Locality-aware data replication in the last-level cache for large scale multicores

Farrukh Hijaz\textsuperscript{1} \cdot Qingchuan Shi\textsuperscript{1} \cdot George Kurian\textsuperscript{2,3} \cdot Srinivas Devadas\textsuperscript{2} \cdot Omer Khan\textsuperscript{1}

Abstract  Next generation large single-chip multicores will process massive data with varying degree of locality. Harnessing on-chip data locality to optimize the utilization of on-chip cache and network resources is of fundamental importance. We propose a locality-aware selective data replication protocol for the last-level cache (LLC). The goal is to lower memory access latency and energy by only replicating cache lines with high reuse in the LLC slice of the requesting core, while simultaneously keep the off-chip miss rate low. The approach relies on low-overhead yet highly accurate in-hardware runtime cache line level classifier that only allows replication of cache lines with high reuse. Furthermore, a classifier captures the LLC pressure at the existing replica locations and adapts its replication decision accordingly. On a set of parallel benchmarks, the proposed protocol reduces overall energy by 14.7, 10.7, 10.5, and 16.7 \% and completion time by 2.5, 6.5, 4.5, and 9.5 \% when compared to the previously proposed Victim Replication, Adaptive Selective Replication, Reactive-NUCA,
and Static-NUCA LLC management schemes. An efficient classifier implementation is evaluated with an overhead of 5.44 KB, which translates to only 1.58 % on top of the Static-NUCA baseline’s cache related per-core storage.

**Keywords**  Multicore · Cache hierarchy · Data management · Energy efficiency

1 Introduction

Computing trends indicate integration of a large number of cores on a single chip. Since the diameter of on-chip networks increases with core count, the cost of moving data on-chip is becoming expensive. Furthermore, emerging technologies such as 3D die stacking [1] make the “distance” to access data even more non-uniform and costly. On the technology front, on-chip wires are not scaling at the same rate as transistors. While compute energy is projected to scale down by $6 \times$ from 45 to 7 nm, interconnect energy only scales down by $1.6 \times$ [2]. Similar trends are projected for transistor and wire delay scaling. The many-core processors built from these technologies will execute emerging big data and streaming parallel applications that are expected to process many data structures with varying degrees of locality and reuse. Current multicore caches either fail to capture temporal locality when the reuse distance is too large, or they are simply inefficient for data structures with no or little reuse. Hence, unnecessary data movement not only impacts memory access latency, but also incurs wasteful energy consumption of the networks-on-chip and cache resources [2,3].

1.1 Data access in the multicore last-level cache

A large monolithic on-chip cache that holds the application working set does not scale beyond a small number of cores, and the only practical option is to physically distribute on-chip memory in pieces so that every core is near some portion of the cache [4]. In theory this provides a large amount of aggregate cache capacity and fast private memory for each core. Unfortunately, it is difficult to manage the distributed cache and network resources effectively since they require architectural support for cache coherence and consistency under the ubiquitous shared memory model.

Popular directory-based protocols enable fast local caching to exploit data locality, but scale poorly with increasing core counts [5,6]. Many recent proposals have addressed directory scalability in single-chip multicores using sharer compression techniques or limited directories [7–10]. Yet, fast private caches still suffer from two major problems: (1) due to capacity constraints, they cannot hold the working set of applications that operate on massive data, and (2) due to frequent communication between cores, data is often displaced from them [11]. This leads to increased network traffic and request rate to the last-level cache.

Last-level cache (LLC) organizations offer trade-offs between on-chip data locality and off-chip miss rate. While private LLC organizations (e.g., [12]) have low hit latencies, their off-chip miss rates are high in applications that have uneven distributions of
working sets or exhibit high degrees of sharing (due to cache line replication). Shared LLC organizations (e.g., [13]), on the other hand, lead to non-uniform cache access (NUCA) [14] that hurts on-chip locality, but their off-chip miss rates are low since cache lines are not replicated.

Several proposals have explored the idea of hybrid LLC organizations [15–21]. These proposals attempt to combine the good characteristics of private and shared LLC organizations by relying on either dynamic data placement, data replication, or both. These proposals either do not quickly adapt their policies to dynamic program changes, or replicate cache lines without paying attention to their locality. In addition, some of them either add significant hardware overheads, or complicate coherence and do not scale to large core counts.

1.2 Motivation for locality-aware replication in LLC

Static data placement optimally utilizes the LLC capacity. However, it suffers from sub-optimal placement of all data, thus results in increased on-chip network traffic and high LLC access latency. On the other hand, some key challenges with dynamic data placement schemes are: (1) the sub-optimality of shared data placement, (2) the potential over-subscription of private data at the local LLC slice, and (3) false sharing of cache lines due to misclassification of OS pages. If cache line replication is enabled in the local LLC slice of the requesting core, these challenges can be mitigated, resulting in improved data locality that lowers the LLC access latency, on-chip traffic, and off-chip miss rate.

The utility of data replication in the LLC can be understood by measuring cache line reuse. Figure 1 plots the distribution of the number of accesses to cache lines in the LLC as a function of run-length. Run-length is defined as the number of accesses to a cache line (at the LLC) from a particular core before a conflicting access by another core or before it is evicted. Cache line accesses from multiple cores are conflicting if at least one of them is a write. For example, in BARNES benchmark, over 90% of the

![Figure 1](image-url) Distribution of instructions, private data, shared read-only data, and shared read-write data accesses to the LLC as a function of run-length. The classification is done at the cache line granularity for the evaluated benchmarks, using the methodology described in Sect. 4.

 Springer
accesses to the LLC occur to shared (read-write) data that has a run-length of 10 or more. Greater the number of accesses with higher run-length, greater is the benefit of replicating the cache line in the requester’s LLC slice. Hence, BARNES would benefit from replicating shared (read-write) data. Similarly, FACESIM would benefit from replicating instructions and PATRICIA would benefit from replicating shared (read-only) data. On the other hand, FLUIDANIMATE and OCEAN-C would not benefit since most cache lines experience just 1 or 2 accesses to them before a conflicting access or an eviction. For such cases, replication would increase the LLC pollution without improving data locality. Furthermore, BLACKSCHOLES would benefit because the misclassified private data would now be replicated.

1.3 Proposed idea of locality-aware replication in LLC

We propose a low-overhead yet highly accurate hardware-only predictive mechanism to track and classify the reuse of each cache line in the LLC. A runtime classifier only allows replicating those cache lines that demonstrate reuse at the LLC while bypassing replication for others. When a cache line replica is evicted or invalidated, the classifier adapts by adjusting its future replication decision accordingly. This reuse tracking mechanism is decoupled from the sharer tracking structures that cause scalability concerns in traditional cache coherence protocols. Further optimizations are proposed to the reuse tracking hardware to lower the storage overheads. For this purpose, a limited number of cache lines and a limited number of sharers for each tracked cache line are used to make the LLC replication decisions. The proposed scheme improves over static and dynamic-NUCA baselines by intelligently replicating all types of cache lines in the requester’s local LLC slice. The evaluation using a 64-core multicore shows an improvement of 14.7, 10.7, 10.5, and 16.7 % in energy and 2.5, 6.5, 4.5, and 9.5 % in completion time over Victim Replication (VR) [16], Adaptive Selective Replication (ASR) [17], Reactive-NUCA (R-NUCA) [19], and Static-NUCA (S-NUCA) [14] LLC management schemes, respectively. The locality-aware protocol is advantageous because it:

1. Enables lower memory access latency and energy by selectively replicating cache lines that show high reuse in the LLC slice of the requesting core.
2. Better exploits the LLC by balancing the off-chip miss rate and on-chip locality using a classifier that adapts to the runtime reuse at the granularity of cache lines.
3. Mitigates the necessity of complex dynamic data placement schemes by demonstrating efficient implementations of locality-aware replication in LLC on top of both dynamic and static-NUCA baselines.
4. Allows coherence complexity almost identical to that of a traditional non-hierarchical coherence protocol, since replicas are only allowed to be placed at the LLC slice of the requesting core.
5. Incurs low area overhead by tracking a limited number of cache lines and a limited number of sharers for each tracked cache line. This results in a storage overhead of 5.44 KB per core, on top of the S-NUCA’s on-chip cache capacity. However, the
overall completion time is improved by 9.5% and dynamic energy consumption by 16.7%.

The rest of the paper is organized as follows. Section 2 presents the details of the baseline multicore, and outlines the related work for the data placement and replication in the last-level cache (LLC). Section 3 presents the protocol operations as well as the architecture requirements and description of the proposed locality-aware replication in LLC. A discussion section outlines the replica creation strategy, coherence complexity, replication at the cluster granularity, and the storage efficient classifier organization. Moreover, the overheads section presents a detailed bit storage level breakdown of the proposed classifier. Section 4 describes the modeling strategy for performance and dynamic energy, as well as the parallel benchmarks and LLC schemes used for evaluation. Section 5 evaluates the proposed locality-aware replication in LLC scheme on top of S-NUCA and R-NUCA baselines, and compares it to four popular state-of-the-art LLC management schemes. Sensitivity studies are also presented to evaluate the proposed organization of the locality classifier, replication threshold, impact of LLC size variations, as well as the impact of using an out-of-order core type to implement the multicore. Section 6 concludes the paper.

