ISSN: 2278-4632
Vol-10 Issue-1 January 2020

CACHE HOMOGENEITY IN COLLECTIONS

Mrs.Pragyan Paramita Panda[1], Mr.Alok Kumar Pattnaik[2]*
*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
pragyanparamita@thenalanda.com alokkumar@thenalanda.com*

***Abstract*— An effective approach for chip multiprocessors and multicores is directory-based cache coherence. The collection of acknowledgement messages and invocation messages for the directory protocol, however, need multicast, which can be expensive in terms of latency and network traffic. Furthermore, as the number of cores rises, so does the directory's size. We offer Library Cache Coherence (LCC), which eliminates the need for invalidation acknowledgements or broadcasting or multicasting invalidations. A library is a collection of timestamps that are used to delay writing to shared cache lines until all shared copies have expired and to automatically invalidate shared cache lines. The number of cores has no bearing on the size of the library. LCC produces fewer network messages since it does away with the intricate invalidation procedure of directory-based cache coherence protocols. When a write to a cache block is being delayed, LCC also permits reads on the block to occur without compromising sequential consistency. Our experimental results on LCC with an ideal timestamp scheme (though not implementable) show the potential for further improvement for LCC with more sophisticated timestamp schemes. As a result, LCC has 1.85X less average memory latency than a MESI directory-based protocol on our set of benchmarks.**

I. INTRODUCTION

With the demise of Dennard scaling, the increase in processor clock frequencies from 1980-2003 has slowed down significantly [1]. To improve performance, archi- tects are exploring many parallel architectures includ- ing manycore architectures in academia (e.g., Raw [2], TRIPS [3]) and industry (e.g., Tilera [4], Intel Ter- aFLOPS [5]). In a manycore or multi-core architecture, cores with relatively low complexity are connected to memory and each other via high-bandwidth on-chip interconnect.

How will these multicores be programmed? Amongst many different types of parallel programming models, message passing and shared memory are the most dom- inant ones. Some multicores, for example, the TileraTile-Gx 100 provide a shared memory abstraction tothe programmer, while other architectures like Intel [1]equal contributors TeraFLOPS rely on message passing. Message passingis a very efficient programming model for certain types of applications, such as scientific computation. However,many programs and operating systems are based on the shared memory abstraction, so it is indispensablefor general-purpose multicores to support the shared memory abstraction.

On-chip cache memory cannot be *directly* imple- mented as a single large cache primarily because the energy consumption of caches grows quadratically with cache size, and because the number of read and write ports do not scale with the number of cores. To maintainperformance, we need distributed caches that behave likea logically shared cache.

A. Directory-Based Cache Coherence (DirCC) Architec-ture

When we consider a two-level on-chip cache hierar- chy for a tiled multicore architecture, there

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

are many choices in implementing a logically shared cache. Oneof the most common approaches in modern multicore processors is to implement a *private* L1 cache and a *shared* L2 cache slice for each core (e.g., Tilera's 64- core processor [4], Cavium Octeon 32-core processor). A shared L2 cache architecture unifies the per-core L2 cache slices into one large logically shared cache, and a memory address has a unique location where it can be cached on-chip, which is termed the *home* cache for the address. Therefore, data can be replicated in L1 but not in L2. On an L1 miss, the *home* L2 cache, where thedata block may be located, needs to be looked up; the home can be a local L2 cache if the address is mappedto the requesting core's cache (*core hit*), or a remoteL2 cache if the core is not the home for the address(*core miss*). While these remote accesses may lead to a higher average L2 hit latency, the on-chip cache is better utilized by not replicating any data in L2, reducing the number of expensive off-chip accesses when compared to a private L2 design.

The common shared memory abstraction model re- quires data coherence between cores. In this example of the shared L2 cache architecture, private L1 caches need to be kept coherent. When the number of cores is large (> 32), snoopy caches are no longer viable [6], and we are left with directory-based cache coherence [7]. There are difficulties in scaling directories to hundreds of cores, since the directory sizes grow with the number of cores. A fullmap directory is a directory that keeps track ofall the sharers of each cache block [8]. If, say, we have 256 cores on the chip, we will require a 256-bit vector for each shareable cache block, unless compression tech- niques, e.g., [9], [10], are applied, which may degrade performance.

