Locality-Aware Data Replication in the Last-Level Cache

George Kurian, Srinivas Devadas
Massachusetts Institute of Technology
Cambridge, MA USA
{gkurian, devadas}@csail.mit.edu

Omer Khan
University of Connecticut
Storrs, CT USA
khan@uconn.edu

Abstract

Next generation multicores will process massive data with varying degree of locality. Harnessing on-chip data locality to optimize the utilization of cache and network resources is of fundamental importance. We propose a locality-aware selective data replication protocol for the last-level cache (LLC). Our goal is to lower memory access latency and energy by replicating only high locality cache lines in the LLC slice of the requesting core, while simultaneously keeping the off-chip miss rate low. Our approach relies on low overhead yet highly accurate in-hardware runtime classification of data locality at the cache line granularity, and only allows replication for cache lines with high reuse. Furthermore, our classifier captures the LLC pressure at the existing replica locations and adapts its replication decision accordingly.

The locality tracking mechanism is decoupled from the sharer tracking structures that cause scalability concerns in traditional coherence protocols. Moreover, the complexity of our protocol is low since no additional coherence states are created. On a set of parallel benchmarks, our protocol reduces the overall energy by 16%, 14%, 13% and 21% and the completion time by 4%, 9%, 6% and 13% when compared to the previously proposed Victim Replication, Adaptive Selective Replication, Reactive-NUCA and Static-NUCA LLC management schemes.

1 Introduction

Next generation multicore processors and applications will operate on massive data. A major challenge in future multicore processors is the data movement incurred by conventional cache hierarchies that impacts the off-chip bandwidth, on-chip memory access latency and energy consumption [7].

A large, monolithic on-chip cache does not scale beyond a small number of cores, and the only practical option is to physically distribute memory in pieces so that every core is near some portion of the cache [5]. 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 [3, 22]. Many recent proposals have addressed directory scalability in single-chip multicores using sharer compression techniques or limited directories [24, 31, 29, 12]. 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 [19]. This leads to increased network traffic and request rate to the last-level cache. Since on-chip wires are not scaling at the same rate as transistors, data movement not only impacts memory access latency, but also consumes extra power due to the energy consumption of network and cache resources [16].

Last-level cache (LLC) organizations offer trade-offs between on-chip data locality and off-chip miss rate. While private LLC organizations (e.g., [10]) 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., [1]), on the other hand, lead to non-uniform cache access (NUCA) [17] 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 that attempt to combine the good characteristics of private and shared LLC organizations [9, 30, 4, 13], 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 significantly complicate coherence and do not scale to large core counts (see Section 5 for details).

We propose a data replication mechanism for the LLC that retains the on-chip cache utilization of the shared LLC while intelligently replicating cache lines close to the requesting cores so as to maximize data locality. To achieve this goal, we propose a low-overhead yet highly accurate in-hardware locality classifier at the cache line granularity that *only* allows the replication of cache lines with high reuse.

1.1 Motivation

The utility of data replication at 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

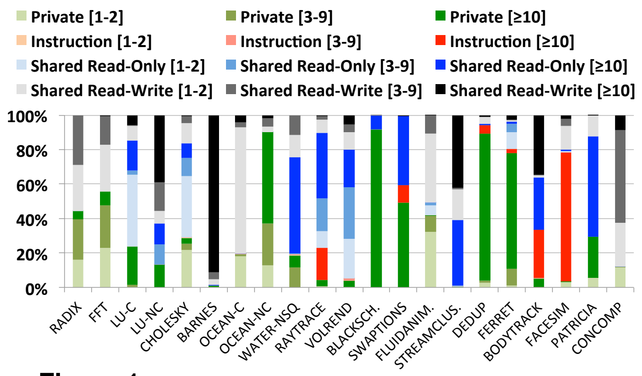

Figure 1. Distribution of instructions, private data, shared read-only data, and shared read-write data accesses to the LLC as a function of *run-length*. The classification is done at the cache line granularity.

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, over 90% of the 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.

Hence, the replication decision should not depend on the type of data, but rather on its locality. Instructions and shared-data (both read-only and read-write) can be replicated if they demonstrate good reuse. It is also important to adapt the replication decision at runtime in case the reuse of data changes during an application's execution.

1.2 Proposed Idea

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. Our 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, our 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. Our 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. Allows coherence complexity almost identical to that of a traditional non-hierarchical (flat) coherence protocol since replicas are only allowed to be placed at the LLC slice of the requesting core. The additional coherence complexity only arises within a core when the LLC slice is searched on an L1 cache miss, or when a cache line in the core's local cache hierarchy is evicted/invalidated.

2 Locality-Aware LLC Data Replication

2.1 Baseline System

The baseline system is a tiled multicore with an electrical 2-D mesh interconnection network as shown in Figure 2. 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. The 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 [8, 5]) 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 [20]. Some cores have a connection to a memory controller as well.

Even though our protocol can be built on top of both Static-NUCA and Dynamic-NUCA configurations [17, 13], we use Reactive-NUCA's data placement and migration mechanisms to manage the LLC [13]. Private data is placed at the LLC slice of the requesting core and shared data is address interleaved across all LLC slices. Reactive-NUCA also proposed replicating instructions at a cluster-level (e.g., 4 cores) using a rotational interleaving mechanism. However, we do not use this mechanism, but instead build a locality-aware LLC data replication scheme for all types of cache lines.

