The Locality-Aware Adaptive Cache Coherence Protocol

George Kurian
Massachusetts Institute of Technology
Cambridge, MA USA
gkurian@csail.mit.edu

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

Srinivas Devadas
Massachusetts Institute of Technology
Cambridge, MA USA
devadas@mit.edu

ABSTRACT

Next generation multicore applications will process massive amounts of data with significant sharing. Data movement and management impacts memory access latency and consumes power. Therefore, harnessing data locality is of fundamental importance in future processors. We propose a scalable, efficient shared memory cache coherence protocol that enables seamless adaptation between private and logically shared caching of on-chip data at the fine granularity of cache lines. Our data-centric approach relies on in-hardware yet low-overhead runtime profiling of the locality of each cache line and only allows private caching for data blocks with high spatio-temporal locality. This allows us to better exploit the private caches and enable low-latency, low-energy memory access, while retaining the convenience of shared memory. On a set of parallel benchmarks, our low-overhead locality-aware mechanisms reduce the overall energy by 25% and completion time by 15% in an NoC-based multicore with the Reactive-NUCA on-chip cache organization and the ACKwise limited directory-based coherence protocol.

Categories and Subject Descriptors

C.1.2.g [**Processor Architectures**]: [Parallel processors]; B.3.2.g [**Memory Structures**]: [Shared memory]

General Terms

Design, Performance

Keywords

Cache Coherence, Multicore

This work was funded by the U.S. Government under the DARPA UHPC program, and a faculty startup grant from the University of Connecticut. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

ISCA '13 Tel-Aviv, Israel

1. INTRODUCTION

In the era of multicores, programmers now need to invest more effort in designing software capable of exploiting multicore parallelism. To reduce memory access latency and power consumption, a programmer can manually orchestrate communication and computation or adopt the familiar programming paradigm of shared memory. But will current shared memory architectures scale to many cores? This paper addresses the question of how to enable *low-latency, low-energy memory access while retaining the convenience of shared memory.*

Current semiconductor trends project the advent of single-chip multicores dealing with data at an unprecedented scale. Hence, in future processors, the main bottleneck shifts from computation capabilities to data management, including on-chip memory and communication. Memory scalability is critically constrained by off-chip bandwidth, on-chip latency and energy consumption [11]. Memory access latency and energy consumption are now first-order design constraints.

A large, monolithic physically-shared on-chip cache does not scale beyond a small number of cores, and the only practical option is to physically distribute it in pieces so that every core is near some portion of the cache [2]. In theory this provides a large amount of aggregate cache capacity and fast private access for each core. Unfortunately, it is difficult to manage distributed private caches effectively as they require architectural support for cache coherence and consistency.

Popular directory-based protocols enable fast local caching to exploit data locality, but scale poorly with increasing core counts [16]. Many recent proposals (e.g., Tagless Directory [23], SPATL [25], SCD [19], and ACKwise [13]) have addressed directory scalability in single-chip multicores using either sharer compression techniques or limited directories that require complex on-chip network capabilities [9]. But they still suffer from a major drawback, that is the private caches are left unmanaged. A request for data allocates and replicates a data block in the private cache hierarchy even if the data has no spatial or temporal locality. This leads to cache pollution since such low locality data can displace more frequently used data, or suffer from expensive communication due to the coherence protocol. Since on-chip wires are not scaling at the same rate as transistors [11], unnecessary data movement and replication not only impacts latency, but also consumes extra power due to wasteful energy consumption of network and cache resources [16].

A popular option is to organize the last-level cache as logically shared, leading to non-uniform cache access (NUCA) [12]. Although the NUCA configuration yields better on-

chip cache utilization, exploiting locality using low-latency private caching becomes even more challenging (as memory latency is now also sensitive to data placement). To address this problem, coarse-grain data migration and restrictive replication schemes have been proposed (e.g., Reactive-NUCA [5]); however, these mechanisms typically assume private low-level caches and still require directories that leave the private caches unmanaged.

In this paper we propose a *Locality-Aware Adaptive Coherence* protocol to better manage the private caches in shared memory multicores. When a core makes a memory request that misses the private cache, the coherence protocol either brings the entire cache line using a traditional directory protocol, or just accesses the requested word at the shared cache location. This decision is based on the *spatio-temporal locality* of a particular data block. Our approach relies on runtime profiling of the locality of each cache line and only allows private caching for data blocks with high spatio-temporal locality. We propose a low-overhead (18KB storage per core) yet highly accurate predictive mechanism to track the locality of cache lines in hardware. This locality tracking mechanism is decoupled from the sharer tracking structures that cause scalability concerns in cache coherence protocols.

We implement our protocol on top of an existing Private-L1 Shared-L2 cache organization, where the shared L2 cache is physically distributed in slices throughout the chip and managed using Reactive-NUCA's [5] data placement and migration policy. We choose the ACKwise limited directory protocol because it has been shown to scale to large core counts [13].

Our *Locality-Aware Adaptive Coherence* protocol is advantageous because it:

1. Better exploits on-chip private caches by intelligently controlling data caching and replication.
2. Enables lower memory access latency by trading off unnecessary cache evictions or expensive invalidations with much cheaper word accesses.
3. Lowers energy consumption by better utilizing on-chip network and cache resources.

2. BACKGROUND

2.1 Related Work

Previous research on cache organizations in multicore processors mostly focused on the last-level cache (LLC). Proposals have been made to organize the LLC as private, shared or a combination of both. All other cache levels have traditionally been organized as private to a core [4, 6, 24, 5].

The benefits of having a private or shared LLC organization depend on the degree of sharing in an application as well as data access patterns. While private LLC organizations have low hit latencies, their off-chip miss rates are high in applications that exhibit high degrees of sharing due to data replication. Shared LLC organizations, on the other hand, have high hit latencies since each request has to complete a round-trip over the interconnection network. This hit latency increases as more cores are added since the diameter of most on-chip networks increases with the number of cores. However, their off-chip miss rates are low because data is not replicated.

Private LLC organizations limit the cache capacity available to a thread to that of the private LLC slice. This has an adverse effect on workloads with large private working sets and uneven distribution of working set sizes. Shared LLC organizations are not affected by this issue since they have the flexibility in storing the data of a thread in various locations throughout the LLC.

Both private and shared LLC organizations incur significant protocol latencies when a writer of a data block invalidates multiple readers; the impact being directly proportional to the degree of sharing of the data block. Previous research has proposed hybrid LLC organizations that combine the benefits of private and shared caches. Two such proposals are Reactive-NUCA [5] and Victim Replication [24].

Reactive-NUCA classifies data as private or shared using OS page tables at page granularity and manages LLC allocation according to the type of data. For a 16-core processor, R-NUCA places private data at the LLC slice of the requesting core, shared data at a single LLC slice whose location is determined by computing a hash function of the address, and replicates instructions at a single LLC slice for every cluster of 4 cores using a rotational interleaving mechanism.

Victim replication starts out with a private L1 and shared L2 organization and uses the local L2 slice as a victim cache for data that is evicted from the L1 cache. By only replicating the L1 capacity victims, this scheme attempts to combine the low hit latency of private designs with the low off-chip miss rates of shared LLCs.

The above schemes suffer two major drawbacks:

1. They leave the private caches unmanaged. A request for data allocates a cache line in the private cache hierarchy even if the data has no spatial or temporal locality. This leads to cache pollution since such low locality data can displace more frequently used data, or suffer from expensive communication due to the coherence protocol.
2. Management of on-chip data is based on coarse-grain data classification mechanisms and/or pays no attention to the locality of the cache lines. For example, victim replication places all L1 cache victims into the local L2 cache irrespective of whether they will be reused in the future. R-NUCA has a fixed policy for managing shared data and does not allow less or more replication based on the usefulness of data.

