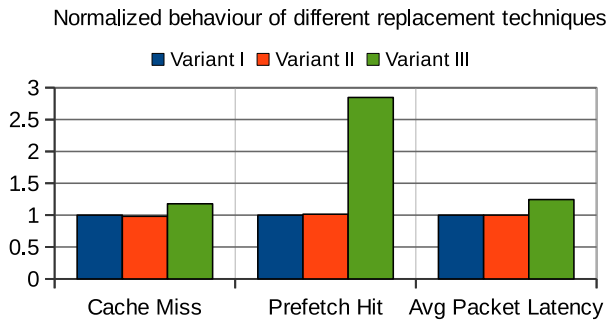

Fig. 3: Performance impact on three variants of cache replacements when prefetching is enabled in L1 cache.

This scenario is much more prevalent for heavy applications with a larger working set than its L1 cache space. From the figure, we can summarize that the placement of a newly arriving prefetch block must use either *Variant I* or *Variant II*. *Variant I* is not a practical approach because the cache will become full in due course of time. But in *Variant II* if no demand blocks are replaced, then after some time interval the cache will be full of demand blocks and hence no further prefetch blocks can be placed.

IV. Near Vicinity Prefetching

NVP is proposed to improve application performance by reducing cache space wastage in the system. On one hand, light applications may not use the L1 cache space efficiently, while on the other hand for a heavy application the L1 cache space may not be sufficient enough to hold the working set of the application. Prefetching in such applications may evict useful demand cache blocks from the caches. Hence in order to avoid such replacements, prefetch blocks can be placed either in the unused cache space of light applications or in the PBP of local tiles. To fetch such blocks faster on a miss, NVP chooses tiles that are at one-hop distance only. Therefore, the caching strategy that the prefecher uses for prefetch blocks is known as *Near Vicinity Prefetcher*.

In NVP, the target location of a prefetch block can be either in: a) L1 cache of local tile, b) L1 cache of neighboring tiles or c) PBP of local tile. The first choice of target location for a prefetch block is same as that of SCP. But if the target location is a heavily used set, then the core searches for a neighboring tile with a lightly used set as described in subsection IV-A. If no such neighbors are found, then the target location for the prefetch block is its PBP. Once the target location is fixed, the core stores the prefetch block in the desired location and uses a mapping vector in the local tile to identify the location. We use a simple mapping vector as discussed in subsection IV-B. Note that NVP is not a prefetch engine. A prefetch engine predicts the next block to be used in near future. NVP uses caching strategy that works on top of a prefetch engine to place the prefetch blocks efficiently. Hence, it can be used with other types of prefetchers like stream prefetcher [13], correlating prefetchers [14] etc.

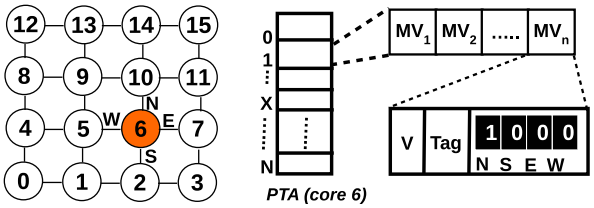

Fig. 4: Mapping vector in L1 cache for neighbor placement. Here the bit vector represents the direction of target location.

A. Neighbor Selection Strategy

A simple neighbor placement policy is employed to identify a target location for the prefetch blocks. The neighbor selection strategy uses a probing method where the source core probes the L1 caches of neighboring tiles in search of a lightly used set[1]. If one such target location is found, the neighbor replies back with a positive response and books the location by setting the valid bit in the targeted cache way. It also sets the direction flag of the neighbor whose block will be placed. This is done in order to avoid inconsistency in the system when another neighbor tries to use the same location to store its own block. Thus, a neighbor can book a cache location on a first come basis. Upon receiving a positive response, the requesting core places its block in the booked location at the neighboring L1 cache. If no such suitable location is found then the block is stored in its PBP. During neighbor probing, prefetch block with set index 'S_i' is placed on the same set index in the L1 caches of its neighbors. This is done in order to avoid the extra bits needed to store the set number in the mapping vector.

B. Mapping Vector in NVP

Each tile uses an additional storage called as Prefetch Tag Array (PTA) to store the target location of a prefetch block. PTA size is equivalent to the number of L1 cache sets and it is indexed with the same set index as that of the L1 cache. PTA can store at most four prefetch blocks ($n = 4$) in the neighbors for per PTA index. It uses a mapping vector (MV_i) for each prefetch block that is stored in the neighbors. Figure 4 shows a mapping vector which consists of a valid bit (V), tag bits and four direction flags (N, S, E, W). In a mesh topology, a tile is surrounded by four adjacent tiles where each bit in the direction flag corresponds to one of the four neighbors.