2.2 Protocol Operation

The four essential components of data replication 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. We first define a few terms to facilitate describing our protocol.

1. **Home Location**: The core where all requests for a cache line are serialized for maintaining coherence.
2. **Replica Sharer**: A core that is granted a replica of a cache line in its LLC slice.
3. **Non-Replica Sharer**: A core that is *NOT* granted a replica of a cache line in its LLC slice.
4. **Replica Reuse**: The number of times an LLC replica is accessed before it is invalidated or evicted.
5. **Home Reuse**: The number of times a cache line is accessed at the LLC slice in its home location before a conflicting write or eviction.
6. **Replication Threshold (RT)**: The reuse above or equal to which a replica is created.

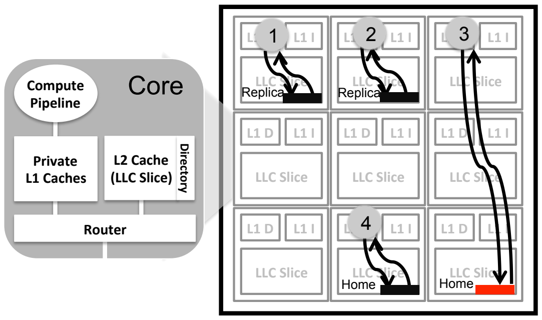

Figure 2. ① – ④ are mockup requests showing the *locality-aware LLC replication* protocol. The *black* data block has high reuse and a local LLC replica is allowed that services requests from ① and ②. The low-reuse *red* data block is not allowed to be replicated at the LLC, and the request from ③ that 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 (e.g., ④).

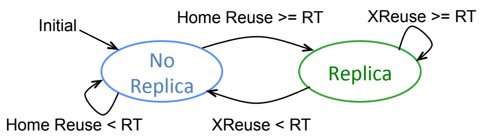

Figure 3. Each directory entry is extended with *replication mode* bits to classify the usefulness of LLC replication. Each cache line is initialized to non-replica mode with respect to all cores. Based on the reuse counters (at the home as well as the replica location) and the parameter RT, the cores are transitioned between replica and non-replica modes. Here XReuse is (Replica + Home) Reuse on an invalidation and Replica Reuse on an eviction.

Note that for a cache line, one core can be a replica sharer while another can be a non-replica sharer. Our 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 Figure 3). Let us understand the handling of read requests, write requests, evictions, invalidations and downgrades as well as cache replacement policies under this protocol.

2.2.1 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 Figure 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

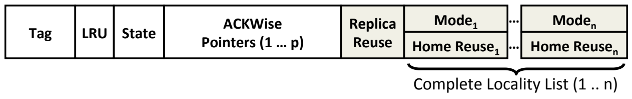

Figure 4. ACKwise$_p$-*Complete* locality classifier LLC tag entry. It contains the tag, LRU bits and directory entry. The directory entry contains the state, ACKwise$_p$ pointers, a *Replica reuse* counter as well as *Replication mode* bits and *Home reuse* counters for every core in the system.

directory entry is augmented with additional bits as shown in Figure 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, we will present a cost-efficient classifier implementation in Section 2.2.5. 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.

2.2.2 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 [25]. The acknowledgements received are processed as described in Section 2.2.3. After all such acknowledgements 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.

2.2.3 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 acknowledgement is sent to the LLC home location. In addition, if a valid LLC replica exists, the replica reuse counter is communicated back with the acknowledgement. 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 Figure 3). 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 acknowledgement 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 acknowledgement 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 acknowledgement 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 acknowledgement 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 (2) our LLC replacement policy prioritizes retaining cache lines that have L1 cache copies.

2.2.4 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 [15]. However, this incurs additional network traffic.

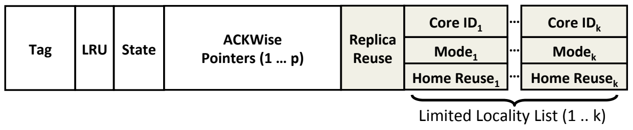

Figure 5. ACKwise$_p$-*Limited*$_k$ locality classifier LLC tag entry. It contains the tag, LRU bits and directory entry. The directory entry contains the state, ACKwise$_p$ pointers, a *Replica reuse* counter as well as the Limited$_k$ classifier. The Limited$_k$ classifier contains a *Replication mode* 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.

Our 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 and outperforms the LRU policy (cf. Section 4.2).

2.2.5 Limited Locality Classifier Optimization

The classifier described earlier which keeps track of locality information for all the cores in the directory entry is termed the *Complete* locality classifier. It has a storage overhead of 30% (calculated in Section 2.4) at 64 cores and over 5× at 1024 cores. In order to mitigate this overhead, we develop a classifier 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 implemented efficiently on top of other scalable directory organizations. We now describe the working of the limited locality classifier.

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 Section 4.3, we evaluate the storage and accuracy tradeoffs for the *Limited*$_k$ classifier. Based on our observations, we pick the *Limited*$_3$ classifier.