2 Background and related work

2.1 Baseline system

The baseline system is a tiled multicore with an electrical 2-D mesh interconnection network. Each core consists of a compute pipeline, private L1 instruction and data caches, a physically distributed shared LLC cache with integrated directory, and a network router. Cache coherence is maintained using a MESI protocol. The coherence directory is integrated with the LLC slices by extending the tag arrays (in-cache directory organization [22,23]), and tracks the sharing status of the cache lines in the per-core private L1 caches. The private L1 caches are kept coherent using the ACKwise limited directory-based coherence protocol [24]. Some cores have a connection to a memory controller as well. The memory controllers are placed in a way that minimizes the average distance from the tiles. The electrical mesh network uses dimension-order X-Y routing and wormhole flow control.

The ACKwise protocol maintains a limited set of hardware pointers \( p \) to track the sharers of a cache line. It operates like a full-map protocol when the number of sharers is less than or equal to the number of hardware pointers. When the number of sharers exceeds the number of hardware pointers, the ACKwise \( p \) protocol does not track the identities of the sharers anymore. Instead, it tracks the number of sharers and performs a broadcast invalidate on an exclusive request. However, acknowledgments need to be received from only the actual sharers of the data. In conjunction with a broadcast network, the ACKwise protocol has been shown to scale to large number of cores [24]. The 2-D mesh network is also augmented with broadcast support. Each router selectively replicates a broadcasted message on its output links such that all cores are reached with a single injection.
2.2 Data placement

This section describes state-of-the-art static-NUCA and dynamic-NUCA data placement schemes.

2.2.1 Static-NUCA

The application’s address space is divided in subsets and each subset is mapped to an LLC slice, known as the “home” for that subset of addresses. In case of a private cache miss, a request is sent to the home LLC slice based on a simple address-based decode. Static interleaving at the fine granularity of cache lines fully exploits the on-chip LLC capacity. However, majority of the accesses are to remote LLC slices, resulting in an increase in on-chip network traffic. In short, it utilizes the on-chip capacity efficiently; however, it hurts the locality of private data. This scheme is straightforward to implement and does not come with any area overhead.

2.2.2 Dynamic-NUCA

Reactive-NUCA (R-NUCA) [19] classifies data as private or shared at page granularity using the existing operating system virtual memory management mechanism. All the cache lines of a private page are placed at the requesting core’s LLC slice. Shared data tends to exhibit limited affinity to a particular core since it is accessed by multiple cores over time. Therefore, cache lines of shared pages are assigned a home core based on static address interleaving. This allows R-NUCA to exploit locality on private data and enable low average access latency/energy for private data. However, it still suffers from high average access latency/energy for shared data access.

A drawback of R-NUCA is that it requires modifications in the virtual memory system, which has several serious ramifications. Firstly, the OS needs to be modified to take into account the new information in the page table and Translation Lookaside Buffer (TLB). Secondly, the page table and TLB structures are modified and the hardware now needs to support migration of data pages at runtime. The OS support for page classification extends each TLB entry by 1 bit and each page table entry by an additional \( \log_2(n) \) bits, where \( n \) is the number of cores. Additionally, in workloads with primarily private data, the on-chip cache capacity is underutilized and can result...
in higher off-chip miss rate. Similar underutilization can occur in a multi-program setup. Finally, the page-level classification under certain circumstances results in false classification of private data as shared. This is evident in Fig. 2 which plots the distribution of the number of accesses in the LLC to instructions, private data, shared read-only data, and shared read–write data at the cache line and page level separately. Due to false sharing, almost all true private data in the BLACKSCHOLES and OCEAN_NON_CONTIGUOUS benchmarks is classified as shared at the page level.

2.2.3 Other related work

Cho et al. [25] propose an OS-assisted data allocation scheme using S-NUCA organization and page coloring. It uses the first-touch color assignment, i.e., a page is assigned a color that places it in the LLC slice nearest to the core that first touches that page. This approach potentially leads to inefficient use of on-chip cache capacity, and puts pressure on some heavily used LLC slices. To mitigate this challenge, it proposes to augment first-touch with page-spreading policies. Therefore, subsequent page requests from that core are assigned colors that map the pages to neighboring tile’s LLC slices. Awasthi et al. [26] and Chaudhuri et al. [18] build on top of first-touch page coloring to solve the challenge of sub-optimal home location for shared data. They propose a further level of indirection to migrate pages in order to improve the locality of shared data. Although a step in the right direction, they introduce several non-trivial overheads in the process. R-NUCA is a relatively simpler dynamic-NUCA scheme that avoids adding complexity to the underlying coherence protocol. Therefore, R-NUCA is chosen as the representative dynamic data placement scheme.

2.3 Data replication

This section describes the data replication schemes used in our evaluation, as well as other related work to complete the discussion. Cache line replication in the LLC can be deployed to improve locality for instructions, shared, and private data. Furthermore, misclassified private data, as seen in Fig. 2, can also be replicated in the local LLC slice of the requester. In order to enable data replication at the cache line granularity, the baseline lookup strategy for the LLC is slightly modified for all schemes. The local LLC slice is always looked up on an L1 cache miss or eviction. Additionally, both the L1 cache and LLC slice are probed on every asynchronous coherence request (i.e., invalidate, downgrade, flush or write-back). This is needed because the directory only has a single pointer to track the local cache hierarchy of each core. This method also allows the coherence complexity to be similar to that of a non-hierarchical (flat) coherence protocol.

2.3.1 Victim Replication

In Victim Replication (VR) [16], a cache line is replicated in the local LLC slice upon eviction from the L1 cache if either, (1) an invalid cache line exists, (2) a home cache
line with no sharers exists, or (3) another replica exists in the target set. VR implements a random replacement policy within a candidate group, with priority in the order listed above. If no such cache line exists, the evicted cache line is communicated back to its home core. Upon a subsequent access resulting in a hit, the replica is invalidated in the local LLC slice and moved to the L1 cache. Because of this exclusive relationship, clean cache lines that would otherwise be simply invalidated are sent back to the local LLC upon eviction. This results in unnecessary movement of data between the caches, increasing the cache energy consumption.

2.3.2 Adaptive Selective Replication

Adaptive Selective Replication (ASR) [17] replicates cache lines in the requester’s local LLC slice on an L1 cache eviction. However, it only allows LLC replication for cache lines that are classified as shared read-only and does not allow replication for other types of data. ASR implements hardware monitoring circuits that quantify the replication effectiveness and then probabilistically decides whether to replicate or not on a per-core basis. ASR assumes an LLC lookup mechanism must search all LLC slices, which does not scale to large core counts. Furthermore, the hardware monitoring circuits add a storage overhead of more than 18 KB per core.

2.3.3 Locality-aware data replication

Motivated by the fact that cache lines exhibit varying degrees of reuse at the LLC, as discussed earlier in Fig. 1, the locality-aware data replication scheme is proposed [27]. This scheme replicates cache lines in the local LLC slice based on their reuse captured using an in-hardware cache line level classifier. A cache-line level classifier is introduced at the LLC to distinguish between low and high reuse cache lines. Cache lines that exhibit high reuse are allowed to be replicated in the local LLC slice of the requesting core, while bypassing replication in the local LLC slice for the others.

The overhead of tracking the reuse information for all cache line sharers and for every cache line in the LLC (called “Complete classifier”), is prohibitively high (cf. Sect. 3.5). To mitigate this bottleneck, a horizontal reduction is proposed by tracking only a few sharers for each cache line and taking a majority vote for other sharers, this is called the Limited$^k$ classifier, where ‘k’ represents the number of sharers tracked for each cache line. To make the replication scheme even more practical, a vertical reduction is proposed by tracking only a subset of the LLC cache lines, this is called the Limited$^k$-$m$ classifier, where ‘m’ represents the number of cache lines tracked in each LLC slice. The locality information for these limited number of cache lines is stored in a self-standing structure, called “locality table”. This organization completely decouples the locality tracking structure from the directory. The locality tracking does not interfere with the underlying coherence protocol. However, a parallel lookup is now required to extract the locality information. This does not impact performance, as the lookup is not in the critical path and is hidden by the parallel LLC slice lookup. The locality-aware replication scheme and its optimizations are discussed in detail in Sect. 3.
2.3.4 Other related work

**CMP-NuRAPID** [15] decouples tag and data arrays. It maintains private per-core tag arrays and a shared data array divided into multiple distance groups. It replicates shared read-only cache lines in the cache bank closest to the requesting core on its second access. CMP-NuRAPID does not allow replication of shared read-write data and only one copy can exist in the cache. Also, it forces write-through for such cache lines to maintain coherence using an additional Communication (C) coherence state. As each private per-core tag array potentially has to store pointers to the entire data array, CMP-NuRAPID does not scale with the number of cores. In addition, CMP-NuRAPID requires snooping coherence to broadcast invalidate replicas, and complicates the coherence protocol significantly by introducing many additional race conditions that arise from the tag and data array decoupling.

Proposals that start with a private-L1, private LLC organizations have been proposed [28–32]. These proposals attempt to improve the negative feature of private LLC, i.e., the low hit rates, at the same time try to capitalize on the favorable property, i.e., low LLC access latency. They start with a private-private organization and then either spill data into other private caches, or reconfigure the private cache size to reduce the off-chip misses and efficiently utilize the on-chip capacity. However, keeping track of and figuring out where to place the spilled data is a challenge by itself. These schemes generally rely on coarse-grain information that is not very accurate. Furthermore, they all suffer from the complex hardware structures required to keep the private caches coherent.