There has been significant research on managing the resource sharing of the LLC [18, 8]. These schemes either dynamically partition the cache to optimize for applications that benefit from the LLC, or better manage the LLC replacement policy based on the locality of cache lines in the private caches. To the contrary, our protocol focuses on intelligent management of private caches. Managing private caches is important because they replicate shared data without paying attention to its locality, and they are generally capacity-stressed due to strict size and latency limitations.

Previously, researchers have studied techniques for management of private caches in the context of uniprocessors. In Cache Bursts [15], the private cache fetches the entire cache line on every miss and evicts it as soon as it is detected to be a dead block. This does not accrue the network traffic or memory access latency benefits that our protocol enables by just fetching a single-use word for low locality data.

Selective caching has been suggested in the context of single processors to selectively cache data in the on-chip caches

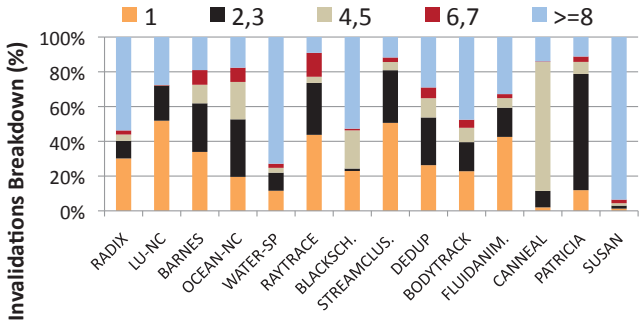

Figure 1: Invalidations vs Utilization.

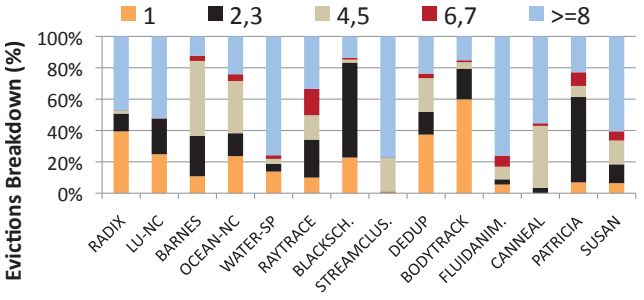

Figure 2: Evictions vs Utilization.

based on its locality [21, 10]. Our protocol is based on the similar central idea, but differs from selective caching in two key aspects: (1) In addition to performance we also optimize for energy consumption in the memory system. In [10], the referenced cache line is always brought into a set-associative buffer. To the contrary, our protocol selectively decides to move a cache line from the shared LLC to the private cache or simply accesses the requested word at the shared LLC, thereby reducing the energy consumption of moving unnecessary data through the on-chip network. (2) All prior selective caching proposals have focused on uniprocessors, while in this paper, we target large-scale shared memory multicores running multithreaded applications with *shared* data. In addition to the traditional private data, our protocol also tracks the locality of shared data and potentially converts expensive invalidations into much cheaper word accesses that not only improve memory latency but also reduce the energy consumption of the network and cache resources.

2.2 Motivation

In this paper we motivate the need for a *locality-aware* allocation of cache lines in the private caches of a shared memory multicore processor. The locality of each cache line is quantified using a *utilization* metric. The utilization is defined as the number of accesses that are made to the cache line by a core after being brought into its private cache hierarchy and before being invalidated or evicted.

Figures 1 and 2 show the percentage of invalidated and evicted cache lines as a function of their utilization. We observe that many cache lines that are evicted or invalidated in the private caches exhibit low locality (e.g., in STREAMCLUSTER, 80% of the cache lines that are invalidated have *utilization* < 4). To avoid the performance penalties of invalidations and evictions, we propose to only bring those cache lines that have high spatio-temporal locality into the private caches. This is accomplished by tracking vital statistics at the private caches and the on-chip directory to quantify the utilization of data at the granularity of cache lines. This utilization information is subsequently used to classify data as cache line or word accessible.

3. LOCALITY-AWARE ADAPTIVE CACHE COHERENCE PROTOCOL

We describe how the *Locality-Aware Adaptive Coherence* protocol works by implementing it on top of a Private-L1 Shared-L2 cache organization with R-NUCA's data placement and migration mechanisms and ACKwise limited directory coherence protocol.

3.1 Baseline System

The baseline system is a tiled multicore with an electrical 2-D mesh interconnection network as shown in Figure 3. Each core consists of a compute pipeline, private L1 instruction and data caches, a physically distributed shared L2 cache with integrated directory, and a network router. The coherence directory is integrated with the L2 slices by extending the L2 tag arrays 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 [13]. Some cores have a connection to a memory controller as well.

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 numbers of cores [13]. Our 2-D mesh network is also augmented with broadcast support. Each router selectively replicates a broadcast'ed message on its output links such that all cores are reached with a single injection.

3.2 Protocol Operation

We first define a few terms to facilitate describing our protocol.

- **Private Sharer**: A private sharer is a core which is granted a private copy of a cache line in its L1 cache.
- **Remote Sharer**: A remote sharer is a core which is *NOT* granted a private copy of a cache line. Instead, its L1 cache miss is handled at the shared L2 cache location using word access.
- **Private Utilization**: Private utilization is the number of times a cache line is used (read or written) by a core in its private L1 cache before it gets invalidated or evicted.
- **Remote Utilization**: Remote utilization is the number of times a cache line is used (read or written) by a core at the shared L2 cache before it is brought into its L1 cache or gets written to by another core.
- **Private Caching Threshold (PCT)**: The private utilization above or equal to which a core is "promoted" to be a private sharer, and below which a core is "demoted" to be a remote sharer of a cache line.

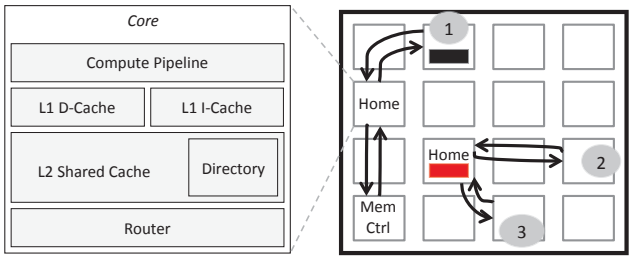

Figure 3: ①, ② and ③ are mockup requests showing the two modes of accessing on-chip caches using our *locality-aware* protocol. Since the *black* data block has high locality with respect to the core at ①, the directory at the home-node hands out a private copy of the cache line. On the other hand, the low-locality *red* data block is always cached in a single location at its home-node, and all requests (②, ③) are serviced using roundtrip remote-word accesses.

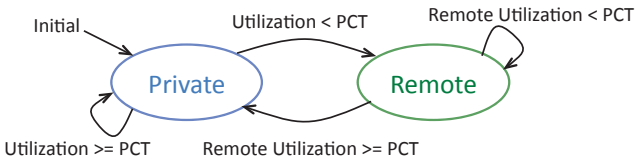

Figure 4: Each cache line is initialized to *Private* with respect to all sharers. Based on the utilization counters that are updated on each memory access to this cache line and the parameter *PCT*, the sharers are transitioned between *Private* and *Remote* modes. Here utilization = (private + remote) utilization.

Note that a cache line can have both private and remote sharers. We first describe the basic operation of the protocol. Later, we present heuristics that are essential for a cost-efficient hardware implementation. Our protocol starts out as a conventional directory protocol and initializes all cores as private sharers of all cache lines (as shown by *Initial* in Figure 4). Let us understand the handling of read and write requests under this protocol.