2.3 Discussion

2.3.1 Replica Creation Strategy

In the protocol described earlier, replicas are 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 the *Exclusive* or *Modified* state. Benchmarks that exhibit both the above access patterns are observed in our evaluation (cf. Section 4.1).

2.3.2 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 optimizations, 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 compared our 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 was less than 1%. Hence, in the interest of avoiding the additional complexity, the LLC replica is always looked up for the above coherence requests.

2.3.3 Classifier Organization

The 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.

The storage overhead for the in-cache organization is calculated in Section 2.4. The performance and energy overhead for this organization is small because: (1) The classifier lookup incurs a relatively small energy and latency penalty when compared to the data array lookup of the LLC slice and communication over the network (justified in our results). (2) Only a single tag lookup is needed for accessing the classifier and LLC data. In a sparse organization, a separate lookup is required for the classifier and the LLC data. Even though these lookups could be performed in parallel with no latency overhead, the energy expended to lookup two CAM structures needs to be paid.

2.3.4 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.

We explored the clustering under our protocol after making the appropriate changes. The changes include (1) blocking at the replica location (the core in the cluster where a replica could be found) before forwarding the request to the home location so that multiple cores on the same cluster do not have outstanding requests to the LLC home location, (2) additional coherence message types for communication between the requester, replica and home cores, and (3) hierarchical invalidation and downgrade of the replica and the L1 caches that it tracks. We do not explain all the details here to save space. Cluster-level replication was not found to be beneficial in the evaluated 64-core system, for the following reasons (see Section 4.4 for details).

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 increased our design and verification time significantly.

2.4 Overheads

2.4.1 Storage

The locality-aware protocol requires extra bits at the LLC tag arrays to track locality information. Each LLC directory entry requires 2 bits for the replica reuse counter (assuming an optimal *RT* of 3). The *Limited*$_3$ 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 *Limited*$_3$ classifier requires an additional 27 (= 3 × 9) bits of storage per LLC directory entry. The *Complete* classifier, on the other hand, requires 192 (= 64 × 3) bits of storage.

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 $\frac{2\times256}{64\times8} = 1KB$. The storage overhead of the *Limited*$_3$ classifier is $\frac{27\times256}{64\times8} = 13.5KB$. For the complete classifier, it is $\frac{192\times256}{64\times8} = 96KB$. Now, the storage overhead of the *ACKwise*$_4$ protocol in this processor is $12KB$ (assuming 6 bits per *ACKwise* pointer) and that for a *Full Map* protocol is $32KB$. Adding up all the storage components, the ***Limited*$_3$ classifier with *ACKwise*$_4$ protocol uses slightly less storage than the *Full Map* protocol and 4.5% more storage than the baseline *ACKwise*$_4$ protocol.** The *Complete* classifier with the *ACKwise*$_4$ protocol uses 30% more storage than the baseline *ACKwise*$_4$ protocol.

2.4.2 LLC Tag & Directory 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. This additional expense is accounted in our evaluation.

2.4.3 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.

Architectural Parameter	**Value**
Number of Cores	64 @ 1 GHz
Compute Pipeline per Core	In-Order, Single-Issue
Processor Word Size	64 bits
Memory Subsystem	
L1-I Cache per core	16 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	32 KB, 4-way Assoc., 1 cycle
L2 Cache (LLC) per core	256 KB, 8-way Assoc., 2 cycle tag, 4 cycle data Inclusive, R-NUCA [13]
Directory Protocol	Invalidation-based MESI, ACKwise$_4$ [20]
DRAM Num., Bandwidth, Latency	8 controllers, 5 GBps/cntlr, 75 ns latency
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Flit Width	64 bits
Header (Src, Dest, Addr, MsgType)	1 flit
Cache Line Length	8 flits (512 bits)
Locality-Aware LLC Data Replication	
Replication Threshold	*RT* = 3
Classifier	Limited$_3$

Table 1. Architectural parameters used for evaluation

3 Evaluation Methodology

We evaluate a 64-core multicore. The default architectural parameters used for evaluation are shown in Table 1.

3.1 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 [23] multicore simulator. All the mechanisms and protocol overheads discussed in Section 2 are modeled. The Graphite simulator requires the memory system (including the cache hierarchy) to be functionally correct to complete simulation. This is a good test that all our cache coherence protocols are working correctly given that we have run 21 benchmarks to completion.

The electrical mesh interconnection network uses XY routing. Since modern network-on-chip routers are pipelined [11], and 2- or even 1-cycle per hop router latencies [18] have been demonstrated, we model a 2-cycle per hop delay. In addition to the fixed per-hop latency, network link contention delays are also modeled.

3.2 Energy Models

For dynamic energy evaluations of on-chip electrical network routers and links, we use the DSENT [26] tool. The dynamic energy estimates for the L1-I, L1-D and L2 (with integrated directory) caches as well as DRAM are obtained using McPAT/CACTI [21, 27]. The energy evaluation is performed at the 11 nm technology node to account for future scaling trends. As clock frequencies are relatively slow, high threshold transistors are assumed for lower leakage.

3.3 Baseline LLC Management Schemes

We model four baseline multicore systems that assume private L1 caches managed using the *ACKwise*$_4$ protocol.