On prefetching a block, if the set index S_i in the L1 cache is lightly used then the block is stored in its local L1 cache else the L1 caches of neighboring tiles are probed as explained in the neighbor selection strategy. On discovering such a neighbor, a mapping vector is created in the PTA at the same set index as S_i. In the mapping vector, the valid bit is set and tag address of the block along with the direction of the neighbor is also stored. During block identification for an L1 cache hit, the PTA is also searched in parallel. If one such entry is found in PTA, the mapping vector forwards the request to the corresponding neighbor in order to fetch the

[1]The cache sets are classified as light or heavy by counting the number of accesses in each cache sets within an interval of 1024 cycles.

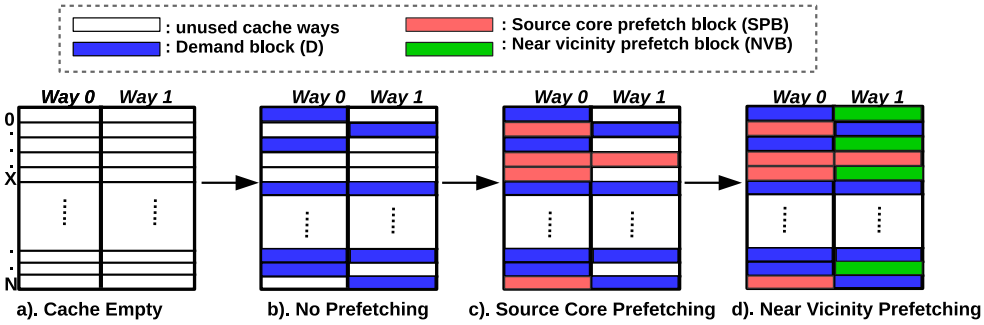

Fig. 5: Cache content when no prefetching, source core prefetching and near vicinity prefetching is enabled. a) cache is empty initially. b) cache has *demand* blocks only when no prefetching is enabled. c) cache has *demand* and *source core prefetch block* (SPB) when SCP is enabled and d) cache has a combination of *demand, source core prefetch and neighbor* blocks when NVP is enabled.

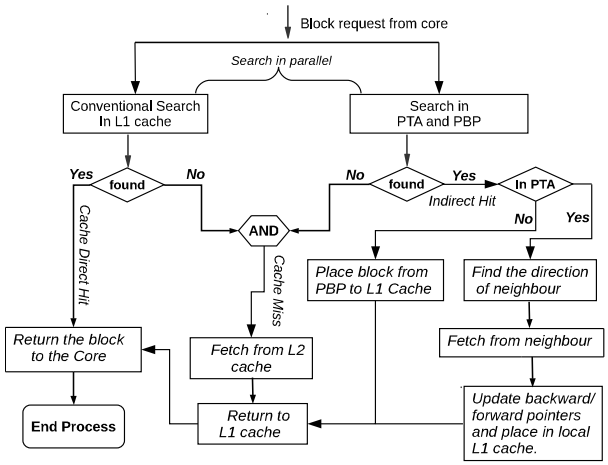

Fig. 6: Flow diagram for cache block searching in NVP.

block and store in its local L1 cache. Therefore, the mapping vector is also known as the *forward pointer*.

In NVP, the L1 cache may contain a combination of demand, prefetch, and neighbor's block. In order to indicate if the block stored in its set belongs to itself or to a neighbor, an additional four flags are used per block along with the conventional block information as in SCP. Each flag marks the four directions of the neighbors. If any of the flags is set, it indicates the neighbor whose block is stored. Therefore, the flags are called as the *backward pointer*. If none of the flags are set, then the block belongs to itself. Hence, for a particular block in the cache at most one flag will be set indicating the owner of the prefetch block. Figure 5(d) shows different types of blocks present in an L1 cache when NVP is employed.

C. Block searching in NVP

Figure 6 illustrates the flowchart for block searching in NVP. Whenever a core requests for a word, the block is searched simultaneously in the local L1 cache, PTA and PBP. If the block is found in the local L1 cache, it is called as *direct hit* otherwise in the other two cases it is called an *indirect hit*. On a direct hit, the requested word is served from the block to the core immediately. During an indirect hit, the block is brought from the target location to local L1 cache and thereafter the block is considered as a demand block. Accordingly, the backward pointer and the PTA entry in the neighbor and local cache are updated, respectively.