3 Locality-aware data replication in the last-level cache

Locality-aware data replication protocol for the LLC starts off with the static data placement scheme (S-NUCA). It then captures the reuse of cache lines using an in-hardware cache line level classifier. If enough reuse is observed for a certain sharer of a cache line, it replicates that cache line in that sharer’s local LLC slice. For sharers that exhibit only low reuse, it disallows replication in the local LLC slice. This section first describes the protocol operation, classifier organization and architecture, and finally the overheads.

3.1 Protocol operation

The four essential components of data replication in LLC are: (1) choosing which cache lines to replicate, (2) determining where to place a replica, (3) how to lookup a replica, and (4) how to maintain coherence for replicas. Lets define a few terms to facilitate describing the protocol.

- **Home location** is the core where all requests for a cache line are serialized for maintaining coherence.
- **Replica sharer** is a core that is granted a replica of a cache line in its LLC slice.
Locality-aware data replication in the last-level cache...

Fig. 3  a Tiled architecture. b Classifier state machine. a ①–③ are mockup requests showing the locality-aware LLC replication protocol. The black data block has high reuse and a local LLC replica is allowed that services requests from ① and ②. The low reuse data block is not allowed to be replicated at the LLC, and the request from ③ that misses in the L1, must access the LLC slice at its home core. The home core for each data block can also service local private cache misses (④).

(b) Each directory entry is extended with replication mode bits to classify the usefulness of LLC replication. Each cache line is initialized to non-replica mode with respect to all cores. Based on the reuse counters (at the home as well as the replica location) and the parameter RT, the cores are transitioned between replica and non-replica modes. Here XReuse is (Replica + Home) Reuse on an invalidation and Replica Reuse on an eviction.

- **Non-replica sharer** is a core that is NOT granted a replica of a cache line in its LLC slice.
- **Replica reuse** is the number of times an LLC replica is accessed before it is invalidated or evicted.
- **Home reuse** is the number of times a cache line is accessed at the LLC slice in its home location before a conflicting write or eviction.
- **Replication threshold (RT)** is the reuse above or equal to which a replica is created.

Note that for a cache line, one core can be a replica sharer while another can be a non-replica sharer. The protocol starts out as a conventional directory protocol and initializes all cores as non-replica sharers of all cache lines (as shown by Initial in Fig. 3b). Let us understand the handling of read requests, write requests, evictions, invalidations, and downgrades as well as cache replacement policies under this protocol.
3.1.1 Read requests

On an L1 cache read miss, the core first looks up its local LLC slice for a replica. If a replica is found, the cache line is inserted at the private L1 cache. In addition, a Replica Reuse counter (as shown in Fig. 4) at the LLC directory entry is incremented. The replica reuse counter is a saturating counter used to capture reuse information. It is initialized to ‘1’ on replica creation and incremented on every replica hit.

On the other hand, if a replica is not found, the request is forwarded to the LLC home location. If the cache line is not found there, it is either brought in from the off-chip memory or the underlying coherence protocol takes the necessary actions to obtain the most recent copy of the cache line. The directory entry is augmented with additional bits as shown in Fig. 4. These bits include (a) Replication Mode bit and (b) Home Reuse saturating counter for each core in the system. Note that adding several bits for tracking the locality of each core in the system does not scale with the number of cores, therefore, cost-efficient classifier implementations are presented in Sects. 3.2 and 3.3. The replication mode bit is used to identify whether a replica is allowed to be created for the particular core. The home reuse counter is used to track the number of times the cache line is accessed at the home location by the particular core. This counter is initialized to ‘0’ and incremented on every hit at the LLC home location.

If the replication mode bit is set to true, the cache line is inserted in the requester’s LLC slice and the private L1 cache. Otherwise, the home reuse counter is incremented. If this counter has reached the Replication Threshold (RT), the requesting core is “promoted” (the replication mode bit is set to true) and the cache line is inserted in its LLC slice and private L1 cache. If the home reuse counter is still less than RT, a replica is not created. The cache line is only inserted in the requester’s private L1 cache.

If the LLC home location is at the requesting core, the read request is handled directly at the LLC home. Even if the classifier directs to create a replica, the cache line is just inserted at the private L1 cache.

3.1.2 Write requests

On an L1 cache write miss for an exclusive copy of a cache line, the protocol checks the local LLC slice for a replica. If a replica exists in the Modified (M) or Exclusive (E) state, the cache line is inserted at the private L1 cache. In addition, the Replica Reuse counter is incremented.
If a replica is not found or exists in the Shared (S) state, the request is forwarded to the LLC home location. The directory invalidates all the LLC replicas and L1 cache copies of the cache line, thereby maintaining the single-writer multiple-reader invariant [33]. The acknowledgments received are processed as described in Sect. 3.1.3. After all such acknowledgments are processed, the Home Reuse counters of all non-replica sharers other than the writer are reset to ‘0’. This has to be done since these sharers have not shown enough reuse to be “promoted”.

If the writer is a non-replica sharer, its home reuse counter is modified as follows. If the writer is the only sharer (replica or non-replica), its home reuse counter is incremented, else it is reset to ‘1’. This enables the replication of migratory shared data at the writer, while avoiding it if the replica is likely to be downgraded due to conflicting requests by other cores.

3.1.3 Evictions and invalidations

On an invalidation request, both the LLC slice and L1 cache on a core are probed and invalidated. If a valid cache line is found in either caches, an acknowledgment is sent to the LLC home location. In addition, if a valid LLC replica exists, the replica reuse counter is communicated back with the acknowledgment. The locality classifier uses this information along with the home reuse counter to determine whether the core stays as a replica sharer. If the (replica + home) reuse is \( \geq RT \), the core maintains replica status, else it is demoted to non-replica status (as shown in Fig. 3b). The two reuse counters have to be added since this is the total reuse that the core exhibited for the cache line between successive writes.

When an L1 cache line is evicted, the LLC replica location is probed for the same address. If a replica is found, the dirty data in the L1 cache line is merged with it, else an acknowledgment is sent to the LLC home location. However, when an LLC replica is evicted, the L1 cache is probed for the same address and invalidated. An acknowledgment message containing the replica reuse counter is sent back to the LLC home location. The replica reuse counter is used by the locality classifier as follows. If the replica reuse is \( \geq RT \), the core maintains replica status, else it is demoted to non-replica status. Only the replica reuse counter has to be used for this decision since it captures the reuse of the cache line at the LLC replica location.

After the acknowledgment corresponding to an eviction or invalidation of the LLC replica is received at the home, the locality classifier sets the home reuse counter of the corresponding core to ‘0’ for the next round of classification.

The eviction of an LLC replica back-invalidates the L1 cache (as described earlier). A possibly more optimal strategy is to maintain the validity of the L1 cache line. This requires two message types as well as two messages, one to communicate back the reuse counter on the LLC replica eviction and another to communicate the acknowledgment when the L1 cache line is finally invalidated or evicted. We opted for the back-invalidation for two reasons: (1) to maintain the simplicity of the coherence protocol, and (2) the energy and performance improvements of the more optimal strategy are negligible since (a) the LLC is more than 4× larger than the L1 cache, thereby keeping the probability of evicted LLC lines having an L1 copy extremely low, and (b) our LLC replacement policy prioritizes retaining cache lines that have L1 cache copies.
3.1.4 LLC replacement policy

Traditional LLC replacement policies use the least recently used (LRU) policy. One reason why this is sub-optimal is that the LRU information cannot be fully captured at the LLC because the L1 cache filters out a large fraction of accesses that hit within it. In order to be cognizant of this, the replacement policy should prioritize retaining cache lines that have L1 cache sharers. Some proposals in literature accomplish this by sending periodic Temporal Locality Hint messages from the L1 cache to the LLC [34]. However, this incurs additional network traffic.

The proposed replacement policy accomplishes the same using a much simpler scheme. It first selects cache lines with the least number of L1 cache copies and then chooses the least recently used among them. The number of L1 cache copies is readily available since the directory is integrated within the LLC tags (“in-cache” directory). This reduces back invalidations to a negligible amount. It should be noted that a replica is treated the same as a home cache line with no sharers. Furthermore, a replica insertion can also evict a home cache line, which can have some undesirable effects (e.g. too many dirty home evictions) in some benchmarks.

3.2 Limited \( k \) locality classifier

The classifier described so far keeps track of the locality information for all cores in the directory entry. It is termed the Complete locality classifier, and has a storage overhead of \( \sim 30\% \) (calculated in Sect. 3.5) at 64 cores and over \( 5\times \) at 1024 cores. In order to mitigate this overhead, a space efficient classifier is proposed that maintains locality information for a limited number of cores, and classifies the other cores as replica or non-replica sharers based on this information.

The locality information for each core consists of (1) the core ID, (2) the replication mode bit and (3) the home reuse counter. The classifier that maintains a list of this information for a limited number of cores \( (k) \) is termed the Limited \( k \) classifier. Figure 5 shows the information that is tracked by this classifier. The sharer list of the ACKwise limited directory entry cannot be reused for tracking locality information because of its different functionality. While the hardware pointers of ACKwise are used to maintain coherence, the limited locality list serves to classify cores as replica or non-replica sharers. Decoupling in this manner also enables the locality-aware protocol to be

![Fig. 5 ACKwise, Limited locality classifier LLC tag entry. It contains the tag, LRU bits and directory entry. The directory entry contains the state, ACKwise pointers, a Replica reuse counter as well as the Limited classifier. The Limited classifier contains a Replication mode bit and Home reuse counter for a limited number of cores. A majority vote of the modes of tracked cores is used to classify new cores as replicas or non-replicas.](image-url)
implemented efficiently on top of other scalable directory organizations. The working of the limited locality classifier will now be described.