Moreover, directory protocols that maintain sequen- tially consistent operation may require up to four net- work messages, that include broadcast (multicast) to all (a subset of) cores, and return acknowledgements. Broadcast/multicast can be expensive in terms of latency and network traffic and therefore many techniques have been developed to alleviate this expense (e.g., [11], [12]).

B. Our contribution: Library Cache Coherence Protocol(LCC)

We propose *Library Cache Coherence* (LCC), a novel cache coherence protocol which 1) does not require broadcast/multicast of invalidations (and therefore, no collection of invalidation acknowledgements), 2) is scal-able to any number of cores, and 3) guarantees sequentialconsistency.

There are several points worth of note before we describe the LCC protocol:

- We assume a global timer that is available to all cores and caches. (This timer does not have to increment with the processor clock, and can be significantly slower.)
- Each memory address has a unique home L2 loca- tion.
- Each line in the L1/L2 cache has an additional timestamp. This timestamp is interpreted in two different ways. If the data block is *not* in its home L2 location, the timestamp indicates the time until when the line is valid (i.e., can be read), and if it is at its home L2, the timestamp holds the *maximum* value of timestamps among all the copies of the datablock.
- A line can only be written at its home location, i.e., writes requested by a processor have to be sent to the home L2 cache.

We will describe the protocol informally here and more formally in Section III. A high-level example of LCC with a shared L2 cache is also illustrated in Figure 1. While a data block can only be written in its home L2 location, when a processor makes a request

for a word in the data block, read-only copies of data blocks with timestamps can be stored in L1

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

caches. The timestamp assigned to the data block is assigned by the home L2 cache when it sends the block to the requesting core (cf. Figure 1a,1b). The requested *word* from the block is loaded into a register, and a copyof the block is stored in the local L1 cache providedthat its timestamp has not expired. The home L2 cache keeps the information about the timestamps assigned to each data block that is stored in it, for those blocks that have been shared. Crucially, for any given data block,the home cache needs to just keep information about the *maximum* value of the timestamp assigned to any copyof the data block. Any write request to the data block in the home cache will not occur until the timestamp has expired. In other words, we delay writes to a block until all the read copies of the data block have expired in theirrespective locations to maintain sequential semantics (cf.Figure 1c,1d).

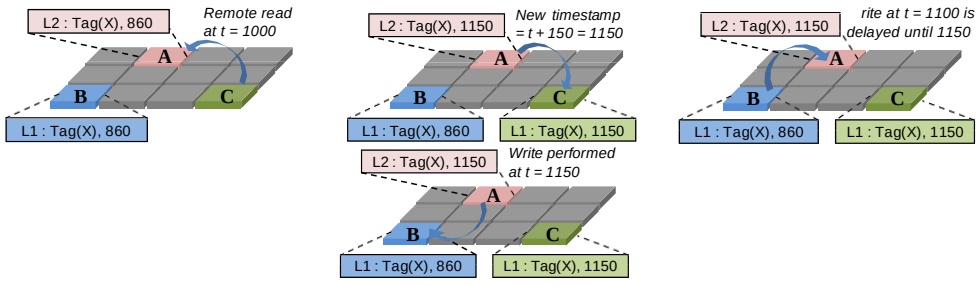
C. Comparing the Library and the Directory Protocol

The library protocol does not require broad-cast/multicast of invalidation messages, and thereforeit need not wait for invalidation acknowledgements toarrive, when writing to a shared cache line. However,as writes are only done at the L2 cache of the homecore, they do not exploit spatio-temporal locality andmay be delayed if the cache line being written has an unexpired copy elsewhere. This implies that the choiceof the timestamps is crucial to performance and we showhow it affects the performance of LCC in Section IV-A.

For the directory-based protocol, directories are stored at L2 caches combined with cache blocks, and lines inL1 caches have state information on whether the lineis shared or exclusive. On the other hand, the library protocol requires per-cache-line timestamps for both the L1 and L2 caches. In terms of scalability, the area overhead of directory-based protocols becomes greateras the number of cores grows, as the sharer information grows linearly with the number of cores in a full-map directory scheme, whereas the library protocol incursonly a constant overhead independent of the number of cores by storing timestamps of fixed size in the L1 and L2 caches.