TABLE I: Details of SPEC CPU 2006 workload constituents; $X(Y)^n$: Benchmark X runs in Y cores with repetition of n times.

Benchmarks used:	specrand, sjeng, calculix, namd, bzip2, gcc, sphinx, leslie3d, hmmer, lbm, mcf
B1 (Mix1)	$P(16), Q(16), R(16), S(16)$
B2 (Mix2)	$(P(4), Q(4), R(4), S(4))^2$
B3 (Mix3)	$(P(2), Q(2), R(2), S(2))^8$
B4 (Mix4)	$(P(4), Q(4))^4, (R(4), S(4))^4$
B5 (Mix5)	$(P, Q, R, S, S, R, Q, P)^8$

D. Block replacement in NVP

Whenever the local replacement policy in L1 cache chooses a block as a victim, the cache controller checks the direction flags. If any of the flags is set, it indicates that the block does not belong to the tile. Hence such blocks cannot be replaced immediately. The tile then sends an invalid request packet to the owner of the block (following the direction flags). The owner upon receiving the invalid request invalidates the forward pointer in its local L1 cache. It then acknowledges back to the neighboring core for facilitating the eviction of that block for a new incoming block. Meanwhile, the tile stores the replaced block in a temporary storage and felicitates storing of the new incoming packet. Upon receiving the acknowledgment, the block from the temporary storage is deleted. Since the system is lossless therefore the invalid request will eventually reach the owner. During the invalidation process when the block resides in the temporary storage, the state of the block is transient and any request for this block must have to wait until completion of the replacement process.

V. Experimental Analysis

To implement NVP we use gem5 [11], a cycle accurate simulator that models the out-of-order superscalar cores and cache hierarchy. Booksim 2.0 [15] is integrated with gem5 to model NoC. We use Cacti 6.0 [16] for area and power analysis. The performance of NVP is evaluated with TCMP workloads

TABLE II: gem5 and Booksim configurations.

Processor	64, x86 cores with out-of-order execution
L1 cache/core	64KB, 2-way associative, 32B block
L2 cache/core	1MB, 16-way associative, 64B block
Physical memory	4GB
Prefetcher	Simple next-line prefetcher
NoC Topology	8×8 2D Mesh with XY-routing.
Virtual Channels	4 per input port
Packet size	1-flit request, 5-flit reply

consisting of SPEC CPU 2006 benchmark mixes. We create five workloads (B1 to B5) using the benchmarks given in Table I. The workloads consist of a combination of SPEC CPU 2006 benchmarks with varying MPKI values. In our experiments, each of the workloads consist of sixteen instances of four different benchmarks. For example, B1 consists of four benchmarks: P, Q, R, and S, each with 16 instances. P, Q, R, and S can be any benchmark given in Table I. The representation of the mix as P(16), Q(16), R(16), S(16) indicates the spatial mapping of the benchmarks on the 64-core TCMP. P runs on first 16 cores followed by Q in the next 16 cores and so on.

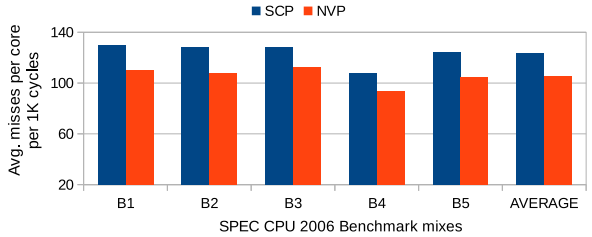

Fig. 7: L1 cache miss comparison.

Every L1 cache miss is analyzed by the respective cache controller for finding the L2 cache bank using the SNUCA mapping policy. Accordingly, the cache miss request packets are generated in the NoC. The demanded cache block and the prefetch block travel as reply packets in the NoC. All micro-architectural features of NVP design is implemented in gem5 and Booksim simulators with the configuration parameters mentioned in Table II.