1. The **Static-NUCA** baseline address interleaves all cache lines among the LLC slices.
2. The **Reactive-NUCA** [13] baseline 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 in a single LLC slice.
3. The **Victim Replication (VR)** [30] baseline 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, a replica itself or has no sharers in the L1 cache.
4. The **Adaptive Selective Replication (ASR)** [4] 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. We do not model the hardware monitoring circuits or the dynamic adaptation of replication levels. Instead, we run ASR at five different replication levels (0, 0.25, 0.5, 0.75, 1) and choose the one with the lowest energy-delay product for each benchmark.

3.4 Evaluation Metrics

Each multithreaded benchmark is run to completion using the input sets from Table 2. We measure the energy consumption of the memory system including the on-chip caches, DRAM and the network. We also measure the completion time, 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.

Application	**Problem Size**
SPLASH-2 [28]	
RADIX	4M integers, radix 1024
FFT	4M complex data points
LU-C, LU-NC	1024×1024 matrix
OCEAN-C	2050×2050 ocean
OCEAN-NC	1026×1026 ocean
CHOLESKY	tk29.O
BARNES	64K particles
WATER-NSQUARED	512 molecules
RAYTRACE	car
VOLREND	head
PARSEC [6]	
BLACKSCHOLES	65,536 options
SWAPTIONS	64 swaptions, 20,000 sims.
STREAMCLUSTER	8192 points per block, 1 block
DEDUP	31 MB data
FERRET	256 queries, 34,973 images
BODYTRACK	4 frames, 4000 particles
FACESIM	1 frame, 372,126 tetrahedrons
FLUIDANIMATE	5 frames, 300,000 particles
Others: Parallel MI-Bench [14], UHPC Graph benchmark [2]	
PATRICIA	5000 IP address queries
CONNECTED-COMPONENTS	Graph with 2^{18} nodes

Table 2. Problem sizes for our parallel benchmarks.

One of the important memory system metrics we track to evaluate our protocol are the various cache miss types. 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 Results

4.1 Comparison of Replication Schemes

Figures 6 and 7 plot the energy and completion time breakdown for the replication schemes evaluated. The *RT-1*, *RT-3* and *RT-8* bars correspond to the locality-aware scheme with replication thresholds of 1, 3 and 8 respectively.

The energy and completion time trends can be understood based on the following 3 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

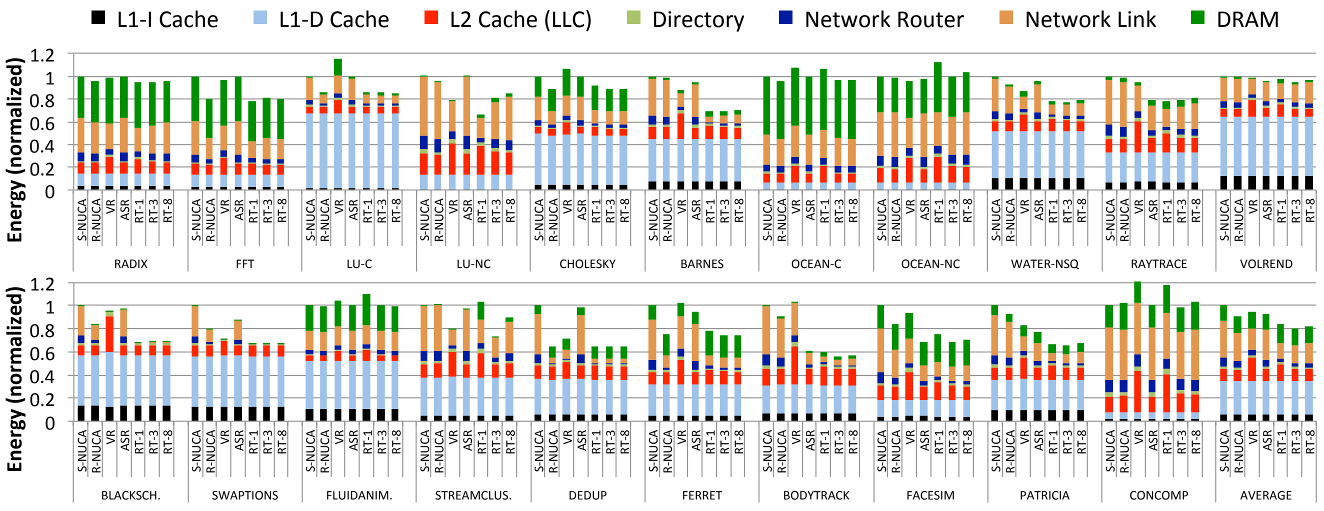

Figure 6. Energy breakdown for the LLC replication schemes evaluated. Results are normalized to that of S-NUCA. Note that *Average* and not *Geometric-Mean* is plotted here.