At startup, all entries in the limited locality list are free and this is denoted by marking all core IDs’ as Invalid. When a core makes a request to the home location, the directory first checks if the core is already being tracked by the limited locality list. If so, the actions described previously are carried out. Else, the directory checks if a free entry exists. If it does exist, it allocates the entry to the core and the same actions are carried out.

Otherwise, the directory checks if a currently tracked core can be replaced. An ideal candidate for replacement is a core that is currently not using the cache line. Such a core is termed an inactive sharer and should ideally relinquish its entry to a core in need of it. A replica core becomes inactive on an LLC invalidation or an eviction. A non-replica core becomes inactive on a write by another core. If such a replacement candidate exists, its entry is allocated to the requesting core. The initial replication mode of the core is obtained by taking a majority vote of the modes of the tracked cores. This is done so as to start off the requester in its most probable mode.

Finally, if no replacement candidate exists, the mode for the requesting core is obtained by taking a majority vote of the modes of all the tracked cores. The limited locality list is left unchanged. The storage overhead for the Limited\(_k\) classifier is directly proportional to the number of cores (\(k\)) for which locality information is tracked. In Sect. 5.2, the storage and accuracy tradeoffs for the Limited\(_k\) classifier are evaluated.

### 3.3 Limited\(_k\)−\(m\) locality classifier

A completely decoupled locality classifier from the LLC tag array is explored to further reduce the storage overhead. This is achieved by tracking a limited number of cache lines, as opposed to all the cache lines in Limited\(_k\) classifier. To enable tracking of a limited number of cache lines, a set-associative cache like structure, locality table is introduced that is accessed in parallel with the LLC, shown in Fig. 6. Each entry in the locality table stores the Limited\(_k\) classifier information for one cache line along with the tag, valid bit, and LRU bits. LRU replacement policy is used to determine a replacement candidate within a set. This is termed as a Limited\(_k\)−\(m\) classifier, where ‘\(m\)’ represents the number of cache lines being tracked. The working of the locality table will now be described.

At startup, all entries in the locality table are free and this is denoted by marking all valid bits as Invalid. When a core makes a request to the home location, the locality table is also looked up in parallel to the LLC slice. If it results in a hit, it means that the cache line is being tracked. At this point, it is checked whether the core is already being tracked by the limited locality list and the actions described in Sect. 3.2 are performed. On the other hand, if the lookup results in a miss, the classifier information for that cache line is inserted into the locality table, and the cache line is now tracked for reuse classification. Note that the classifier information for the evicted cache line from the table is lost and the classifier needs to re-learn if the same cache line is inserted again in the locality table at some time in the future.
The eviction candidate is selected using the LRU replacement policy. In the locality table, if a cache line is heavily communicated back and forth between the cache hierarchies, it remains in the locality table and the mode is updated accordingly. On the other hand, if a cache line is used sparingly, its entry in the locality table is eventually taken by another cache line. The LRU policy captures this access pattern, and retains cache lines with useful reuse behavior in the locality table.

The storage overhead for theLimited$_k-m$ classifier is directly proportional to the number of cache lines ($m$) for which locality information is tracked. In Sect. 5.2, the storage and accuracy tradeoffs for theLimited$_k-m$ classifier are evaluated. Based on our observations, we pick the Limited$_3-512$ classifier.

### 3.4 Discussion

#### 3.4.1 Replica creation strategy

In the locality-aware protocol, replicas are allowed to be created in all valid cache states. A simpler strategy is to create an LLC replica only in the Shared cache state. This enables instructions, shared read-only and shared read–write data that exhibit high read run-length to be replicated so as to serve multiple read requests from within the local LLC slice. However, migratory shared data cannot be replicated with this simpler strategy because both read and write requests are made to it in an interleaved manner. Such data patterns can be efficiently handled only if the replica is created in both Exclusive and Modified states as well. Benchmarks that exhibit both the above access patterns are observed in our evaluation (cf. Sect. 5.1.1).

#### 3.4.2 Coherence complexity

The local LLC slice is always looked up on an L1 cache miss or eviction. Additionally, both the L1 cache and LLC slice is probed on every asynchronous coherence request (i.e., invalidate, downgrade, flush or write-back). This is needed because the directory only has a single pointer to track the local cache hierarchy of each core. This method
also allows the coherence complexity to be similar to that of a non-hierarchical (flat) coherence protocol.

To avoid the latency and energy overhead of searching the LLC replica, one may want to optimize the handling of asynchronous requests, or decide intelligently whether to lookup the local LLC slice on a cache miss or eviction. In order to enable such optimization, additional sharer tracking bits are needed at the directory and L1 cache. Moreover, additional network message types are needed to relay coherence information between the LLC home and other actors.

In order to evaluate whether this additional coherence complexity is worthwhile, we compare the proposed protocol to a dynamic oracle that has perfect information about whether a cache line is present in the local LLC slice. The dynamic oracle avoids all unnecessary LLC lookups. The completion time and energy difference when compared to the dynamic oracle is less than 1%. Hence, in the interest of avoiding the additional complexity, the LLC replica is always looked up for the above coherence requests.

3.4.3 Cluster-level replication

In the locality-aware protocol, the location where a replica is placed is always the LLC slice of the requesting core. An additional method by which one could explore the trade-off between LLC hit latency and LLC miss rate is by replicating at a cluster-level. A cluster is defined as a group of neighboring cores where there is at most one replica for a cache line. Increasing the size of a cluster would increase LLC hit latency and decrease LLC miss rate, and decreasing the cluster size would have the opposite effect. The optimal replication algorithm would optimize the cluster size so as to maximize the performance and energy benefit.

Cluster-level replication is not found to be beneficial [27]. The reasons include: (1) Using clustering increases network serialization delays since multiple locations now need to be searched/invalidated on an L1 cache miss. (2) Cache lines with low degree of sharing do not benefit because clustering just increases the LLC hit latency without reducing the LLC miss rate. (3) The added coherence complexity of clustering increases the design and verification time significantly.

3.4.4 Classifier organization

The Complete locality classifier for the locality-aware protocol is organized using an in-cache structure, i.e., the replication mode bits and home reuse counters are maintained for all cache lines in the LLC. However, this is a not an essential requirement. The classifier is logically decoupled from the directory and could be implemented using a sparse organization.

Number of tracked sharers Limited\(k\) with \(k = 64\) corresponds to the Complete locality classifier for a 64-core multicore. To reduce the prohibitively high overhead of the Complete locality classifier, a limited number of sharers are tracked, as discussed in Sect. 5.2. Although Limited\(1\) has the lowest overhead, the classifier is more unstable than the others and performs significantly worse for BARNES and STREAMCLUSTER.
Limited3 is found to be the optimal classifier with substantially lower overhead and performance close to that of the Complete classifier (cf. Sect. 5.2).

**Number of tracked cache lines** Limited \( k_m \) classifier with \( m = 4096 \) corresponds to the Limited3 classifier for a 256 KB per-core LLC slice. Reducing the number of tracked cache lines potentially impacts the accuracy of the classifier. On the other hand, not all cache lines are used at all times and tracking only a subset of cache lines can help reduce the associated overheads. The goal is to find a balance between the overhead reduction and the completion time/energy penalty paid for it. In order to quantify this tradeoff, experiments are run with varying number of tracked cache lines. It is observed that tracking 512 cache lines provides completion time/energy approaching that of tracking all cache lines (cf. Sect. 5.2), while reducing the space overhead by more than 2.5\times over Limited3 classifier.

**Locality table associativity** To find the right structure of the locality table, experiments are performed with varying associativity of the locality table while keeping the number of tracked cache lines constant. We do not observe significant variations, but find associativity of 8 provides the best energy-delay product (cf. Sect. 5.2).

### 3.5 Overheads

#### 3.5.1 Storage

**Complete classifier** The locality-aware protocol requires extra bits at the LLC tag arrays to track cache line reuse information. Each LLC directory entry requires 2 bits for the replica reuse counter (assuming an optimal replication threshold, \( RT \) of 3). Tracking one core requires 2 bits for the home reuse counter, and 1 bit to store the replication mode. Hence the Complete classifier, requires \( 192 (=64 \times 3) \) bits of storage per LLC directory entry.

All the following calculations are for one core but they are applicable for the entire processor since all the cores are identical. The sizes of the per-core L1 and LLC caches used in our system are shown in Table 1. The storage overhead of the replica reuse bit is \( \frac{2 \times 256}{64 \times 8} = 1 \) KB. For the complete classifier, it is \( \frac{192 \times 256}{64 \times 8} = 96 \) KB, resulting in a total overhead of 97 KB. The Complete classifier uses 28.25% more storage than the S-NUCA baseline’s cache related per-core storage.