A. Analysis of L1 cache misses

Figure 7 shows the number of misses per 1000 cycle in SCP and NVP. From the figure, it can be seen that the number of misses for applications running in the system reduces by 14.5% in NVP when compared to SCP. We observe that NVP efficiently utilizes the unused space of neighboring L1 caches. L1 cache hit rate is very crucial for applications running on the core. If the L1 cache capacity is large enough to accommodate the working set of the application, then we experience higher hit rates. In the conventional SCP design, L1 cache capacity cannot be dynamically changed due to its rigid cache organization. NVP uses neighbor core placement strategy to virtually increase the cache capacity of heavy applications. Hence, NVP results in fewer cache misses as compared to SCP.

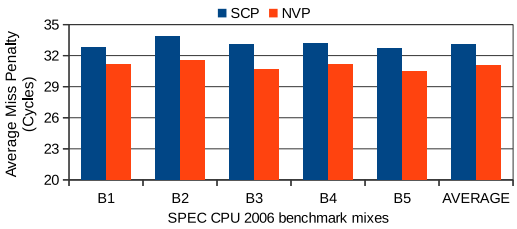

Fig. 8: Cache miss penalty comparison.

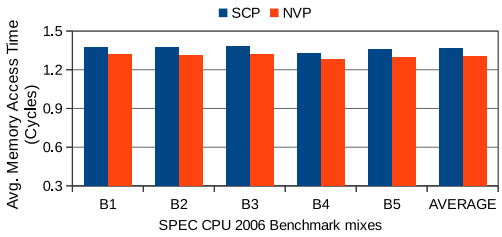

Fig. 9: Average memory access time comparison.

B. Analysis of L1 cache miss penalty

Figure 8 shows the average miss penalty in SCP and NVP. From the figure we can see that NVP reduces the miss penalty by an average of 7% w.r.t SCP. L1 caches of different applications experience non-uniform load. Certain applications use most of the available cache space due to its larger memory footprint. If we enable prefetching (SCP) in such caches, it results in frequent swap-in and swap-out of cache blocks leading to even more cache block replacements. Since every cache block replacement in TCMP involves NoC packets, it may increase network traffic and congestion.

NVP reduces the number of cache block replacements by carefully placing prefetch blocks in either L1 caches of neighboring tiles or in the PBP of the source core. This in turn reduces network traffic due to cache block evictions. This helps in reducing the cache miss penalties as shown in the figure.

C. Analysis of average memory access time (AMAT)

The average memory access time of L1 cache is dependent on the hit time, miss rate, and miss penalty of the cache. In NVP, the average hit time is slightly increased due to the extra cycles required for indirect hits. We have observed that NVP reduces the miss rate and miss penalty. The reduction in miss rate and miss penalty over-dominates the increase in hit time leading to lower AMAT. Figure 9 shows that NVP reduces average memory access time by around 4.42% as compared to SCP.

D. Impact of NVP in L1 caches of neighboring tiles.

As expected, NVP adaptively increases the L1 cache capacity per core at run-time. Figure 10 shows the distribution of misses per core-wise (L1 cache) in a 64-core TCMP (average of workloads B1 to B5). From the figure, we observe that the L1 cache of some application experiences less cache misses in NVP than in SCP without affecting the misses in L1 caches

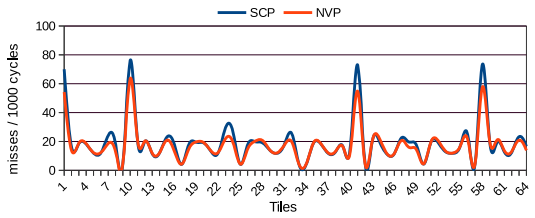

Fig. 10: L1 cache miss distribution per tile.

TABLE III: Additional hardware storage for NVP design.

Cache size per tile:			**1024 KB(L2) + 64 KB(L1) = 1088 KB**
Component	Bits/entry	number of entries	Bytes per tile
Backward Pointer	4	1024 * 2 (ways)	1024
PTA	22	1024 *4	11264
PBP	274	8	274
Additional storage in NVP per tile:			**≈13 KB**
Hardware overhead in NVP per tile :			**1.012**

of neighboring tiles. We find that, this runtime change of L1 cache capacity is not affecting the cache performance of tiles whose cache sets are temporarily occupied by neighboring heavy applications.

TABLE IV: Area and power consumption comparison.