II. RELATED WORK

Reducing coherence overhead in distributed shared memory (DSM) has been widely explored over decades. Dynamic Self-Invalidation (DSI) [13] eliminates inval- idation messages by having a processor self-invalidateits local copy of a cache block before a conflicting



(a) Suppose core B has a copy of address X in its L1 and its timestamp is 860. Core C wants to read X, but misses in its L1 cache, resulting in a remote read to the L2 cache at core A. The remote read arrives at supposedly, t = 1000.

(b) *Tdelta*, 150, is added to the current clock, 1000, and data is sent back to core C (L1 cache) with this new timestamp of 1150. The timestamp at core A's L2 cache is also updated. When core B wants to writeon X, it needs to perform a remote write to the home core A, and suppose it

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

arrives at t = 1100. Since the current time has not past the timestamp (1150) yet, it should wait until t be- comes greater than 1150. Once the timestamp has ex- pired (i.e., t 1150), the write can be performed and the ac- knowledgement is sent back.

Fig. 1. An example of LCC for the private L1 and shared L2 configuration : assume core A to be the home core for memory address X, and we add a constant *Tdelta* of 150 to the system time to decide a new timestamp. Each box in the figure consists of *Tag* and *Timestamp* of the cache line for address X. Note that the entry in the home L2 cache always keeps the maximum value of timestamps given out by the home core, and other cores can hold the block only in their L1 caches.

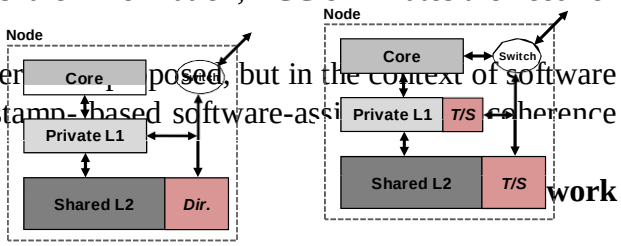
memory request by another processor, and the directory needs to identify the blocks to be self-invalidated. DSI, however, still requires invalidation acknowledgements to maintain sequential consistency. Last-Touch Predictor (LTP) [14] predicts the ”last touch” to a memory block by one processor before the block is accessed and subsequently invalidated by another using trace-based correlation. This, however, comes at a high cost; to learn a last-touch, all the invalidation messages for a processor must be exposed to the corresponding LTP, and the DSM controller needs to be integrated in the processor. Moreover, both DSI [13] and LTP [14] are built on top of a directory-based protocol since blocks that are not self- invalidated still need to be explicitly invalidated in the in [17]). The scheme in [17] is not sequentially consis- tent, and only supports a weaker notion of consistency, namely release consistency. In order to be sequentially consistent, we need to maintain a library of timestamps at the home cache, which also assigns timestamps for data requested. Our timestamps have two different meanings, and unlike in the scheme of [17], it is quite possible for a cache block to have expired by the time it arrives at the requesting core because of interconnect delays; we can, however, still use the word that was requested by the processor.

III. LIBRARY CACHE COHERENCE

A. Baseline Architecture

conventional manner. This necessitates directories and sharer information; LCC eliminates the need for sharer information and therefore, is more scalable.

Timestamp-based approaches for self-invalidation were proposed, but in the context of software cache coherence. Min and Baer [15] proposed a timestamp-based software-assisted cache coherence scheme which detects and invalidates the possibly incoherent



(a) Directory-based
(b) Library

Network

entry, essentially reassigning the entire burden of maintaining cache coherence to the compiler/software. Timestamp-based Selective Invalidation Scheme (TB- SIS) [16] explicitly invalidates stale cache copies using a special invalidation instruction if the corresponding variable is modified in the current epoch. Again, the compiler must insert this instruction at the proper place at compilation time. Unlike these schemes, library cache coherence is a full hardware coherence scheme and does not require additional compiler or software support.

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

Nandy and Narayan [17] proposed a hardware-based, auto-invalidate cache coherence protocol. They assume a fixed life-time t_c for each cache line that begins with the load of the data in the *requesting* cache (cf. Figure 3bFig. 2. Baseline architectures for DirCC and LCC (T/S = timestamp)

Figure 2 shows the baseline architecture for directory- based CC (DirCC) and LCC. Although LCC can be implemented on either the private L2 or the shared L2 organization, we choose the shared L2 architecture (withprivate L1) as a baseline, and use the same architecture for both DirCC and LCC for a fair comparison.