Read Requests: When a core makes a read request and misses in its private L1 cache, the request is sent to the L2 cache. If the cache line is not present in the L2 cache, it is brought in from off-chip memory. The L2 cache then hands out a private read-only copy of the cache line if the core is marked as a *private sharer* in its integrated directory (① in Figure 3). (Note that when a cache line is brought in from off-chip memory, all cores start out as private sharers). The core then tracks the locality of the cache line by initializing a *private utilization* counter in its L1 cache to **1** and incrementing this counter for every subsequent read. Each cache line tag is extended with utilization tracking bits for this purpose as shown in Figure 5.

Figure 5: Each L1 cache tag is extended to include additional bits for tracking (a) private utilization, and (b) last-access time of the cache line.

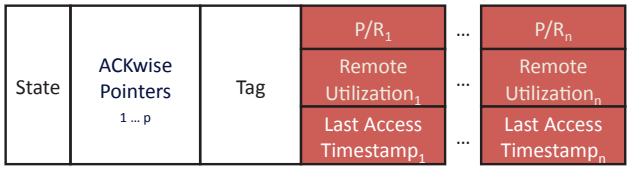

Figure 6: *ACKwise$_p$* - *Complete* classifier directory entry. The directory entry contains the state, tag, ACKwise$_p$ pointers as well as (a) mode (P/R), (b) remote utilization counters and (c) last-access timestamps for tracking the locality of all the cores in the system.

On the other hand, if the core is marked as a remote sharer, the integrated directory either increments a core-specific remote utilization counter or resets it to **1** based on the outcome of a *Timestamp* check (that is described below). If the remote utilization counter has reached *PCT*, the requesting core is promoted, i.e., marked as a private sharer and a copy of the cache line is handed over to it (as shown in Figure 4). Otherwise, the L2 cache replies with the requested word (② and ③ in Figure 3).

The *Timestamp* check that must be satisfied to increment the remote utilization counter is as follows. **The last-access time of the cache line in the L2 cache is greater than the minimum of the last-access times of all valid cache lines in the same set of the requesting core's L1 cache.** Note that if at least one cache line is invalid in the L1 cache, the above condition is trivially true. Each directory entry is augmented with a per-core *remote utilization* counter and a *last-access timestamp* (64-bits wide) for this purpose as shown in Figure 6. Each L1 cache tag also contains a *last-access timestamp* (shown in Figure 5) and this information is used to calculate the above minimum last-access time in the L1 cache. This minimum is then communicated to the L2 cache on an L1 miss.

The above *Timestamp* check is added so that when a cache line is brought into the L1 cache, other cache lines that are equally or better utilized are not evicted, i.e., the cache line does not cause L1 cache pollution. For example, consider a benchmark that is looping through a data structure with low locality. Applying the above *Timestamp* check allows the system to keep a subset of the working set in the L1 cache. Without the *Timestamp* check, a remote sharer would be promoted to be a private sharer after a few accesses (even if the other lines in the L1 cache are well utilized). This would result in cache lines evicting each other and ping-ponging between the L1 and L2 caches.

Write Requests: When a core makes a write request that misses in its private L1 cache, the request is sent to the L2 cache. The directory performs the following actions if the core is marked as a private sharer: (1) it invalidates all the private sharers of the cache line, (2) it sets the remote utilization counters of all its remote sharers to '0', and (3) it hands out a private read-write copy of the line to the requesting core. The core then tracks the locality of the cache line by initializing the private utilization counter in its L1 cache to **1** and incrementing this counter on every subsequent read/write request.

On the other hand, if the core is marked as a remote sharer, the directory performs the following actions: (1) it invalidates all the private sharers, (2) it sets the remote utilization counters of all remote sharers other than the requesting core to '0', and (3) it increments the remote utilization

counter for the requesting core, or resets it to **1** using the same *Timestamp* check as described earlier for read requests. If the utilization counter has reached PCT, the requesting core is promoted and a private read-write copy of the cache line is handed over to it. Otherwise, the word to be written is stored in the L2 cache.

When a core writes to a cache line, the utilization counters of all remote sharers (other than the writer itself) must be set to '0' since they have been unable to demonstrate enough utilization to be promoted. All remote sharers must now build up utilization again to be promoted.

Evictions and Invalidations: When the cache line is removed from the private L1 cache due to eviction (conflict or capacity miss) or invalidation (exclusive request by another core), the private utilization counter is communicated to the directory along with the acknowledgement message. The directory uses this information along with the remote utilization counter present locally to classify the core as a private or remote sharer in order to handle future requests.

If the ($private$ + $remote$) utilization is $\geq PCT$, the core stays as a private sharer, else it is demoted to a remote sharer (as shown in Figure 4). The remote utilization is added because if the cache line had been brought into the private L1 cache at the time its remote utilization was reset to **1**, it would not have been evicted (due to the *Timestamp* check and the LRU replacement policy of the L1 cache) or invalidated any earlier. Therefore, the actual utilization observed during this classification phase includes both the private and remote utilization.

Performing classification using the mechanisms described above is expensive due to the area overhead, both at the L1 cache and the directory. Each L1 cache tag needs to store the private utilization and last-access timestamp (a 64-bit field). And each directory entry needs to track locality information (i.e., mode, remote utilization and last-access timestamp) for all the cores. We now describe heuristics to facilitate a cost-effective hardware implementation.

In Section 3.3, **we remove the need for tracking the last-access time from the L1 cache and the directory.** The basic idea is to approximate the outcome of the *Timestamp* check by using a threshold higher than PCT for switching from remote to private mode. This threshold, termed *Remote Access Threshold (RAT)* is dynamically learned by observing the L1 cache set pressure and switches between multiple levels so as to optimize energy and performance.

In Section 3.4, we describe a mechanism for predicting the mode (private/remote) of a core by tracking locality information for only a limited number of cores at the directory.

3.3 Predicting Remote→Private Transitions

The *Timestamp* check described earlier served to preserve well utilized lines in the L1 cache. We approximate this mechanism by making the following two changes to the protocol: (1) de-coupling the threshold for remote-to-private mode transition from that for private-to-remote transition, (2) dynamically adjusting this threshold based on the observed L1 cache set pressure.

The threshold for remote-to-private mode transition, i.e., the number of accesses at which a core transitions from a remote to private sharer, is termed Remote Access Threshold (RAT). Initially, RAT is set equal to PCT (the threshold for private-to-remote mode transition). On an invalidation, if the core is classified as a remote sharer, RAT is unchanged.

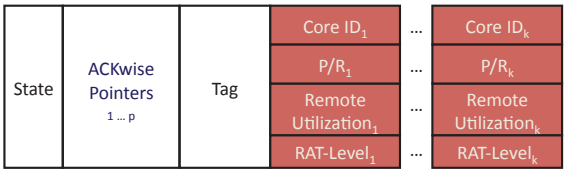

Figure 7: The limited locality classifier extends the directory entry with mode, utilization, and *RAT*-level bits for a limited number of cores. A majority vote of the modes of tracked cores is used to classify new cores as private or remote sharers.

This is because the cache set has an invalid line immediately following an invalidation leading to low cache set pressure. Hence, we can assume that the *Timestamp* check trivially passes on every remote access.