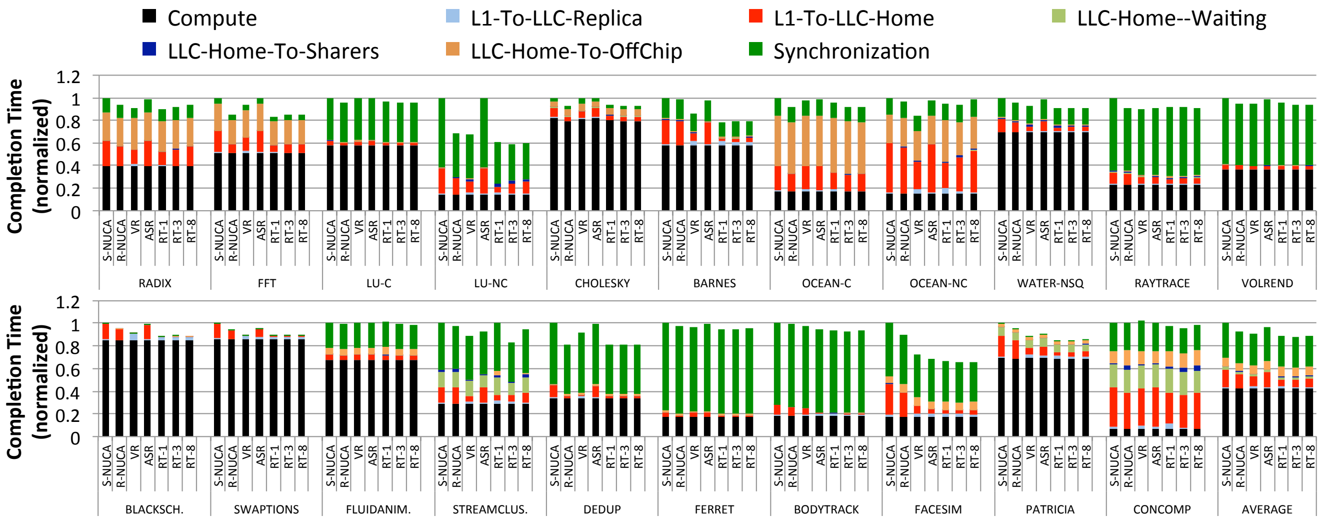

Figure 7. Completion Time breakdown for the LLC replication schemes evaluated. Results are normalized to that of S-NUCA. Note that *Average* and not *Geometric-Mean* is plotted here.

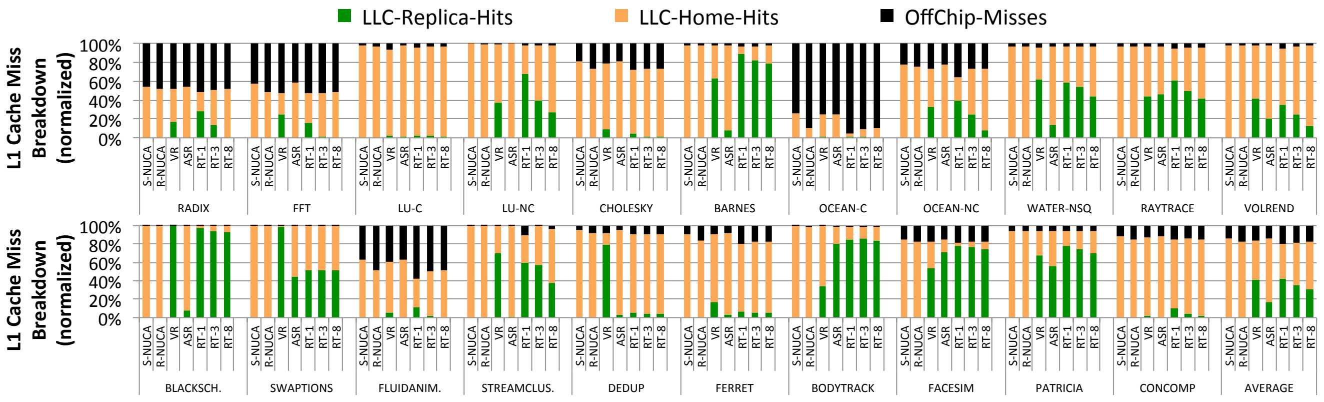

Figure 8. L1 Cache Miss Type breakdown for the LLC replication schemes evaluated.

at the LLC, and (3) working set size of the benchmark. Figure 8, which plots how L1 cache misses are handled by the LLC is also instrumental in understanding these trends.

Many benchmarks (e.g., BARNES) have a working set that fits within the LLC even if replication is done on every L1 cache miss. Hence, all locality-aware schemes (*RT-1*, *RT-3* and *RT-8*) perform well both in energy and performance. In our experiments, we observe that BARNES exhibits a high reuse of cache lines at the LLC through accesses directed at shared read-write data. S-NUCA, R-NUCA and ASR do not replicate shared read-write data and hence do not observe any benefits with BARNES.

VR observes some benefits since it locally replicates read-write data. However, it exhibits higher L2 cache (LLC) energy than the other schemes for two reasons. (1) Its (*almost*) blind 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. This is evident from lower replica hit rate for VR when compared to our locality-aware protocol. (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. This is because a replica hit always causes the line in the LLC slice to be invalidated and inserted into the L1 cache. 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. This explains why VR performs worse than the locality-aware protocol. Similar trends in VR performance and energy exist in the WATER-NSQ, PATRICIA, BODYTRACK, FACESIM, STREAMCLUSTER and BLACKSCHOLES benchmarks.