As shown in Figure 2a, directories and L2 cachesare integrated for DirCC, so the sharer information iscontained together with each cache line. LCC does notrequire directories, but instead requires per-cache-linetimestamps for both the L1 and L2 caches (Figure 2b).Under the shared L2 architecture, each memory ad- dress has a unique home location. In this paper, we use a static data placement scheme, *striping*, at pagegranularity, which allows the computation of the home core from the memory address using simple logic.

Since data cannot be replicated in shared L2 caches, each L2 slice can maintain only the *home* data, and the data fetched from remote L2 caches can only be cached in L1. For the library protocol, it is important to notethat the timestamp of L1 and that of L2 have different meanings. A timestamp in the L1 cache line indicates thetime until when the block can be read. A timestamp in the L2 cache, on the other hand, keeps the *maximum* value of timestamps that have been assigned by the home L2 cache, until when the L2 cache must prevent writing on the block. As all read copies at L1 cacheswill be discarded when their timestamps expire, the home L2 cache can write to the block when the system time becomes larger than the maximum timestamp valuestored at the L2 cache.

B. Baseline Protocol

Below is the full description of library cache coher- ence protocol:

- *Read* operations
  1) At the requesting core : The requesting core first looks up its local L1 cache, and proceeds to the L2 cache of the home location if itmisses in its L1. The request to the L2 cache will be a local access if the request gets a *corehit*, and a remote access if it gets a *core miss*. Note that an L1 cache block will be invalid when its timestamp expires, so it won't get a cache hit on an expired cache line.
  2) At the home L2 cache : If the data does not exist in the L2 cache, it will first be brought to the cache from the DRAM. Once the cache block is ready, it is returned to the requester core along with a timestamp. Timestamp choosing logic is used to decide whether to use the old timestamp which was previously given to other L1 caches, or to use a new timestamp. The maximum timestamp among all timestamps issued to L1 caches is kept in the L2 cache.
- *Write* operations
  1) At the requesting core : A write request isdirectly forwarded to its home L2. L1 caches do not have to be looked up for writes, since writes can be only done in home L2 locations and L1 caches only maintain read-only copies.
  2) At the home L2 cache : If the data does not exist in the L2 cache, it will be first broughtto the cache from the DRAM. Once the cache block is ready, the L2 cache checks if its times-tamp has expired. If it has not expired yet, the write is delayed until the current system time reaches the timestamp value and the line has expired; this is a *write delay*.

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

When a cache block is brought in from DRAM, it has a null timestamp (cf. Section III-E.)

C. Request Scheduling at the L2 Cache (Home)

Multiple requests on the same cache line may arriveat an L2 cache at the same time. Scheduling of these requests is very important in order to provide functional correctness and to prevent starvation. The library proto- col processes one request at a time, serializing both read and write requests on the cache block. Read requests and write requests are treated the same, and we assumea fair scheduling between requests from the local L1 cache and remote L1 caches. When a write request is being delayed due to an unexpired timestamp, however, the protocol holds the blocked write request until the timestamp expires and processes other read requests if available. This may cause a read request that arrivedlater to be processed before a blocked write request. However, this still maintains sequential consistency since the reordered execution sequence is yet another se- quential interleaving of programs. We have assumed in- order cores that have a single outstanding write request to simplify the requirements for sequential consistency. However, this assumption is not strictly required for LCC as conventional techniques to support sequential consistency for multi-issue or out-of-order cores can be applied to LCC as well without significant modifications.

D. Choosing Timestamps

As shown in the protocol, timestamps affect whether memory reads in L1 caches result in hits or misses, and also affect the amount of write delays at L2 caches. If a timestamp is too small, a core gets more L1 read misses as its read copy is invalidated too soon. If it is too large, on the other hand, the write delays will compromisesystem performance. Thus, timestamp choosing logicplays a critical role in determining the performance of library cache coherence.

Here, we present the simplest timestamp strategy, namely a FIXED-DELTA scheme. FIXED-DELTA uses a constant value T_{delta} to decide timestamps. When a read request for block X is being processed at its home L2 cache:

1) if there is a write request on the same block X waiting to be served, the current timestamp of the

   L2 entry is returned without any change to preventincreasing write delay.
2) otherwise, a new timestamp for the read request is calculated as *(current clock* + T_{delta}*)* at the homeL2 cache, and is returned to the requesting core's L1 cache with the data block.