However, on an eviction, if the core is demoted to a remote sharer, RAT is increased to a higher level. This is because an eviction signifies higher cache set pressure. By increasing RAT to a higher level, it becomes harder for the core to be promoted to a private sharer, thereby counteracting the cache set pressure. If there are back-to-back evictions, with the core demoted to a remote sharer on each of them, RAT is further increased to higher levels. However, RAT is not increased beyond a certain value (RAT_{max}) due to the following two reasons: (1) the core should be able to return to the status of a private sharer if it later shows good locality, and (2) the number of bits needed to track remote utilization should not be too high. Also, beyond a particular RAT, keeping the core as a remote sharer counteracts the increased cache pressure negligibly, leading to only small improvements in performance and energy. The protocol is also equipped with a short-cut incase an invalid cache line exists in the L1 cache. In this case, if remote utilization reaches or rises above PCT, the requesting core is promoted to a private sharer since it will not cause cache pollution. The number of RAT levels used is abbreviated as $nRAT_{levels}$. RAT is additively increased in equal steps from PCT to RAT_{max}, the number of steps being equal to ($nRAT_{levels}$ −1).

On the other hand, if the core is classified as a private sharer on an eviction or invalidation, RAT is reset to its starting value of PCT. Doing this is essential because it provides the core the opportunity to re-learn its classification. Varying the RAT in this manner removes the need to track the last-access time both in the L1 cache tag and the directory. However, a field that identifies the current RAT-level now needs to be added to each directory entry. These bits now replace the last-access timestamp field in Figure 6. The efficacy of this scheme is evaluated in Section 5.2. Based on our observations, $RAT_{max} = 16$ and $nRAT_{levels} = 2$ were found to produce results that closely match those produced by the *Timestamp*-based classification scheme.

3.4 Limited Locality Classifier

The classifier described earlier which keeps track of locality information for all cores in the directory entry is termed the *Complete* locality classifier. It has a storage overhead of 60% (calculated in Section 3.6) at 64 cores and over 10× 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 private or remote sharers based on this information.

The locality information for each core consists of (1) the core ID, (2) a mode bit (P/R), (3) a remote utilization counter, and (4) a RAT-level. The classifier that maintains a list of this information for a limited number of cores (k) is termed the Limited$_k$ classifier. Figure 7 shows the information that is tracked by this classifier. The sharer list of ACKwise is not 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 private or remote 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 L2 cache, the directory first checks if the core is already being tracked by the limited locality list. If so, the actions described in Section 3.2 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 actions described in Section 3.2 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 private sharer becomes inactive on an invalidation or an eviction. A remote sharer becomes inactive on a write by another core. If a replacement candidate exists, its entry is allocated to the requesting core. The initial 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 5.3, we will evaluate the accuracy of the *Limited$_k$* classifier. Based on our observations, the *Limited$_3$* classifier produces results that closely match and sometimes exceeds those produced by the *Complete* classifier.

3.5 Selection of PCT

The Private Caching Threshold (PCT) is a parameter to our protocol and combining it with the observed spatio-temporal locality of cache lines, our protocol classifies data as private or remote. The extent to which our protocol improves the performance and energy consumption of the system is a complex function of the application characteristics, the most important being its working set and data sharing and access patterns. In Section 5, we will describe how these factors influence performance and energy consumption as PCT is varied for the evaluated benchmarks. We will also show that choosing a static PCT of 4 for the simulated benchmarks meets our performance and energy consumption improvement goals.

3.6 Overheads of the Locality-Based Protocol

Storage: The *locality-aware* protocol requires extra bits at the directory and private caches to track locality information. At the private L1 cache, tracking locality requires 2 bits for the private utilization counter per cache line (assuming an optimal PCT of 4). At the directory, the *Limited$_3$* classifier tracks locality information for three sharers. Tracking one sharer requires 4 bits to store the remote utilization counter (assuming an RAT_{max} of 16), 1 bit to store the mode, 1 bit to store the RAT-level (assuming 2 RAT levels) and 6 bits to store the core ID (for a 64-core processor). Hence, the *Limited$_3$* classifier requires an additional 36 (= 3 × 12) bits of information per directory entry. The *Complete* classifier, on the other hand, requires 384 (= 64 × 6) bits of information. The assumptions stated here will be justified in the evaluation section.

The following calculations are for one core but they are applicable for the entire system since all cores are identical. The sizes for the per-core L1-I, L1-D and L2 caches used in our system are shown in Table 1. The directory is integrated with the L2 cache, so each L2 cache line has an associated directory entry. The storage overhead in the L1-I and L1-D caches is $\frac{2}{512} \times (16 + 32) = 0.19KB$. We neglect this in future calculations since it is really small. The storage overhead in the directory for the *Limited$_3$* classifier is $\frac{36 \times 256}{64 \times 8} = 18KB$. For the *Complete* classifier, it is $192KB$. Now, the storage required for the *ACKwise$_4$* protocol in this processor is $12KB$ (assuming 24 bits per directory entry) and that for the *Full-Map* protocol is $32KB$. Adding up all the storage components, the ***Limited$_3$* classifier with the *ACKwise$_4$* protocol uses less storage than the *Full-map* protocol and 5.7% more storage than the baseline *ACKwise$_4$* protocol** (factoring in the L1-I, L1-D and L2 cache sizes also). The *Complete* classifier with ACKwise$_4$ uses 60% more storage than the baseline ACKwise$_4$ protocol.

Cache Accesses: Updating the private utilization counter in a cache requires a read-modify-write operation on every cache hit. This is true even if the cache access is a read. However, the utilization counter, being just 2 bits in length, can be stored in the tag array. Since the tag array already needs to be written on every cache hit to update the replacement policy (e.g. LRU) counters, our protocol does not incur any additional cache accesses.

Network Traffic: The locality-aware protocol could create network traffic overhead due to the following three reasons:

1. The private utilization counter has to be sent along with the acknowledgement to the directory on an invalidation or an eviction.
2. In addition to the cache line address, the cache line offset and the memory access length has to be communicated during every cache miss. This is because the requester does not know whether it is a private or remote sharer (only the directory maintains this information as explained previously).
3. The data word(s) to be written has (have) to be communicated on every cache miss due to the same reason.

Some of these overheads can be hidden while others are accounted for during evaluation as described below.

1. Sending back the utilization counter can be 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

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	16 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., 7 cycle Inclusive, R-NUCA
Cache Line Size	64 bytes
Directory Protocol	Invalidation-based MESI ACKwise$_4$ [13]
Num. of Memory Controllers	8
DRAM Bandwidth	5 GBps per Controller
DRAM Latency	100 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
Header (Src, Dest, Addr, MsgType)	1 flit
Word Length	1 flit (64 bits)
Cache Line Length	8 flits (512 bits)
Locality-Aware Coherence Protocol - Default Parameters	
Private Caching Threshold	$PCT = 4$
Max Remote Access Threshold	$RAT_{max} = 16$
Number of RAT Levels	$nRAT_{levels} = 2$
Classifier	Limited$_3$

Table 1: Architectural parameters for evaluation.

IDs and 2 bits for the utilization counter. The remaining 8 bits suffice for storing the message type.

2. The cache line offset needs to be communicated but not the memory access length. We profiled the memory access lengths for the benchmarks evaluated and found it to be 64 bits in the common case. Memory accesses that are $\leq$ 64 bits in length are rounded-up to 64 bits while those $>$ 64 bits always fetch an entire cache line. Only 1 bit is needed to indicate this difference.
3. The data word to be written (64 bits in length) is always communicated to the directory on a write miss in the L1 cache. This overhead is accounted for in our evaluation.

3.7 Simpler One-Way Transition Protocol

The complexity of the above protocol could be decreased if cores, once they were classified as remote sharers w.r.t. a cache line would stay in the same mode throughout the program. If this were true, the storage required to track locality information at the directory could be avoided except for the mode bits. The bits to track private utilization at the cache tags would still be required to demote a core to the status of a remote sharer. We term this simpler protocol $Adapt_{1\text{-}way}$ and in Section 5.4, we observe that this protocol is worse than the original protocol by 34% in completion time and 13% in energy. Hence, a protocol that incorporates dynamic transitioning between both modes is required for efficient operation.