BODYTRACK and FACESIM are similar to BARNES except that their LLC accesses have a greater fraction of instructions and/or shared read-only data. The accesses to shared read-write data are again mostly reads with only a few writes. R-NUCA shows significant benefits since it replicates instructions. ASR shows even higher energy and performance benefits since it replicates both instructions and shared read-only data. The locality-aware protocol also shows the same benefits since it replicates all classes of cache lines, provided they exhibit reuse in accordance with the replication thresholds. VR shows higher LLC energy for the same reasons as in BARNES. ASR and our locality-aware protocol allow the LLC slice at the replica location to be inclusive of the L1 cache, and hence, do not have the same drawback as VR. VR, however, does not have a performance overhead because the evictions are not on the critical path of the processor pipeline.

Note that BODYTRACK, FACESIM and RAYTRACE are the only three among the evaluated benchmarks that have a significant L1-I cache MPKI (misses per thousand instructions). All other benchmarks have an extremely low L1-I MPKI (< 0.5) and hence R-NUCA's replication mechanism is not effective in most cases. Even in the above 3 benchmarks, R-NUCA does not place instructions in the local LLC slice but replicates them at a cluster level, hence the serialization delays to transfer the cache lines over the network still need to be paid.

BLACKSCHOLES, on the other hand, exhibits a large number of LLC accesses to private data and a small number to shared read-only data. Since R-NUCA places private data in its local LLC slice, it obtains performance and energy improvements over S-NUCA. However, the improvements obtained are limited since false sharing is exhibited at the page-level, i.e., multiple cores privately access non-overlapping cache lines in a page. Since R-NUCA's classification mechanism operates at a page-level, it is not able to locally place all truly private lines. The locality-aware protocol obtains improvements over R-NUCA by replicating these cache lines. 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. DEDUP almost exclusively accesses private data (without any false sharing) and hence, performs optimally with R-NUCA.

Benchmarks such as RADIX, FFT, LU-C, OCEAN-C, FLUIDANIMATE and CONCOMP do not benefit from replication and hence the baseline R-NUCA performs optimally. R-NUCA does better than S-NUCA because these benchmarks have significant accesses to thread-private data. ASR, being built on top of S-NUCA, shows the same trends as S-NUCA. VR, on the other hand, shows higher LLC energy because of the same reasons outlined earlier. VR's replication of private data in its local LLC slice is also not as effective as R-NUCA's policy of placing private data locally, especially in OCEAN-C, FLUIDANIMATE and CONCOMP whose working sets do not fit in the LLC.

The locality-aware protocol benefits from the optimizations in R-NUCA and tracks its performance and energy consumption. For the locality-aware protocol, an *RT* of 3 dominates an *RT* of 1 in FLUIDANIMATE because it demonstrates significant off-chip miss rates (as evident from its energy and completion time breakdowns) and hence, it is essential to balance on-chip locality with off-chip miss rate to achieve the best energy consumption and performance. While an *RT* of 1 replicates on every L1 cache miss, an *RT* of 3 replicates only if a reuse ≥ 3 is demonstrated. Using an *RT* of 3 reduces the off-chip miss rate in FLUIDANIMATE and provides the best performance and energy consumption. Using an *RT* of 3 also provides the maximum benefit in benchmarks such as OCEAN-C and OCEAN-NC.

As *RT* increases, the off-chip miss rate decreases but the LLC hit latency increases. For example, with an *RT* of 8, STREAMCLUSTER shows an increased completion time and network energy caused by repeated fetches of the cache line over the network. An *RT* of 3 would bring the cache line into the local LLC slice sooner, avoiding the unnecessary network traffic and its performance and energy impact. This is evident from the smaller "L1-To-LLC-Home" component of the completion time breakdown graph and the higher number of replica hits when using an *RT* of 3. We explored all values of *RT* between 1 & 8 and found that they provide no additional insight beyond the data points discussed here.

LU-NC exhibits 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 accessor. Replication of migratory shared data requires creation of a replica in an *Exclusive* coherence state. The locality-aware protocol makes LLC replicas for such data when sufficient reuse is detected. Since ASR does not replicate shared read-write data, it cannot show benefit for benchmarks with migratory shared data. VR, on the other hand, (almost) blindly replicates on all L1 evictions and performs on par with the locality-aware protocol for LU-NC.

To summarize, the locality-aware protocol provides better energy consumption and performance than the other LLC data management schemes. It is important to balance the on-chip data locality and off-chip miss rate and overall, an *RT* of 3 achieves the best trade-off. It is also important to replicate all types of data and selective replication of certain types of data by R-NUCA (instructions) and ASR (instructions, shared read-only data) leads to sub-optimal energy and performance. Overall, the locality-aware protocol has a 16%, 14%, 13% and 21% lower energy and a 4%, 9%, 6% and 13% lower completion time compared to VR, ASR, R-NUCA and S-NUCA respectively.