**Limited\( k \) classifier** The Limited3 classifier tracks the locality information for three cores. Tracking one core requires 2 bits for the home reuse counter, 1 bit to store the replication mode, and 6 bits to store the core ID (for a 64-core processor). Hence, the Limited3 classifier requires an additional 27 \( (=3 \times 9) \) bits of storage per LLC directory entry. The storage overhead of the Limited3 classifier is \( \frac{27 \times 256}{64 \times 8} = 13.5 \) KB. To track the replica reuse, 2 bits are added to the tag of each cache line, resulting in \( \frac{2 \times 256}{64 \times 8} = 1 \) KB overhead for a 256 KB LLC slice. The overall storage overhead of the Limited3 classifier is 14.5 KB. The Limited3 classifier uses 4.22% more storage than the S-NUCA baseline’s cache related per-core storage.
**Limited** \(_{k-m}\) **classifier** The locality table requires 27 bits of storage per tracked LLC cache line and 2 bits per LLC cache line, similar to Limited\(_3\) classifier. The reduction comes from tracking limited number of cache lines. Each tracked cache line now needs a tag, a valid bit, and LRU bits along with the Limited\(_3\) classifier bits. The total overhead for each tracked cache line is \(40 + 1 + 3 + 27 = 71\) bits. For 512 tracked cache lines, Limited\(_{3-512}\) overhead is \(1\) KB + \(\frac{71 \times 512}{1024 \times 8} = 5.44\) KB per core. This is an increase of 1.58% in storage compared to the S-NUCA baseline’s cache related per-core storage. This also translates to a reduction of \(\sim 2.5\times\) and \(\sim 17.5\times\) over Limited\(_3\) classifier and the Complete classifier respectively.

### 3.5.2 LLC tag, directory, and classifier accesses

Updating the replica reuse counter in the local LLC slice requires a read–modify–write operation on each replica hit. However, since the replica reuse counter (being 2 bits) is stored in the LLC tag array that needs to be written on each LLC lookup to update the LRU counters, our protocol does not add any additional tag accesses.

At the home location, the lookup/update of the locality information is performed concurrently with the lookup/update of the sharer list for a cache line. However, the lookup/update of the directory is now more expensive since it includes both sharer list and the locality information in case of the Limited\(_3\) classifier. For Limited\(_{3-512}\) classifier, each LLC access is now more expensive in energy because of the additional parallel lookup required for the locality table. This additional expense is accounted for in the evaluation. However, the locality table lookup does not incur any additional latency as it is performed in parallel with LLC tag lookup.

### 3.5.3 Network traffic

The locality-aware protocol communicates the replica reuse counter to the LLC home along with the acknowledgment for an invalidation or an eviction. This is accomplished **without** creating additional network flits. For a 48-bit physical address and 64-bit flit size, an invalidation message requires 42 bits for the physical cache line address, 12 bits for the sender and receiver core IDs and 2 bits for the replica reuse counter. The remaining 8 bits suffice for storing the message type.

### 4 Evaluation methodology

The evaluation is conducted for a 64-core shared memory multicore. The default architectural parameters used for evaluation are shown in Table 1. Single-issue, in-order compute cores are modeled because the power consumption of complex out-of-order cores can be prohibitively high at large core count. Furthermore, the high concurrency in present and future applications makes the case for using many simple cores, as compared to a lower number of complex out-of-order cores. However, results for an out-of-order-based multicore are also presented to demonstrate applicability of the proposed architecture in such configuration.
### Table 1  Architectural parameters for evaluation

| Architectural parameter                  | Value                                      |
|-----------------------------------------|--------------------------------------------|
| Number of cores                         | 64 @ 1 GHz                                 |
| Compute pipeline per core               | In-order, single-issue                     |
| Physical address length                 | 48 bits                                    |
| Memory subsystem                        |                                            |
| L1-I Cache per core                     | 32 KB, 4-way Assoc., 1 cycle               |
| L1-D Cache per core                     | 32 KB, 4-way Assoc., 1 cycle               |
| L2 Cache per core                       | 256 KB, 8-way Assoc., 2 cycle tag, 6 cycle data Inclusive |
| Cache line size                         | 64 bytes                                   |
| Directory protocol                      | Invalidation-based MESI ACKwise4 [24]      |
| Num. of memory controllers              | 8                                          |
| DRAM bandwidth                          | 5 GBps per Controller                      |
| DRAM latency                            | 75 ns                                      |
| Electrical 2-D mesh with XY routing     |                                            |
| Hop latency                             | 2 Cycles (1-router, 1-link)                |
| Contention model                        | Only link contention (Infinite input buffers) |
| Flit width                              | 64 bits                                    |

### 4.1 Performance models

All experiments are performed using the core, cache hierarchy, coherence protocol, memory system and on-chip interconnection network models implemented within the Graphite multicore simulator [35]. All mechanisms and protocol overheads discussed in Sect. 3 are modeled. The electrical mesh interconnection network uses XY routing. Since modern network-on-chip routers are pipelined [36], and 2- or even 1-cycle per-hop router latencies [37] have been demonstrated, a 2-cycle per-hop delay is modeled; the appropriate pipeline latencies associated with loading and unloading a packet onto the network are also accounted for. In addition to the fixed per-hop latency, network contention delays are also modeled.

### 4.2 Energy models

For energy evaluations of on-chip electrical network routers and links, the DSENT tool [38] is used. Energy estimates for the L1-I, L1-D, L2 (with integrated directory) caches, and DRAM are obtained using McPAT [39]. McPAT uses CACTI [40] internally to model storage elements. The energy evaluation is performed at the 11 nm technology node to account for future technology trends. The models are derived for
Table 2  Projected transistor parameters for 11 nm tri-gate

| Parameter                                      | Value     |
|------------------------------------------------|-----------|
| Process supply voltage ($V_{DD}$)              | 0.6 V     |
| Gate length                                    | 14 nm     |
| Contacted gate pitch                           | 44 nm     |
| Gate cap/width                                 | 2.420 fF/µm |
| Drain cap/width                                | 1.150 fF/µm |
| Effective on current/width (N/P)               | 739/668 µA/µm |
| Off current/width                              | 1 nA/µm   |

a tri-gate 11 nm electrical technology node using the virtual-source transport models of [41] and the parasitic capacitance model of [42]. These models are used to obtain electrical technology parameters (Table 2) used by both McPAT and DSENT. As clock frequencies are relatively slow, high threshold transistors are assumed for lower leakage.

The overall tool flow for energy modeling is as follows. Graphite simulates a benchmark for the chosen system configuration, producing event counters and performance results. The specified cache and network configurations are also fed into McPAT and DSENT to obtain dynamic per-event energy for each component. Event counters and completion time output from Graphite are then combined with per-event energies to obtain the overall dynamic energy usage of the benchmark.

4.3 Benchmarks and evaluation metrics

Each multithreaded benchmark is run to completion using the input sets from Table 3. For performance, the completion time is measured, i.e., the time in the parallel region of the benchmark. This includes the compute latency, the memory access latency, and the synchronization latency. The memory access latency is further broken down into:

1. **L1 to LLC replica latency** is the time spent by the L1 cache miss request to the LLC replica location and the corresponding reply from the LLC replica including time spent accessing the LLC.
2. **L1 to LLC home latency** is the time spent by the L1 cache miss request to the LLC home location and the corresponding reply from the LLC home including time spent in the network and first access to the LLC.
3. **LLC home waiting time** is the queueing delay at the LLC home incurred because requests to the same cache line must be serialized to ensure memory consistency.
4. **LLC home to sharers latency** is the round-trip time needed to invalidate sharers and receive their acknowledgments. This also includes time spent requesting and receiving synchronous write-backs.
5. **LLC home to off-chip memory latency** is the time spent accessing memory including the time spent communicating with the memory controller and the queueing delay incurred due to finite off-chip bandwidth.
Table 3  Problem sizes for our parallel benchmarks

| Application                  | Problem size                                      |
|------------------------------|--------------------------------------------------|
| SPLASH-2 [43]                |                                                  |
| RADIUS                       | 4M Integers, radix 1024                          |
| FFT                          | 4M complex data points                           |
| LU_CONTIGUOUS                | 1024 × 1024 matrix                               |
| LU_NON_CONTIGUOUS            | 1024 × 1024 matrix                               |
| CHOLESKY                     | tk29.0                                           |
| BARNES                       | 64K particles                                     |
| OCEAN_CONTIGUOUS             | 1026 × 1026 ocean                                |
| OCEAN_NON_CONTIGUOUS         | 1026 × 1026 ocean                                |
| WATER-NSQUARED               | 512 molecules                                    |
| RAYTRACE                     | Car                                              |
| VOLREND                      | Head                                             |
| PARSEC [44]                  |                                                  |
| BLACKSCHOLES                 | 64 K options                                     |
| SWAPIONS                     | 64 swaptions, 40,000 sims                         |
| FLUIDANIMATE                 | 5 frames, 300,000 particles                      |
| STREAMCLUSTER                | 8192 points per block, 1 block                   |
| DEDUP                        | 31 MB data                                       |
| FERRET                       | 256 queries, 34,973 images                        |
| BODYTRACK                    | 4 frames, 4000 particles                         |
| FACESIML                     | 1 frame, 372,126 tetrahedrons                    |
| OLTP Database Management System [45] |                            |
| YCSB                         | 1 GB database                                    |
| TPCC                         | 1 GB database                                    |
| Others: Parallel MI Bench [46], UHPC Graph benchmark [47], CRONO [48] |                  |
| PATRICIA                     | 10,000 IP address queries                        |
| ALL-PAIRS-SHORTEST-PATH      | Graph with $2^{18}$ nodes, 16 edges              |
| BETW_CENT                    | Graph with $2^{18}$ nodes, 16 edges              |
| CONNECTED-COMPONENTS         | Graph with $2^{18}$ nodes                        |

The cache miss types are tracked to evaluate the proposed protocol. They are as follows:

1. **LLC replica hits** are L1 cache misses that hit at the LLC replica location.
2. **LLC home hits** are L1 cache misses that hit at the LLC home location when routed directly to it or LLC replica misses that hit at the LLC home location.
3. **Off-chip misses** are L1 cache misses that are sent to DRAM because the cache line is not present on-chip.
4.4 Simulated schemes

The following LLC management schemes are implemented and evaluated.