Component	L1 cache	L2 cache	PTA	PBP
Area(mm^2)	0.1374	1.92	0.0074	0.00027
Power (nJ)	0.0304	0.2872	0.0017	0.00006
Total Cache Area (in SCP):				**2.0574** mm^2
Total Cache Area (in NVP):				**2.066** mm^2
Total Cache Power (in SCP):				**0.3176** nJ
Total Cache Power (in NVP):				**0.3194** nJ
Increase in area per tile (in %):				**0.42%**
Increase in power per tile (in %):				**0.57%**

VI. Hardware Design Analysis

A. Hardware overhead

The additional hardware involved in NVP design are an enhanced tag array that contains the backward pointer, the PTA that contains the forward pointer, and the PBP. Apart from the normal tag bits (17-bits as per Table II) and control bits we need an additional 4-bits to implement the backward pointer. In our design, we store a maximum of four prefetch blocks per set in the neighbors. Therefore for each set, a maximum of four mapping vectors are required. We need 22-bits (1(valid) + 17(tag) + 4(direction)) for each mapping vector in the PTA. The PBP per tile can accommodate eight cache blocks. Each PBP entry consists of a $32B$ cache block and 18-bits (17-bits tag and 1-bit valid) of control information. Table III shows the estimated hardware cost required to implement NVP in a 64-core TCMP. Therefore, the additional hardware storage in NVP increased by 1.012 times per tile as compared to SCP. The L1 cache has an extra *read-write port* for performing the additional task of neighbor placement.

B. Area and power overheads

The extra hardware used in NVP is PTA and PBP per tile. Table IV analyses the area and power consumption in NVP. In SCP, the total area and power consumption per tile is around $2.0574\ mm^2$ and $0.3176\ nJ$, respectively. Similarly in NVP, the total area and power consumption per tile is around $2.066\ mm^2$ and $0.3194\ nJ$, respectively. Comparing NVP with SCP there is an overall increase in area and power consumption per tile by 0.42% and 0.57%, respectively.

VII. Conclusion

Miss penalty of L1 cache blocks in banked distributed caches is not only dependent upon the L2 access time but is also dependent upon the underlying network latency. Heavy applications with large working set cannot fit in the available cache space. This results in swapping out of important cache blocks. Therefore, the cache block replacements occurring in such application increases the network traffic thereby increasing the average memory access time. Near Vicinity Prefetching helps in mitigating these problems by placing the prefetch blocks in the adjacent tiles. Hence, cores running heavy applications can use the less used cache space of adjacent L1 caches that are running light applications. Experimentally it is found that, NVP reduces the cache misses by around 14.5% as compared to the state-of-the-art next line prefetching. Thus, NVP improves the system performance by reducing the average memory access time for cache blocks at a negligible hardware overhead.

References

[1] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, *Multi-Core Cache Hierarchies*. Morgan and Claypool Publishers, 2011.

[2] W. J. Dally and B. Towles, "Route Packets, not Wires: On-Chip Interconnection Networks," in *DAC*, 2001, pp. 684–689.

[3] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," *SIGARCH Computer Architecture News*, pp. 211–222, 2002.

[4] S. Mittal, "A Survey of Recent Prefetching Techniques for Processor Caches," *ACM Computing Surveys*, vol. 49, no. 2, pp. 35:1–35:35, 2016.

[5] Albericio Jorge et. al, "ABS:A Low-cost Adaptive Controller for Prefetching in a Banked Shared Last-level Cache," *ACM TACO*, vol. 8, no. 4, pp. 19:1–19:20, 2012.

[6] A.Aziz et al., "Balanced Prefetching Aggressiveness Controller for NoC-based Multiprocessor," in *SBCCI*, 2014, pp. 1–7.

[7] M. Cireno, A. Aziz, and E. Barros, "Temporized Data Prefetching Algorithm for NoC-based Multiprocessor Systems," in *ASAP*, 2016, pp. 235–236.

[8] Ebrahimi et. al, "Coordinated Control of Multiple Prefetchers in Multi-core Systems," in *MICRO*, 2009, pp. 316–326.

[9] P. Yedlapalli et. al, "Meeting Midway: Improving CMP Performance with Memory-side Prefetching," in *PACT*, 2013, pp. 289–298.

[10] S. Srinath et. al, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007, pp. 63–74.

[11] Binkert et. al, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[12] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, pp. 1–17.

[13] C. Zhang and S. A. McKee, "Hardware-only Stream Prefetching and Dynamic Access Ordering," in *SC*, 2000, pp. 167–175.

[14] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block Prediction; Dead-block Correlating Prefetchers," in *ISCA*, 2001, pp. 144–154.

[15] Jiang et.al, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *ISPASS*, 2013, pp. 86–96.

[16] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Cacti 6.0: A Tool to Model Large Caches," 2009.