4.2 LLC Replacement Policy

As discussed earlier in Section 2.2.4, we propose to use a modified-LRU replacement policy for the LLC. It first selects cache lines with the least number of sharers and then chooses the least recently used among them. This replacement policy improves energy consumption over the traditional LRU policy by 15% and 5%, and lowers completion time by 5% and 2% in the BLACKSCHOLES and FACESIM benchmarks respectively. In all other benchmarks this replacement policy tracks the LRU policy.

4.3 Limited Locality Classifier

Figure 9 plots the energy and completion time of the benchmarks with the Limited$_k$ classifier when k is varied as (1, 3, 5, 7, 64). 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 stay constant as *k* varies. The experiments are run with the best *RT* value of 3 obtained in Section 4.1. We observe that 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,

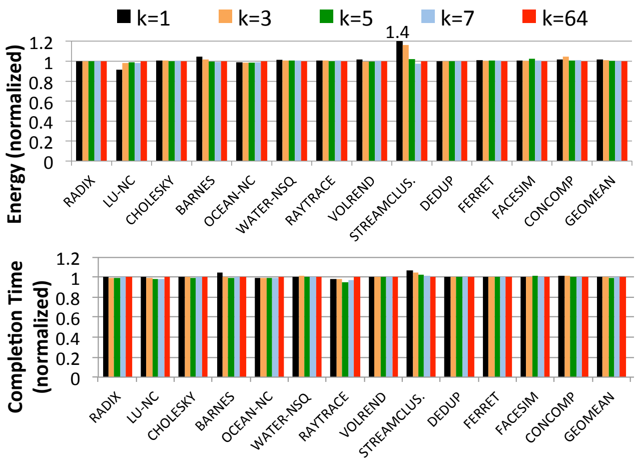

Figure 9. Energy and Completion Time for the Limited$_k$ classifier as a function of number of tracked sharers (k). The results are normalized to that of the Complete (= Limited$_{64}$) classifier.

however, performs as well as the complete classifier, but incurs an additional $9KB$ storage overhead per core when compared to the Limited$_3$ classifier. From the previous section, we observe that the Limited$_3$ 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, we chose $k = 3$ as the default for the limited classifier.

The Limited$_1$ classifier is more unstable than the other classifiers. While it performs better than the Complete classifier for LU-NC, it performs worse for the BARNES and STREAMCLUSTER benchmarks. The better energy consumption in LU-NC is due to the fact that the Limited$_1$ 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.

4.4 Cluster Size Sensitivity Analysis

Figure 10 plots the energy and completion time for the locality-aware protocol when run using different cluster sizes. The experiment is run with the optimal RT of 3. Using a cluster size of 1 proved to be optimal. This is due to several reasons.

In benchmarks such as BARNES, STREAMCLUSTER and BODYTRACK, where the working set fits within the LLC even with replication, moving from a cluster size of 1 to 64 reduced data locality without improving the LLC miss rate, thereby hurting energy and performance.

Benchmarks like RAYTRACE that contain a significant amount of read-only data with low degrees of sharing also do not benefit since employing a cluster-based approach reduces data locality without improving LLC miss rate. A cluster-based approach can be useful to explore the trade-off between LLC data locality and miss rate only if the data is shared by mostly all cores within a cluster.

In benchmarks such as RADIX and FLUIDANIMATE that

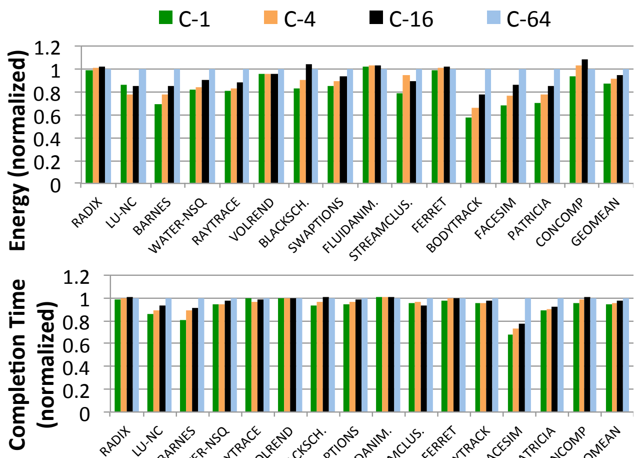

Figure 10. Energy and Completion Time at cluster sizes of 1, 4, 16 and 64 with the locality-aware data replication protocol. A cluster size of 64 is the same as R-NUCA except that it does not even replicate instructions.

show no usefulness for replication, applying the locality-aware protocol bypasses all replication mechanisms and hence, employing higher cluster sizes would not be anymore useful than employing a lower cluster size. Intelligently deciding which cache lines to replicate using an *RT* of 3 was enough to prevent any overheads of replication.

The above reasons along with the added coherence complexity of clustering (as discussed in Section 2.3.4) motivate using a cluster size of 1, at least in the 64-core multicore target that we evaluate.

5 Related Work

CMP-NuRAPID [9] uses the idea of *controlled replication* to place data so as to optimize the distance to the LLC bank holding the data. The idea is to decouple the tag and data arrays and maintain private per-core tag arrays and a shared data array. The shared data array is divided into multiple banks based on distance from each core. A cache line is replicated in the cache bank closest to the requesting core on its second access, the second access being detected using the entry in the tag array. This scheme uses controlled replication to force just one cache line copy for read-write shared data, and forces write-through to maintain coherence for such lines using an additional *Communication (C)* coherence state. The decision to replicate based on the second access for read-only shared data is also limited since it does not take into account the LLC pressure due to replication.