1. **S-NUCA** address interleaves all cache lines among all LLC slices.
2. **R-NUCA** places private data at the requester’s LLC slice, replicates instructions in one LLC slice per cluster of 4 cores using rotational interleaving, and address interleaves shared data among all LLC slices.
3. **VR** starts with S-NUCA and uses the requester’s local LLC slice as a victim cache for data that is evicted from the L1 cache. The evicted victims are placed in the local LLC slice only if a line is found that is either invalid, has no sharers, or is a replica itself in the L1 cache.
4. **ASR** also replicates cache lines in the requester’s local LLC slice on an L1 eviction. However, it only allows LLC replication for cache lines that are classified as shared read-only. ASR pays attention to the LLC pressure by basing its replication decision on per-core hardware monitoring circuits that quantify the replication effectiveness based on the benefit (lower LLC hit latency) and cost (higher LLC miss latency) of replication. The hardware monitoring circuits or the dynamic adaptation of replication levels are not modeled. Instead, ASR is run at five different replication levels (0, 0.25, 0.5, 0.75, 1) and the setting with the lowest energy-delay product is chosen for each benchmark.
5. **RT3_3_512** starts with S-NUCA and replicates a cache line in the local LLC slice if the reuse of that cache line is above the replication threshold ($RT$). Section 5.3 evaluates a sweep study for a range of $RT$ values and chooses ‘3’ for the evaluation. Furthermore, the locality tracking is done using the Limited$_{3-512}$ classifier. This setting is justified in the evaluation Sect. 5.2.
6. **RT3_3_512_R** is similar to **RT3_3_512**, however it is built on top of the R-NUCA data placement policy.

5 Results

5.1 Comparison of LLC replication schemes

5.1.1 Completion time and energy tradeoffs

Figures 7 and 8 plot the energy and completion time breakdown for the LLC replication schemes evaluated. The energy and completion time trends can be understood based on the following factors: (1) the type of data accessed at the LLC (instruction, private data, shared read-only data and shared read–write data), (2) reuse run-length at the LLC, and (3) working set size of the benchmark. Figure 9, which plots how L1 cache misses are handled at the LLC, is also instrumental in understanding these trends. The **RT3_3_512** bar corresponds to the locality-aware scheme with replication threshold of 3 and Limited$_{3-512}$ locality classifier with S-NUCA data placement. **RT3_3_512_R** is similar to **RT3_3_512** with R-NUCA’s data placement.

As a general trend VR shows some benefits in completion time over S-NUCA. However, it exhibits higher LLC energy than the other schemes for two reasons. (1) Its
Fig. 7 Energy breakdown for the LLC replication schemes evaluated
Locality-aware data replication in the last-level cache... 741

Fig. 8 Completion time breakdown for the LLC replication schemes evaluated
Fig. 9 L1 Cache Miss Type breakdown for the LLC replication schemes evaluated
process of creating replicas on all evictions results in the pollution of the LLC, leading to less space for useful replicas and LLC home lines. (2) The exclusive relationship between the L1 cache and the local LLC slice in VR causes a line to be always written back on an eviction even if the line is clean. Hence, in the common case where replication is useful, each hit at the LLC location effectively incurs both a read and a write at the LLC. And a write expends $1.2 \times$ more energy than a read. The first factor leads to higher network energy and completion time as well.

Benchmarks with working set that is smaller than the LLC capacity (e.g., BARNES) tend to benefit from replication. We observe from Fig. 1 that BARNES benchmark shows a high number of accesses to shared read–write data at the LLC. S-NUCA and ASR do not replicate shared read–write data, hence they do not observe any benefits for BARNES. On the other hand, RT3_3_512 and VR replicate shared read–write data and deliver $\sim 20\%$ improvement in completion time and dynamic energy over S-NUCA. This improvement in performance is due to the fact that more LLC accesses hit in the local LLC slice and a round-trip on the network is avoided. The LLC energy is higher in VR because of the reasons discussed previously. However, the network energy is substantially lower than S-NUCA and ASR, due to replication of shared read–write data. Similar trends in VR performance and energy exist in the WATER-NSQUARED, PATRICIA, BODYTRACK, FACESIM, RAYTRACE, SWAPIONS and BLACKSCHOLES benchmarks.

BLACKSCHOLES and DEDUP benchmarks exhibit a large number of LLC accesses to thread-private data with high reuse (cf. Fig. 1). ASR only replicates shared read-only cache lines and identifies these lines by using a per cache line sticky Shared bit. Hence, ASR follows the same trends as S-NUCA. Although VR places all data based on static address interleaving, its replication strategy proves effective, and significant completion time benefits are observed. RT3_3_512 also improves on S-NUCA by replicating private cache lines in the local LLC slice of the requesting core. The LLC energy increases for VR, but it is offset by substantial reductions in the network energy consumption. Dynamic energy consumption closely follows the trend seen in completion time. Same arguments hold for OCEAN_NON_CONTIGUOUS and FERRET but the network energy benefits for VR are not enough to offset the increase in LLC energy.

RADIX, FFT, LU_CONTIGUOUS, and CHOLESKY have significant number of accesses to thread-private data (cf. Fig. 1). ASR, being built on top of S-NUCA, shows the same trends as S-NUCA. VR’s replication is not as effective in these benchmarks and only slightly improves over S-NUCA. Same argument is valid for RT3_3_512 replication as well. These benchmarks benefit more from data placement than from data replication. Similar trends are observed in dynamic energy consumption results. FLUIDANIMATE and CONNECTED-COMPONENTS have working sets that do not fit in the LLC. Therefore, no significant improvement is seen with VR or RT3_3_512.

LU_NON_CONTIGUOUS shows significant number of accesses to migratory shared data. Such data exhibits exclusive use (both read and write accesses) by a unique core over a period of time before being handed to its next requester. Since ASR does not replicate shared read–write data, it cannot show benefit for benchmarks with migratory shared data. VR replicates such data almost blindly, and shows $>30\%$ improvement.
in completion time and energy consumption. Similarly, $RT3_3_512$ replicates data and shows strong gains.

S-NUCA enables efficient utilization of on-chip cache capacity. VR's replication displaces home cache lines with no sharers. This result in a higher off-chip miss rate component (cf. Fig. 9), as the evicted cache lines need to be brought back on-chip on subsequent accesses. This higher off-chip miss rate along with higher LLC energy negates the energy benefits gained through reduction in network energy. $RT3_3_512$ intelligently replicates and adapts to the cache pressure and results in off-chip miss rate degradation of only 0.4 %.

5.1.2 Data placement sensitivity

Figures 7 and 8 plot the energy and completion time breakdown for the R-NUCA, and the locality-aware scheme on top of R-NUCA, i.e., $RT3_3_512_R$. Here, benchmarks that show significant diversion from the S-NUCA trends are discussed. For example, Dedup almost exclusively accesses private data (cf. Fig. 1), and hence performs optimally with R-NUCA. Cache line replication of $RT3_3_512_R$ only slightly improves over R-NUCA, as there is not much opportunity to improve. Even then an energy improvement of $>5\%$ is achieved. The combined effects of R-NUCA data placement and intelligent data replication in $RT3_3_512_R$ improve the completion time and energy by 13 and 26 % over baseline S-NUCA system.

Radix, FFT, LU_CONTIGUOUS, CHOLESKY, and OCEAN_CONTIGUOUS have significant number of accesses to thread-private data (cf. Fig. 1). Therefore, R-NUCA shows benefit over S-NUCA, as it places private data in the local LLC slice. Similar benefits can be seen for $RT3_3_512_R$. CONNECTED- COMPONENTS has a working set that does not fit in the LLC. Therefore, both R-NUCA and $RT3_3_512_R$ fail to improve completion time significantly. However, R-NUCA’s local placement of private data helps decrease its energy consumption compared to the configurations based on S-NUCA data placement.

Blackscholes exhibits a large number of LLC accesses to private data, and a small number of LLC accesses to shared read-only data (cf. Fig. 1). Since R-NUCA places private data in its local LLC slice, one expects it to show some performance and energy benefits over S-NUCA. However, this is not the case since the data classification mechanism of R-NUCA falsely classifies private pages as shared (see Fig. 2). On the other hand, $RT3_3_512_R$ improves over R-NUCA by replicating the falsely classified private cache lines in the local LLC slice of the requesting core. $RT3_3_512_R$ shows a reduction of 5 % in completion time and $>30\%$ in energy over R-NUCA.

The migratory data in LU_NON_CONTIGUOUS is classified as shared data by R-NUCA, hence, it is placed in an LLC slice based on static address interleaving. This is why it performs on-par with S-NUCA. $RT3_3_512_R$ replicates such data and shows significant gains in both completion time (20 %) and energy (26 %). TPCC also shows improvement under R-NUCA’s data placement. The benefits are further enhanced under $RT3_3_512_R$, as it replicates useful cache lines. An improvement of $\sim10\%$ is observed in completion time and energy compared to the S-NUCA baseline. This demonstrates the applicability of the proposed locality-aware data replication scheme to server workloads. Similar trends are observed in YCSB.
R-NUCA improves completion time by 5% and energy by 6% over S-NUCA on average. In the presence of data replication, the benefits of R-NUCA data placement are diminished on average. However, some workloads demonstrate greater benefits and make a case for better data placement of data along with intelligent data replication. It should be noted that R-NUCA comes with additional overheads (discussed in Sect. 2.2.2) and hence there is a tradeoff between design complexity and derived benefits. $RT_3\_3\_512\_R$ improves by $\sim 2.5\%$ in completion time and $\sim 4\%$ in energy over $RT_3\_3\_512$.