4. EVALUATION METHODOLOGY

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

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 [17] multicore simulator. All the mechanisms and protocol overheads discussed in Section 3 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 *Locality-Aware Adaptive Coherence* protocol requires two separate access widths for reading or writing the shared L2 caches i.e., a word for remote sharers and a cache line for private sharers. For simplicity, we assume the same L2 cache access latency for both word and cache line accesses.

4.2 Energy Models

We evaluate just dynamic energy. For energy evaluations of on-chip electrical network routers and links, we use the DSENT [20] tool. Energy estimates for the L1-I, L1-D and L2 (with integrated directory) caches are obtained using McPAT [14]. The evaluation is performed at the 11 nm technology node to account for future technology trends.

When calculating the energy consumption of the L2 cache, we assume a word addressable cache architecture. This allows our protocol to have a more efficient word access compared to a cache line access. We model the dynamic energy consumption of both the word access and the cache line access in the L2 cache.

4.3 Application Benchmarks

We simulate six SPLASH-2 [22] benchmarks, six PARSEC [3] benchmarks, four Parallel-MI-Bench [7], a Travelling-Salesman-Problem (TSP) benchmark, a Depth-First-Search (DFS) benchmark, a Matrix-Multiply (MATMUL) benchmark, and two graph benchmarks (CONNECTED-COMPONENTS & COMMUNITY-DETECTION) [1] using the Graphite multicore simulator. The graph benchmarks model social networking based applications.

4.4 Evaluation Metrics

Each application is run to completion using the input sets from Table 2. For each simulation run, we measure the *Completion Time*, i.e., the time in *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 four components. (1) **L1 to L2 cache latency** is the time spent by the L1 cache miss request to the L2 cache and the corresponding reply from the L2 cache including time spent in the network and the first access to the L2 cache. (2) **L2 cache waiting time** is the queueing delay incurred because requests to the same cache line must be serialized to ensure memory consistency. (3) **L2 cache to sharers latency** is the roundtrip time needed to invalidate private sharers and receive their acknowledgments. This also includes time spent requesting and receiving synchronous write-backs. (4) **L2 cache 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.

We also measure the energy consumption of the memory

Application	Problem Size
SPLASH-2	
RADIX	1M Integers, radix 1024
LU	512 × 512 matrix, 16 × 16 blocks
BARNES	16K particles
OCEAN	258 × 258 ocean
WATER-SPATIAL	512 molecules
RAYTRACE	car
PARSEC	
BLACKSCHOLES	64K options
STREAMCLUSTER	8192 points per block, 1 block
DEDUP	31 MB data
BODYTRACK	2 frames, 2000 particles
FLUIDANIMATE	5 frames, 100,000 particles
CANNEAL	200,000 elements
Parallel MI Bench	
DIJKSTRA-SINGLE-SOURCE	Graph with 4096 nodes
DIJKSTRA-ALL-PAIRS	Graph with 512 nodes
PATRICIA	5000 IP address queries
SUSAN	PGM picture 2.8 MB
UHPC	
CONNECTED-COMPONENTS	Graph with 2^{18} nodes
COMMUNITY-DETECTION	Graph with 2^{16} nodes
Others	
TSP	16 cities
DFS	Graph with 876800 nodes
MATRIX-MULTIPLY	512 × 512 matrix

Table 2: Problem sizes for our parallel benchmarks.

system which includes the on-chip caches and the network.

One of the important memory system metrics we track to evaluate our protocol is the various cache miss types. They are as follows: (1) **Cold misses** are cache misses that occur to a line that has never been previously brought into the cache, (2) **Capacity misses** are cache misses to a line that was brought in previously but later evicted to make room for another line, (3) **Upgrade misses** are cache misses to a line in read-only state when an exclusive request is made for it, (4) **Sharing misses** are cache misses to a line that was brought in previously but was invalidated or downgraded due to an read/write request by another core, and (5) **Word misses** are cache misses to a line that was remotely accessed previously.

5. RESULTS

We have compared the baseline ACKwise$_4$ with a full-map directory protocol and the average performance and energy consumption were found to be within 1% of each other. Since the ACKwise protocol scales well with increasing core counts, we use that as a baseline throughout this section. The architectural parameters from Table 1 are used for the study unless otherwise stated.

In Section 5.1, we perform a sweep study to understand the trends in *Energy* and *Completion Time* for the evaluated benchmarks as *PCT* is varied. In Section 5.2, we evaluate the approximation scheme for the *Timestamp*-based classification and determine the optimal number of remote access threshold levels ($nRAT_{levels}$) and maximum RAT threshold (RAT_{max}). Next, in Section 5.3, we evaluate the accuracy of the limited locality tracking classifier ($Limited_k$) by performing a sensitivity analysis on k. Section 5.4 compares the *Energy* and *Completion Time* of the protocol against the simpler one-way transition protocol ($Adapt_{1\text{-}way}$).

5.1 Energy and Completion Time Trends

Figures 8 and 9 plot the energy and completion time of the evaluated benchmarks as a function of *PCT*. Results are normalized in both cases to a *PCT* of 1 which corresponds to the baseline R-NUCA system with ACKwise$_4$ directory protocol. Both energy and completion time decrease initially as *PCT* is increased. As *PCT* increases to higher values, both completion time and energy start to increase.

5.1.1 Energy

We consider the impact that our protocol has on the energy consumption of the memory system (L1-I cache, L1-D cache, L2 cache and directory) and the interconnection network (both router and link). The distribution of energy between the caches and network varies across benchmarks and is primarily dependent on the L1-D cache miss rate. For this purpose, the L1-D cache miss rate is plotted along with miss type breakdowns in Figure 10.

Benchmarks such as WATER-SP and SUSAN with low cache miss rates (∼0.2%) dissipate 95% of their energy in the L1 caches while those such as CONCOMP and LU-NC with higher cache miss rates dissipate more than half of their energy in the network. The energy consumption of the L2 cache compared to the L1-I and L1-D caches is also highly dependent on the L1-D cache miss rate. For example, WATER-SP has negligible L2 cache energy consumption, while OCEAN-NC's L2 energy consumption is more than its combined L1-I and L1-D energy consumption.

At the 11nm technology node, network links have a higher contribution to the energy consumption than network routers. This can be attributed to the poor scaling trends of wires compared to transistors. As shown in Figure 8, this trend is observed in all our evaluated benchmarks.

The energy consumption of the directory is negligible compared to all other sources of energy consumption. This motivated our decision to put the directory in the L2 cache tag arrays as described earlier. The additional bits required to track locality information at the directory have a negligible effect on energy consumption.

Varying *PCT* impacts energy by changing both network traffic and cache accesses. In particular, increasing the value of *PCT* decreases the number of private sharers of a cache line and increases the number of remote sharers. This impacts the network traffic and cache accesses in the following three ways. (1) Fetching an entire line on a cache miss in conventional coherence protocols is replaced by multiple word accesses to the shared L2 cache. Note that each word access at the shared L2 cache requires a lookup and an update to the utilization counter in the directory as well. (2) Reducing the number of private sharers decreases the number of invalidations (and acknowledgments) required to keep all cached copies of a line coherent. Synchronous write-back requests that are needed to fetch the most recent copy of a line are reduced as well. (3) Since the caching of low-locality data is eliminated, the L1 cache space can be more effectively used for high locality data, thereby decreasing the amount of asynchronous evictions (leading to capacity misses) for such data.