TABLE I
SYSTEM CONFIGURATIONS

Parameter	Settings
Cores	64 in-order, single issue cores
L1 cache/ core	8 KB, 2-way set associativity L2 cache/ core 128 KB, 4-way set associativity

We also implement and evaluate an IDEAL scheme; in IDEAL, home L2 caches are assumed to know when theL1/L2 ReplacementPolicyLRU/LRU next

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

write request on each cache line *will* arrive, and/or when each cache line *will* be evicted by a capacity miss. Timestamps are set to the time when the earlier event willhappen, so read requests get the maximum timestamp values that do not delay any writes or evictions. AlthoughIDEAL is not implementable in real hardware, the per- formance of IDEAL tells us the performance potentialfor LCC, provided we have a smart timestamp choosing scheme that can predict close to optimal timestamp values. While relegating the development of such a scheme to future work, we sweep the value of $T_{delta}$for scheme FIXED-DELTA, and show how close the best FIXED-DELTA scheme can perform when compared to the IDEAL scheme (cf. Section IV).

E. Cache Replacement Policy

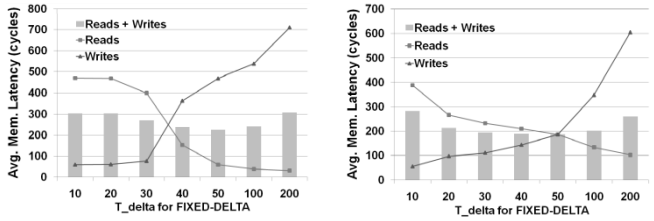
The library protocol can be implemented with anyconventional cache replacement scheme with one modi- fication to ensure the correctness of the protocol. While L1 cache evictions can be done anytime, an L2 cache entry with unexpired copies should not be evicted until all the copies expire. If the cache entry gets evicted, we lose track of the timestamp for the corresponding entry, and thus, we cannot fetch the correct timestamp value when the entry is restored from memory — this will break sequential consistency. This restriction may result in cache eviction delays if all the entries in multiple waysof the L2 cache happen to have unexpired copies.

IV. EVALUATION

Using Pin [18] and Graphite [19], we first gener- ate memory instruction traces for a set of SPLASH-
2 benchmarks [20]: FFT, LU CONTIGUOUS, OCEAN, RADIX and WATER. Then, we feed these traces to HORNET [21], a cycle-level multicore simulator, which models the interconnect behavior (e.g., congestion) in a cycle-accurate manner. The important system parameters used in the experiments are listed in Table I.

A. FIXED-DELTA *with varying* T_{delta}

We first swept the value of T_{delta} for the FIXED- DELTA scheme, and Figure 3 shows a tradeoff betweenL1/L2 Access Latency 2 cycles/4 cyclesCache Line Size 32 bytes
Electrical network 2D Mesh, XY routing, 64-bitflits
Data Placement STRIPE, 4 KB page size Directory Coherence MESI, Full-map distributed di-
rectory
DRAM Acess Latency 50 cycles One-way Off-chip La- 150 cycles
tency
Simulated Cycles 5,000,000 cycles



Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

(a) FFT (b) LU CONTIGUOUS

Fig. 3. Average memory latency for varying *Tdelta*. Other bench- marks are not shown for lack of space, but have the same trends.

the read latency and write latency. As we increase T_{delta}, the L1 cache hit rate for reads increases because the entry becomes valid for a longer period of time, resulting in lower average memory latency for reads.For writes, however, the average latency increases with alarger T_{delta} due to the write delays. The combination of these two factors determine the optimal range of T_{delta} which minimizes the overall memory latency; for our experiments, T_{delta} between 50 and 100 showed the best performance depending on the benchmark, but without a big difference in performance within the range.

B. Performance Comparison with DirCC and LCC-IDEAL

Figure 4 shows the performance of the best FIXED- DELTA LCC and the IDEAL LCC, both normalized to the MESI directory protocol. For all the benchmarks we run except for OCEAN, LCC shows better performance than DirCC, and on average (geometric mean), the best LCC- Fixed outperforms DirCC by 1.85x. This performance gain mainly comes from the fact that 1) LCC does