5.1.3 Summary

R-NUCA performs better than S-NUCA and VR in workloads with mostly private data by placing such data in the local LLC slice. On the other hand, VR performs better than S-NUCA and R-NUCA in workloads with mostly shared data by replicating such data in the local LLC slice. However, it lags behind in energy as it spends more operations on each replica hit. Another reason is that it generally increases the off-chip miss rate, which hurts both performance and energy. ASR only replicates shared read-only data and hence cannot take advantage of opportunities available in replicating other types of data. Overall, ASR performs worse than VR in completion time but improves on energy consumption. Finally, $RT_3\_3\_512$ intelligently replicates all types of data and instructions, availing all opportunities to exploit data locality. $RT_3\_3\_512\_R$ shows further gains by taking advantage of the dynamic data placement policy of R-NUCA.

VR needs 1 bit per tag in the LLC to identify a cache line as replica, resulting in a storage overhead of 0.5 KB per core. On the other hand, the hardware monitoring circuits in ASR add a storage overhead of more than 18 KB per core. In comparison $RT_3\_3\_512$ pays a modest overhead of 5.44 KB per core to enable the hardware-only predictive LLC replication mechanism. The proposed locality-aware data replication protocol reduces overall energy by 14.7, 10.7, 10.5, and 16.7% and the completion time by 2.5, 6.5, 4.5, and 9.5% when compared to the previously proposed VR, ASR, R-NUCA, and S-NUCA LLC management schemes.

5.2 Tuning Limited $k-m$ locality classifier

Figure 10 plots the energy and completion time of the benchmarks with the Limited $k$ classifier when $k$ is varied as $\{1, 3, 5, 7, 64\}$. The configuration with $k = 64$ corresponds to the Complete classifier. The results are normalized to that of the Complete classifier. The benchmarks that are not shown are identical to DEDUP, i.e., the completion time and energy stays constant as $k$ varies. The experiments are run with the best $RT$ value of 3. The completion time and energy of the Limited $3$ classifier never exceeds by more than 2% the completion time and energy consumption of the Complete classifier except for STREAMCLUSTER.

With STREAMCLUSTER, the Limited $3$ classifier starts off new sharers incorrectly in non-replica mode because of the limited number of cores available for taking the majority vote. This results in increased communication between the L1 cache and LLC home location, leading to higher completion time and network energy. The Limited $5$
classifier, however, performs as well as the complete classifier, but incurs an additional 9 KB storage overhead per core when compared to the Limited3 classifier. From the previous section, we observe that the Limited3 classifier performs better than all the other baselines for STREAMCLUSTER. Hence, to trade-off the storage overhead of our classifier with the energy and completion time improvements, \( k = 3 \) is chosen as the default for the limited classifier. The Limited1 classifier is more unstable than the other classifiers. While it performs better than the Complete classifier for LU_NON_CONTIGUOUS, it performs worse for the BARNES and STREAMCLUSTER benchmarks. The better energy consumption in LU_NON_CONTIGUOUS is due to the fact that the Limited1 classifier starts off new sharers in replica mode as soon as the first sharer acquires replica status. On the other hand, the Complete classifier has to learn the mode independently for each sharer leading to a longer training period.

Figure 11 plots the energy and completion time of the benchmarks with the Limited_3−m locality classifier. The number of tracked cache lines parameter, ‘m’, is varied as \{128, 256, 512, 1024\}, while the associativity is kept at 8. The associativity of the locality table is varied as \{1, 2, 4, 8, 16\}, while the number of tracked cache lines is kept at 512. The configuration with 4096 tracked cache lines and associativity of 8 corresponds to the Limited3 locality classifier. The results are normalized to that
5.3 Sensitivity—replication threshold

Figure 12 plots the geometric mean of the energy and completion time of the benchmarks with the Limited3−512 classifier when the RT value is varied as {1, 2, 3, 4, 5, 6, 7, 8}. The results are normalized to the S-NUCA baseline. RT value of 1 shows the worst performance and energy, as it is too aggressive in replicating data. Increasing RT value results in better performance and energy consumption, since only useful cache lines are replicated. The performance delta is <1 % when RT value is swept from 2 to 8. The best completion time is observed at RT value of 3. The spread in energy consumption is ~3 % between RT values of 2 and 8. The lowest energy consumption is observed at RT value of 3. The best energy-delay product is seen at the RT value of 3, therefore, it is chosen as the default for the Limited3−512 locality classifier.

5.4 Sensitivity—LLC cache size

The advantage of RT3_3_512 increases as the LLC capacity is increased. This is because more space is available for replicas to be created. This trend holds for 64 to 256 KB per-core L2 cache size (cf. Fig. 13). However, the increase from 256 to 512 KB per-core cache size does not result in any significant improvement. This is because the active working set of most workloads fit in 256 KB per-core LLC slice. Furthermore, the useful replicas also saturate at this particular cache size. The additional capacity
As the cache size increases, each cache event becomes more costly in terms of energy. This is evident from Fig. 13 (right), where the “L2 Cache” energy component increases with increase in cache size. However, this increase in L2 energy is offset by reduction in network and off-chip energy, as more and more replicas are created. As discussed previously, at 512 KB cache slice the capacity is not utilized effectively and it does not result in more useful replicas. This can be verified from the energy numbers as well, where the network energy remains the same as the 256 KB LLC slice scenario. At the same time, the increase in size pushes the L2 energy consumption higher. The reduction in off-chip energy is not enough to offset the increase in L2 energy and thus we end up with higher overall energy.

5.5 Sensitivity—out-of-order core type

Figure 14 shows the energy and completion time results for a subset of benchmarks in an out-of-order core based multicore processor, using the RT3_3_512 classifier. The results are normalized to the S-NUCA baseline. The out-of-order processor spends available at 512 KB LLC slice is not utilized effectively and results in no appreciable improvement.
significantly lower time in compute and memory stalls due to its dynamic scheduling. The out-of-order core also executes instructions speculatively, allowing long latency operations (such as private cache misses) to be hidden without stalling the pipeline. This results in a relatively lower benefit in completion time compared to the in-order core type. This trend can be observed in FFT, BARNES, STREAMCLUSTER, and BLACKSCHOLES. However, some benchmarks still find plenty of opportunities to show similar or higher benefits, such as LU_NON_CONTIGUOUS, PATRICIA, FACESIM, and CONNECTED_COMPONENTS. On the other hand, significant gains in energy are observed in most benchmarks, such as LU_NON_CONTIGUOUS, WATER-NSQUARED, RAYTRACE, and DEDUP.

6 Conclusion

We have proposed an intelligent locality-aware data replication scheme for the last-level cache of large scale multicore processors. The cache line level reuse is profiled at runtime using a low-overhead yet highly accurate in-hardware classifier. On a set of parallel benchmarks, the locality-aware protocol reduces overall energy by 14.7, 10.7, 10.5, and 16.7 % and the completion time by 2.5, 6.5, 4.5, and 9.5 % when compared to the previously proposed Victim Replication, Adaptive Selective Replication, Reactive-NUCA and Static-NUCA LLC management schemes. The coherence complexity of the proposed protocol is almost identical to that of a traditional non-hierarchical (flat) coherence protocol. This is due to the fact that cache line replicas are only allowed to be created at the LLC slice of the requesting core. The classifier is implemented with only 1.58 % (5.44 KB) storage overhead on top of the S-NUCA baseline's cache related per-core storage.