Benchmarks that yield a significant improvement in energy consumption do so by converting either capacity misses

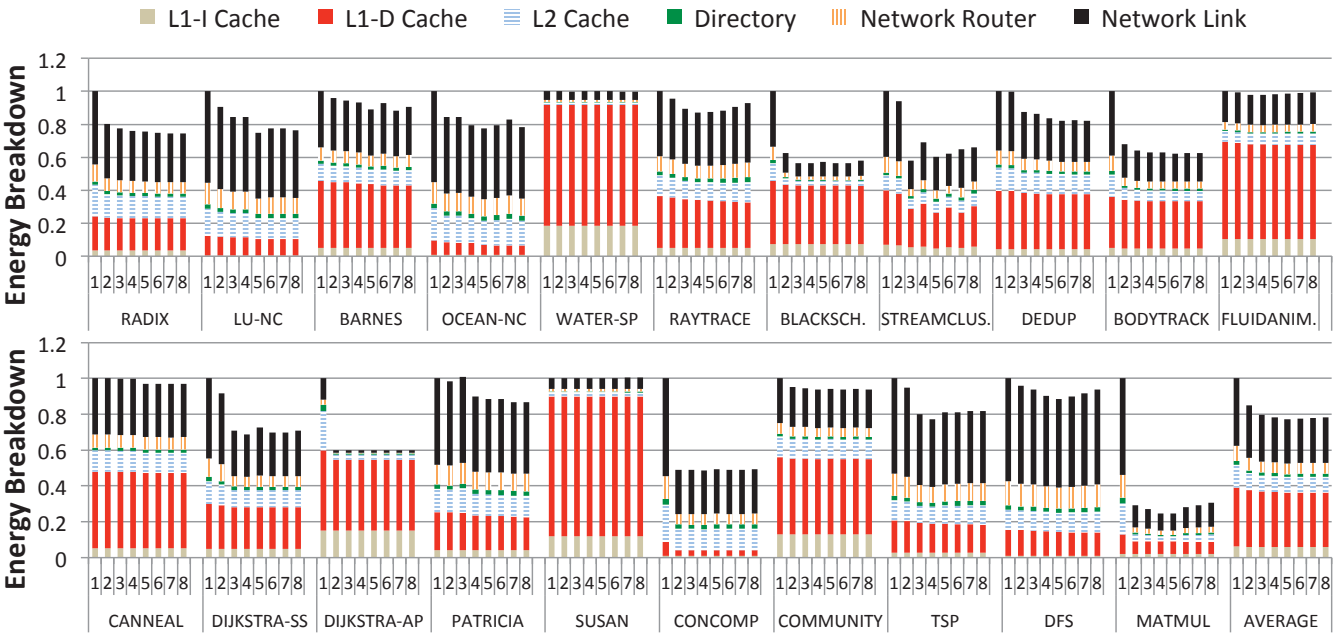

Figure 8: Variation of Energy with PCT. Results are normalized to a PCT of 1. Note that *Average* and not *Geometric-Mean* is plotted here.

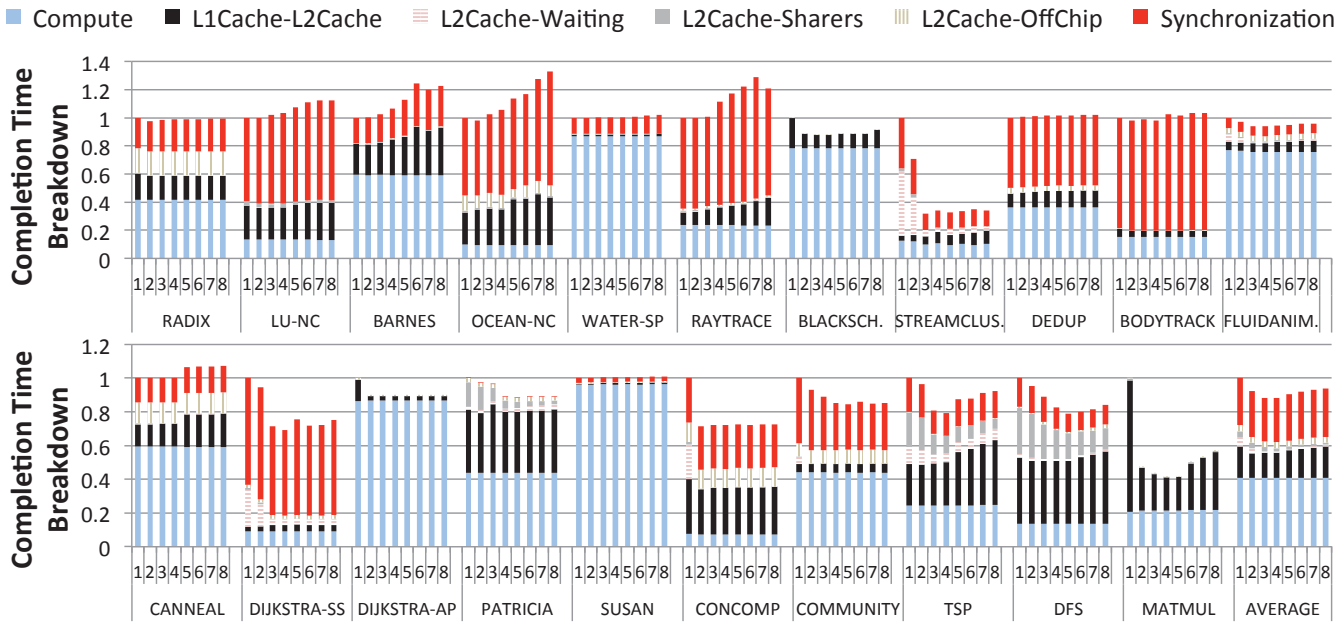

Figure 9: Variation of Completion Time with PCT. Results are normalized to a PCT of 1. Note that *Average* and not *Geometric-Mean* is plotted here.

(in BODYTRACK and BLACKSCHOLES) or sharing misses (in DIJKSTRA-SS and STREAMCLUSTER) into cheaper word misses. This can be observed from Figure 8 when going from a PCT of 1 to 2 in BODYTRACK and BLACKSCHOLES and a PCT of 2 to 3 in DIJKSTRA-SS and STREAMCLUSTER. While a sharing miss is more expensive than a capacity miss due to the additional network traffic generated by invalidations and synchronous write-backs, turning capacity misses into word misses improves cache utilization (and reduces cache pollution) by reducing evictions and thereby capacity misses for other cache lines. This is evident in benchmarks like BLACKSCHOLES, BODYTRACK, DIJKSTRA-AP and MATMUL in which the cache miss rate drops when switching from a PCT of 1 to 2. Benchmarks like LU-NC, and PATRICIA provide energy benefit by converting both capacity and sharing misses into word misses.

At a PCT of 4, the geometric mean of the energy consumption across all benchmarks is less than that at a PCT of 1 by 25%.