CMP-NuRAPID does not scale with the number of cores since each private per-core tag array potentially has to store pointers to the entire data array. Results reported in [9] indicate that the private tag array size should only be twice the size of the per-cache bank tag array but this is because only a 4-core CMP is evaluated. 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.

Reactive-NUCA [13] replicates instructions in one LLC slice per cluster of cores. Shared data is *never* replicated and is always placed at a single LLC slice. The one size fits all approach to handling instructions does not work for applications with heterogeneous instructions. In addition, our evaluation has shown significant opportunities for improvement through replication of shared read-only and shared read-mostly data.

Victim Replication (VR) [30] starts out with a private L1 shared L2 (LLC) organization 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, a replica itself or has no sharers in the L1 cache. VR attempts to combine the low hit latency of the private LLC with the low off-chip miss rate of the shared LLC. The main drawback of this approach is that replicas are created without paying attention to the LLC cache pressure since a replacement candidate with no sharers exists in the LLC with high probability. Furthermore, the *static decision policy to blindly replicate* (without paying attention to reuse) private and shared data in the local LLC slice could lead to increased LLC misses.

Adaptive Selective Replication (ASR) [4] 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 using an additional per-cache-line shared bit. ASR pays attention to the LLC pressure by basing its replication decision on a probability that is obtained *dynamically*. The probability value is picked from discrete replication levels on a per-core basis. A higher replication level indicates that L1 eviction victims are replicated with a higher probability. The replication levels are decided using hardware monitoring circuits that quantify the replication effectiveness based on the benefit (lower L2 hit latency) and cost (higher L2 miss latency) of replication.

A drawback of ASR is that the replication decision is based on coarse-grain information. The per-core counters do not capture the replication usefulness for cache lines with time-varying degree of reuse. Furthermore, the restriction to replicate shared read-only data is limiting because it does not exploit reuse for other types of data. ASR assumes an 8-core processor and the LLC lookup mechanism that needs to search all LLC slices does not scale to large core counts (although we model an efficient directory-based protocol in our evaluation).

6 Conclusion

We have proposed an intelligent locality-aware data replication scheme for the last-level cache. The locality is profiled at runtime using a low-overhead yet highly accurate in-hardware cache-line-level classifier. On a set of parallel benchmarks, our locality-aware protocol reduces the overall energy by 16%, 14%, 13% and 21% and the completion time by 4%, 9%, 6% and 13% when compared to the previously proposed Victim Replication, Adaptive Selective Replication, Reactive-NUCA and Static-NUCA LLC management schemes. The coherence complexity of our protocol is almost identical to that of a traditional non-hierarchical (flat) coherence protocol since replicas are only allowed to be created at the LLC slice of the requesting core. Our classifier is implemented with $14.5KB$ storage overhead per $256KB$ LLC slice.

References

[1] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). White Paper, 2008.

[2] DARPA UHPC Program (DARPA-BAA-10-37), March 2010.

[3] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *International Symposium on Computer Architecture*, 1988.

[4] B. M. Beckmann, M. R. Marty, and D. A. Wood. Asr: Adaptive selective replication for cmp caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 443–454, 2006.

[5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *IEEE Solid-State Circuits Conference*, feb. 2008.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[7] S. Borkar. Thousand core chips: a technology perspective. In *Design Automation Conference*, 2007.

[8] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.*, 27(12):1112–1118, Dec. 1978.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 357–368, 2005.

[10] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2), Mar. 2010.

[11] W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2003.

[12] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 321–332, 2006.

[13] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *International Symposium on Computer Architecture*, 2009.

[14] S. Iqbal, Y. Liang, and H. Grahn. Parmibench - an open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters*, 9(2):45 –48, feb. 2010.

[15] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *International Symposium on Microarchitecture*, 2010.

[16] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar. Near-threshold voltage (NTV) design - opportunities and challenges. In *Design Automation Conference*, 2012.

[17] C. Kim, D. Burger, and S. W. Keckler. 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*, 2002.

[18] A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha. A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator in 65nm cmos. In *International Conference on Computer Design*, 2007.

[19] G. Kurian, O. Khan, and S. Devadas. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 523–534, New York, NY, USA, 2013. ACM.

[20] G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *International Symposium on Microarchitecture*, 2009.

[22] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7), 2012.

[23] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *International Symposium on High-Performance Computer Architecture*, 2010.

[24] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *International Symp. on High-Performance Computer Architecture*, 2012.

[25] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures in Computer Architecture, Morgan Claypool Publishers, 2011.

[26] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *International Symposium on Networks-on-Chip*, 2012.

[27] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *International Symposium on Computer Architecture*, pages 51–62, 2008.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.

[29] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *International Symposium on Microarchitecture*, 2009.

[30] M. Zhang and K. Asanović. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *International Symp. on Computer Architecture*, 2005.

[31] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: sharing pattern-based directory coherence for multicore scalability. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 135–146, 2010.