References

1. Dreslinski RG, Fick D, Giridhar B, Kim G, Seo S, Fojtik M, Satpathy S, Lee Y, Kim D, Liu N, Wieckowski M, Chen G, Sylvester D, Blauw D, Mudge T (2013) Centip3de: a 64-core, 3d stacked near-threshold system. IEEE Micro 33(2):8–16. doi:10.1109/MM.2013.4
2. Kaul H, Anders M, Hsu S, Agarwal A, Krishnamurthy R, Borkar S (2012) Nearthreshold voltage (ntv) design: opportunities and challenges. In: Design Automation Conference. ACM, pp 1149–1154
3. Borkar S (2007) Thousand core chips: a technology perspective. In: Proceedings of the 44th annual design automation conference. ACM, New York, NY, USA, DAC’07, pp 746–749. doi:10.1145/1278480.1278667
4. Bell S, Edwards B, Amann J, Conlin R, Joyce K, Leung V, MacKay J, Reif M, Bao L, Brown J, Mattina M, Miao CC, Ramey C, Wentzlaff D, Anderson W, Berger E, Fairbanks N, Khan D, Montenegro F, Stickney J, Zook J (2008) Tile64-processor: a 64-core soc with mesh interconnect. In: IEEE international solid-state circuits conference, 2008. ISSCC 2008. Digest of Technical Papers, pp 88–598. doi:10.1109/ISSCC.2008.4523070
5. Agarwal A, Simoni R, Hennessy JL, Horowitz M (1988) An Evaluation of Directory Schemes for Cache Coherence. In: International symposium on computer architecture
6. Martin MMK, Hill MD, Sorin DJ (2012) Why on-chip cache coherence is here to stay. Commun ACM 55(7):78–89
7. Sanchez D, Kozyrakis C (2012) SCD: a scalable coherence directory with flexible sharer set encoding. In: International symposium on high-performance computer architecture
8. Zhao H, Shriraman A, Dwarkadas S (2010) SPACE: sharing pattern-based directory coherence for multicore scalability. In: International conference on parallel architectures and compilation techniques, pp 135–146
9. Zebchuk J, Srinivasan V, Qureshi MK, Moshovos A (2009) A tagless coherence directory. In: International symposium on microarchitecture
10. Eisley N, Peh LS, Shang L (2006) In-network cache coherence. In: IEEE/ACM International symposium on microarchitecture, MICRO 39:321–332. doi: 10.1109/MICRO.2006.27
11. Kurian G, Khan O, Devadas S (2013) The locality-aware adaptive cache coherence protocol. In: Proceedings of the 40th annual international symposium on computer architecture. ACM, New York, NY, USA, ISCA’13, pp 523–534. doi:10.1145/2485922.2485967
12. Conway P, Kalyanasundharam N, Donley G, Lepak K, Hughes B (2010) Cache hierarchy and memory subsystem of the amd opteron processor. Micro IEEE 30(2):16–29. doi: 10.1109/MM.2010.31
13. First the tick, now the tock: next generation intel microarchitecture (Nehalem). White Paper (2008)
14. Kim C, Burger D, Keckler SW (2002) An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: International conference on architectural support for programming languages and operating systems (ASPLOS), pp 211–222
15. Chishti Z, Powell MD, Vijaykumar TN (2005) Optimizing replication, communication, and capacity allocation in cmps. In: Proceedings of the 32Nd Annual international symposium on computer architecture, IEEE computer society, Washington, DC, USA, ISCA’05, pp 357–368. doi: 10.1109/ISCA.2005.39
16. Zhang M, Asanovic K (2005) Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: International symposium on computer architecture. doi: 10.1109/ISCA.2005.53
17. Beckmann BM, Marty MR, Wood DA (2006) Wood. Asr: adaptive selective replication for cmp caches. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE computer society, Washington, DC, USA, MICRO 39, pp 443–454. doi:10.1109/MICRO.2006.10
18. Chaudhuri M (2009) PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches. In: HPCA, pp 227–238
19. Hardavellas N, Ferdman M, Falsafi B, Ailamaki A (2009) Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In: Proceedings of the 36th annual international symposium on computer architecture (ISCA’09). ACM, New York, NY, USA, pp 184–195
20. Shi Q, Hijaz F, Khan O (2013) Towards efficient dynamic data placement in noc-based multicores. In: IEEE 31st International Conference on Computer Design (ICCD), 2013, pp 369–376. doi:10.1109/ICCD.2013.6657067
21. Merino J, Puente V, Gregorio J (2010) Esp-nuca: a low-cost adaptive non-uniform cache architecture. In: IEEE 16th international symposium on high performance computer architecture (HPCA), 2010, pp 1–10. doi:10.1109/HPCA.2010.5416641
22. Censier LM, Feautrier P (1978) A new solution to coherence problems in multicache systems. IEEE Trans Comput 27(12):1112–1118. doi:10.1109/TC.1978.1675013
23. Bell S, Edwards B, Amann J, Conlin R, Joyce K, Leung V, MacKay J, Reif M, Bao L, Brown J, Mattina M, Miao C, Ramey C, Wentzloff D, Anderson W, Berger E, Fairbanks N, Khan D, Montenegro F, Stickney J, Zook J (2008) TILE64-processor: a 64-Core SoC with mesh interconnect. In: International Solid-State Circuits Conference
24. Kurian G, Miller J, Psoa J, Eastep J, Liu J, Michel J, Kimerling L, Agarwal A (2010) ATAC: a 1000-core cache-coherent processor with on-chip optical network. In: International conference on parallel architectures and compilation techniques
25. Cho S, Jin L (2006) Managing distributed, shared L2 caches through os-level page allocation. In: Proceedings of the 39th Annual IEEE/ACM International symposium on microarchitecture, IEEE computer society, Washington, DC, USA, MICRO 39, pp 455–468. doi:10.1109/MICRO.2006.31. http://dl.acm.org/citation.cfm?doid=1194858
26. Awasthi M, Sudan K, Balasubramonian R, Carter J (2009) Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In: IEEE 15th international symposium on high performance computer architecture, 2009. HPCA 2009, pp 250–261. doi:10.1109/HPCA.2009.4798260
27. Kurian G, Devadas S, Khan O (2014) Locality-aware data replication in the last-level cache. In: IEEE 120th international symposium on high performance computer architecture (HPCA2014), 2014
28. Chang J, Sohi G (2006) Cooperative caching for chip multiprocessors. In: 33rd international symposium on computer architecture, 2006. ISCA’06, pp 264–276. doi: 10.1109/ISCA.2006.17
29. Herrero E, González J, Canal R (2010) Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In: Proceedings of the 37th Annual international symposium on computer architecture. ACM, New York, NY, USA, ISCA’10, pp 419–428. doi: 10.1145/1815961.1816018
30. Qureshi MK (2009) Adaptive spill-receive for robust high-performance caching in cmps. In: IEEE 15th international symposium on high performance computer architecture, 2009. HPCA 2009, pp 45–54. doi: 10.1109/HPCA.2009.4798236
31. Srikantaiah S, Kultursay E, Zhang T, Kandemir M, Irwin MJ, Xie Y (2011) Morphcache: a reconfigurable adaptive multi-level cache hierarchy. In: IEEE 17th international symposium on high performance computer architecture (HPCA), 2011 pp 231–242. doi: 10.1109/HPCA.2011.5749732
32. Lee H, Cho S, Childers B (2011) Cloudcache: Expanding and shrinking private caches. In: IEEE 17th international symposium on high performance computer architecture (HPCA), 2011 pp 219–230. doi: 10.1109/HPCA.2011.5749731
33. Sorin DJ, Hill MD, Wood DA (2011) A primer on memory consistency and cache coherence. Synthesis lectures in computer architecture. Morgan Claypool Publishers, San Rafael
34. Jaleel A, Borch E, Bhandaru M, Steely Jr SC, Emer J (2010) Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In: Proceedings of the 2010 43rd annual IEEE/ACM international symposium on microarchitecture, IEEE computer society, Washington, DC, USA, MICRO’43, pp 151–162. doi: 10.1109/Micro.2010.52
35. Miller JE, Kasture H, Kurian G, Gruenwald C, Beckmann N, Celio C, Eastep J, Agarwal A (2010) A distributed parallel simulator for multicore. In: 16th international symposium on high performance computer architecture (HPCA), pp 1–12
36. Daily WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufmann
37. Park S, Krishna T, Chen CH, Daya B, Chandrakasan A, Peh LS (2012) Approaching the theoretical limits of a mesh noc with a 16-node chip prototype in 45nm soi. In: Proceedings of the 49th annual design automation conference (DAC’12). ACM, New York, NY, USA, pp 398–405
38. Sun C, Chen CHO, Kurian G, Wei L, Miller J, Agarwal A, Peh LS, Stojaranovic V (2012) DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In: 6th IEEE/ACM international symposium on symposium on networks-on-chip (NoCS), pp 201–210, 9–11 May 2012
39. Li S, Ahn JH, Strong RD, Brockman JB, Tullsen DM, Jouppi NP (2009) Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: 42nd annual IEEE/ACM international symposium on microarchitecture, MICRO-42, pp 469–480, 12–16 Dec 2009
40. Thoziyoor S, Ahn JH, Monchiero M, Brockman JB, Jouppi NP (2008) A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In: 35th international symposium on computer architecture, ISCA’08, pp 51–62, 21–25 June 2008
41. Khakifirooz A, Nayfeh OM, Antoniadis D (2009) A simple semiempirical short-channel MOSFET current-voltage model continuous across all regions of operation and employing only physical parameters. IEEE Transactions Electron Devices 56(8):1674–1680
42. Wei L, Boeuf F, Skotnicki T, Wong HS (2011) Parasitic capacitances: analytical models and impact on circuit-Level performance. IEEE Transactions on Electron Devices 58(5):1361–1370
43. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The SPLASH-2 Programs: characterization and methodological considerations. In: Proceedings of 22nd annual international symposium on computer architecture, pp 24–36, 22–24 June 1995
44. Biena C, Kumar S, Singh JP, Li K (2008) The PARSEC Benchmark Suite: characterization and architectural implications. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques (PACT’08). ACM, New York, NY, USA, pp 72–81
45. Yu X, Bezerra G, Pavlo A, Devadas S, Stonebraker M (2014) Staring into the abyss: an evaluation of concurrency control with one thousand cores. Proc VLDB Endow 8(3):209–220. doi: 10.14778/2735508.2735511
46. Iqbal S, Liang Y, Grahn H (2010) ParMiBench - an open-source benchmark for embedded multiprocessor systems. Comput Archit Lett
47. DARPA UHPC Program BAA. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-37/listing.html (2010)
48. Ahmad M, Hijaz F, Shi Q, Khan O (2015) A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In: IEEE international symposium on workload characterization (IISWC), 2015 pp 44–55. doi:10.1109/IISWC.2015.11