5.1.2 *Completion Time*

As shown in Figure 9, our protocol reduces the completion

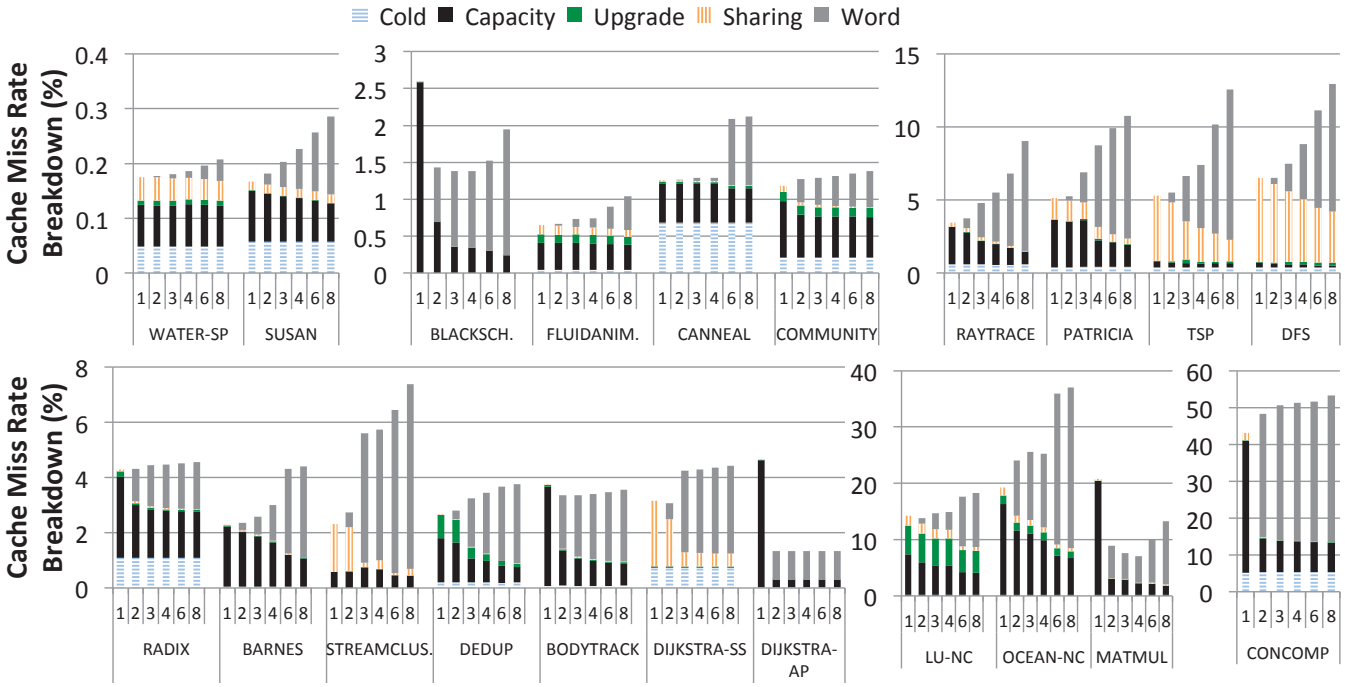

Figure 10: L1 Data Cache Miss Rate and Miss Type Breakdown vs PCT. Note that in this graph, the miss rate increases from left to right as well as from top to bottom.

time as well. Noticeable improvements (>5%) occur in 11 out of the 21 evaluated benchmarks. Most of the improvements occur for the same reasons as discussed for energy, and can be attributed to our protocol identifying low locality cache lines and converting the capacity and sharing misses on them to cheaper word misses. Benchmarks such as BLACKSCHOLES, DIJKSTRA-AP and MATMUL experience a lower miss rate when PCT is increased from 1 to 2 due to better cache utilization. This translates into a lower completion time. In CONCOMP, cache utilization does not improve but capacity misses are converted into almost an equal number of word misses. Hence, the completion time improves.

Benchmarks such as STREAMCLUSTER and TSP show completion time improvement due to converting expensive sharing misses into word misses. From a performance standpoint, sharing misses are expensive because they increase: (1) the L2 cache to sharers latency and (2) the L2 cache waiting time. Note that the L2 cache waiting time of one core may depend on the L2 cache to sharers latency of another since requests to the same cache line need to be serialized. In these benchmarks, even if cache miss rate increases with PCT, the miss penalty is lower because a word miss is much cheaper than a sharing miss. A word miss does not contribute to the L2 cache to sharers latency and only contributes marginally to the L2 cache waiting time. Hence, the above two memory access latency components can be significantly reduced. Reducing these components may decrease synchronization time as well if the responsible memory accesses lie within the critical section. STREAMCLUSTER and DIJKSTRA-SS mostly reduce the L2 cache waiting time while PARTRICIA and TSP reduce the L2 cache to sharers latency.

In a few benchmarks such as LU-NC and BARNES, completion time is found to increase after a PCT of 3 because the added number of word misses overwhelms any improvement obtained by reducing capacity misses.

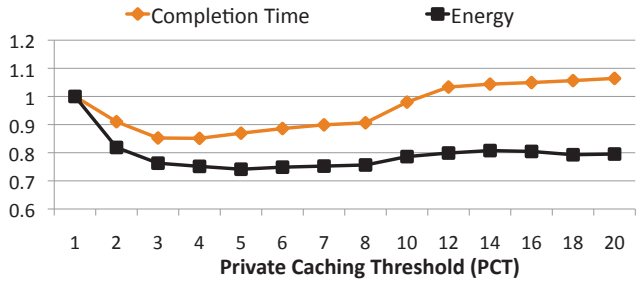

Figure 11: Variation of *Geometric-Means* of Completion Time and Energy with Private Caching Threshold (PCT). Results are normalized to a PCT of 1.

At a PCT of 4, the geometric mean of the completion time across all benchmarks is less than that at a PCT of 1 by 15%.

5.1.3 *Static Selection of PCT*

To put everything in perspective, we plot the geometric means of the *Completion Time* and *Energy* for our benchmarks in Figure 11. We observe a gradual decrease of completion time up to *PCT* of 3, constant completion time till a *PCT* of 4, and an increase in completion time afterward. Energy consumption decreases up to *PCT* of 5, then stays constant till a *PCT* of 8 and after that, it starts increasing. We conclude that a *PCT* of 4 meets our goal of simultaneously improving both completion time and energy consumption. A completion time reduction of 15% and an energy consumption improvement of 25% is obtained when moving from *PCT* of 1 to 4.

5.2 Tuning Remote Access Thresholds

As explained in Section 3.3, the *Timestamp*-based classification scheme was expensive due to its area overhead, both

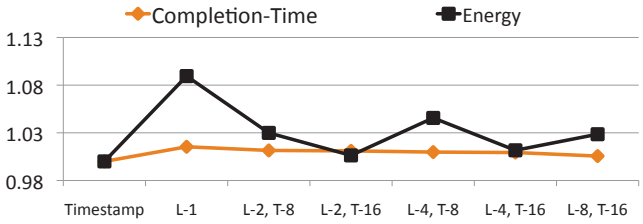

Figure 12: Remote Access Threshold sensitivity study for $nRAT_{levels}$ (L) and RAT_{max} (T)

at the L1 cache and directory. The benefits provided by this scheme can be approximated by having multiple *Remote Access Threshold (RAT) Levels* and dynamically switching between them at runtime to counteract the increased L1 cache pressure. We now perform a study to determine the optimal number of threshold levels ($nRAT_{levels}$) and the maximum threshold (RAT_{max}).

Figure 12 plots the completion time and energy consumption for the different points of interest. The results are normalized to that of the *Timestamp*-based classification scheme. The completion time is almost constant throughout. However, the energy consumption is nearly 9% higher when $nRAT_{levels} = 1$. With multiple RAT levels ($nRAT_{levels} > 1$), the energy is significantly reduced. Also, the energy consumption with $RAT_{max} = 16$ is found to be slightly lower (2%) than with $RAT_{max} = 8$. With $RAT_{max} = 16$, there is almost no difference between $nRAT_{levels} = 2, 4, 8$, so we choose $nRAT_{levels} = 2$ since it minimizes the area overhead.

5.3 Limited Locality Tracking

As explained in Section 3.4, tracking the locality information for all the cores in the directory results in an area overhead of 60% per core. So, we explore a mechanism that tracks the locality information for only a few cores, and classifies a new core as a private or remote sharer based on a majority vote of the modes of the tracked cores.

Figure 13 plots the completion time and energy 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 WATER-SP, i.e., the *completion time* and *energy* stay constant as k varies. The experiments are run with the best static PCT value of 4 obtained in Section 5.1.3. We observe that the completion time and energy consumption of the Limited$_3$ classifier never exceeds by more than 3% the completion time and energy consumption of the Complete classifier.

In STREAMCLUSTER and DIJKSTRA-SS, the Limited$_3$ classifier does better than the Complete classifier because it learns the mode of sharers quicker. While the Complete classifier starts off each sharer of a cache line independently in private mode, the Limited$_3$ classifier infers the mode of a new sharer from the modes of existing sharers. This enables the Limited$_3$ classifier to put the new sharer in remote mode without the initial per-sharer classification phase. We note that the Complete locality classifier can also be equipped with such a learning short-cut.

Inferring the modes of new sharers from the modes of existing sharers can be harmful too, as is illustrated in the case of RADIX and BODYTRACK for the Limited$_1$ classifier. While RADIX starts off new sharers incorrectly in remote mode, BODYTRACK starts them off incorrectly in private

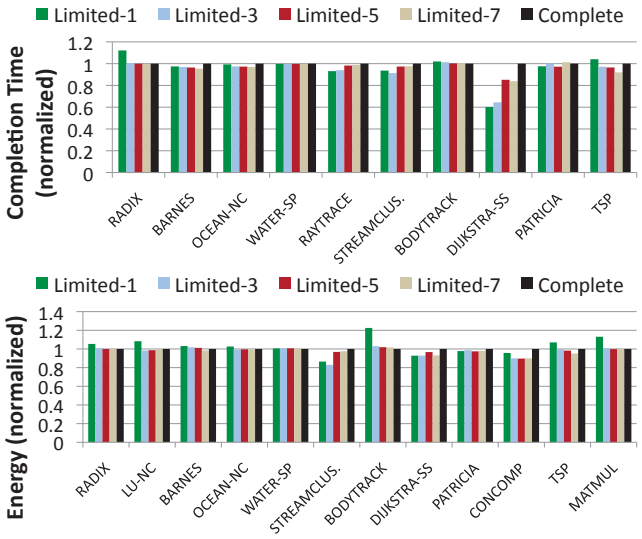

Figure 13: Variation of *Completion Time* and *Energy* with the number of hardware locality counters (k) in the Limited$_k$ classifier. Limited$_{64}$ is identical to the Complete classifier. Benchmarks for which results are not shown are identical to WATER-SP, i.e., the *Completion Time* and *Energy* stay constant as k varies.

mode. This is because the first sharer is classified as remote (in RADIX) and private (in BODYTRACK). This causes other sharers to also reside in that mode while they actually want to be in the opposite mode. Our observation from the above sensitivity experiment is that tracking the locality information for three sharers suffices to offset such incorrect classifications.

5.4 Simpler One-Way Transition Protocol

In order to quantify the efficacy of the dynamic nature of our protocol, we compare the protocol to a simpler version having only one-way transitions ($Adapt_{1\text{-}way}$). The simpler version starts off all cores as private sharers and demotes them to remote sharers when the utilization is less than the private caching threshold (PCT). However, these cores then stay as remote sharers throughout the lifetime of the program and can never be promoted. The experiment is run with the best PCT value of 4.

Figure 14 plots the ratio of completion time and energy for the $Adapt_{1\text{-}way}$ protocol over our protocol (which we term $Adapt_{2\text{-}way}$). Higher the ratio, higher is the need for two-way transitions. We observe that the $Adapt_{1\text{-}way}$ protocol is worse in completion time and energy by 34% and 13% respectively. In benchmarks such as BODYTRACK and DIJKSTRA-SS, the completion time ratio is worse by 3.3× and 2.3× respectively.

6. CONCLUSION

In this paper, we have proposed a *locality-aware* adaptive cache coherence protocol that enables seamless adaptation between private and logically shared caching at the fine granularity of cache lines. Our data-centric approach relies on in-hardware runtime profiling of the *locality* of each cache line and only allows private caching for data blocks with high spatio-temporal locality. We have proposed low-overhead locality tracking mechanisms and used extensive simulation results to verify their accuracy. We implemented our proto-

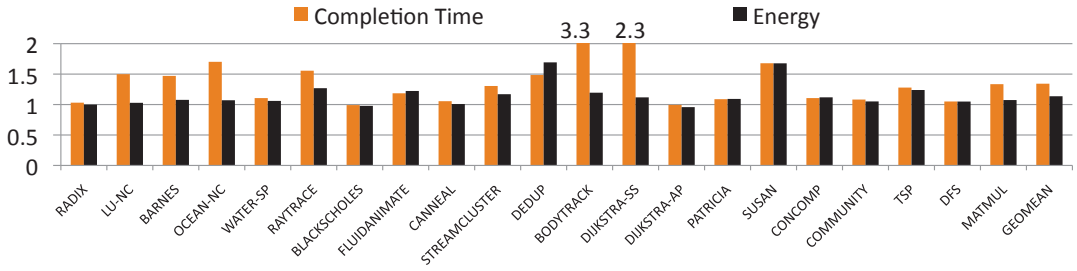

Figure 14: Ratio of *Completion Time* and *Energy* of $Adapt_{1\text{-}way}$ over $Adapt_{2\text{-}way}$

col on top of a Private-L1, Shared-L2 implementation that uses the Reactive-NUCA data management scheme and the ACKwise$_4$ limited directory protocol. The results indicate that our protocol reduces the overall *Energy Consumption* in a 64-core multicore by 25%, while improving the *Completion Time* by 15%. Our protocol can be implemented with only 18 KB storage overhead per core when compared to the ACKwise$_4$ limited directory protocol, and has a lower storage overhead than a full-map directory protocol.

7. REFERENCES

[1] DARPA UHPC Program BAA. `https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-37/listing.html`, March 2010.

[2] 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 *International Solid-State Circuits Conference*, 2008.

[3] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, 2008.

[4] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *In IEEE Micro*, 30(2):16–29, 2010.

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

[6] H. Hoffmann, D. Wentzlaff, and A. Agarwal. Remote Store Programming: A memory model for embedded multicore. In *International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[7] S. Iqbal, Y. Liang, and H. Grahn. ParMiBench - an open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters*, 2010.

[8] 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 *Int'l Symposium on Microarchitecture*, 2010.

[9] N. E. Jerger, L.-S. Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *Int'l Symposium on Computer Architecture*, 2008.

[10] T. L. Johnson and W.-M. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Int'l Symposium on Computer architecture*, 1997.

[11] 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.

[12] C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[13] 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 *Int'l Conference on Parallel Architectures and Compilation Techniques*, 2010.

[14] 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 *Int'l Symposium on Microarchitecture*, 2009.

[15] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Int'l Symposium on Microarchitecture*, 2008.

[16] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[17] 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 *Int'l Symposium on High Performance Computer Architecture*, 2010.

[18] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Int'l Symposium on Microarchitecture*, 2006.

[19] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Int'l Symposium on High Performance Computer Architecture*, 2012.

[20] 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 *Int'l Symposium on Networks-on-Chip*, 2012.

[21] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Int'l Symposium on Microarchitecture*, 1995.

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

[23] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *Int'l Symposium on Microarchitecture*, 2009.

[24] M. Zhang and K. Asanović. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Int'l Symposium on Computer Architecture*, 2005.

[25] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan. SPATL: Honey, I Shrunk the Coherence Directory. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, 2011.