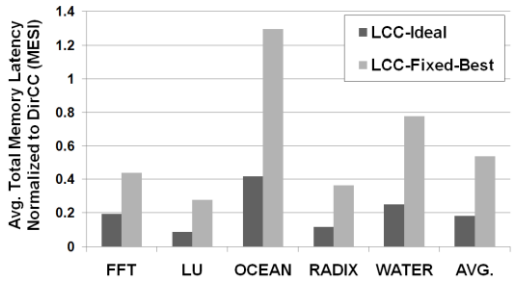


Fig. 4. Average overall memory latency for the LCC-Ideal and the best LCC-Fixed normalized to the MESI directory protocol.

not send out invalidation requests and wait for their acknowledgements, and 2) LCC can serve subsequent read requests while delaying a write on the same line maintaining sequential consistency. Moreover, the per- formance of the ideal LCC is much better (outperforms DirCC by 5.5x), showing the potential for further im- provement of LCC. For example, since each cache line has a different access pattern, a timestamp choosing scheme that captures such patterns and assigns different T_{delta}*'s* for each line at runtime may provide better performance.

V. CONCLUSIONS

We presented a library cache coherence protocol that can replace conventional directory-based cache coher- ence protocols in manycore architectures. LCC outper- forms a directory-based MESI protocol as it avoids expensive invalidations before a write on a shared cache line. Also, it serves read requests first when a write is pending because it is not allowed to write on a cache line, which further increases the memory access perfor- mance while maintaining sequential consistency. LCC is also more scalable than directory-based coherence since library size does not grow with the number ofcores. We believe that the performance of LCC can be improved further with more sophisticated schemes of choosing timestamps, which provides an interesting research problem to the community. Finally,

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

while we have focused on sequential consistency in this paper, LCC can also improve performance for weaker memory consistency models.

REFERENCES

[1] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007, pp. 746–749.

[2] E. Waingold, M. Taylor, D. Srikrishna *et al.*, "Baring it all to Software: Raw Machines," in *IEEE Computer*, September 1997, pp. 86–93.

[3] K. Sankaralingam, R. Nagarajan, H. Liu *et al.*, "Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture," in *International Symposium on Computer Architecture (ISCA)*, June 2003, pp. 422–433.

[4] David Wentzlaff et al, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sept/Oct 2007.

[5] S. R. Vangal, J. Howard, G. Ruhl *et al.*, "An 80-Tile Sub-100- W TeraFLOPS processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, 2008.

[6] A. Agarwal, R. Simoni, J. Hennessy *et al.*, "An evaluation of directory schemes for cache coherence," in *In Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 280–289.

[7] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coher- ence schemes," in *International Conference on Parallel Processing*, 1990.

[8] D. Chaiken, C. Fields, K. Kurihara *et al.*, "Directory-based cache coherence in large-scale multiprocessors," in *COM- PUTER*, 1990.

[9] J. Zebchuk, V. Srinivasan, M. K. Qureshi *et al.*, "A tag- less coherence directory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 423–434.

[10] H. Zhao, A. Shriraman, and S. Dwarkadas, "Space: sharing pattern-based directory coherence for multicore scalability," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 135–146.

[11] N. Eisley, L.-S. Peh, and L. Shang, "In-network cache coher- ence," in *Proceedings of the 39th Annual IEEE/ACM Interna- tional Symposium on Microarchitecture*, 2006, pp. 321–332.

[12] J. A. Brown, R. Kumar, and D. Tullsen, "Proximity-aware directory-based coherence for multi-core processor architec- tures," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, 2007, pp. 126–134.

[13] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Re- ducing coherence overhead in shared-memory multiprocessors," in *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[14] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self- invalidation using last-touch prediction," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00, 2000.

[15] S. L. Min and J. L. Baer, "Design and analysis of a scalable cache coherence scheme based on clocks and timestamps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 25–44, January 1992.

[16] X. Yuan, R. Melhelm, and R. Gupta, "A timestamp-based selec- tive invalidation scheme for multiprocessor cache coherence," *International Conference on Parallel Processing*, vol. 3, p. 0114, 1996.

Copyright @ 2020 Authors

ISSN: 2278-4632
Vol-10 Issue-1 January 2020

[17] S. K. Nandy and R. Narayan, "An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems," in *Pro- ceedings of the First International Workshop on Parallel Processing*, 1994.

[18] M. M. Bach, M. Charney, R. Cohn *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, pp. 34–41, 2010.

[19] J. E. Miller, H. Kasture, G. Kurian *et al.*, "Graphite: A dis- tributed parallel simulator for multicores," in *HPCA*, 2010, pp. 1–12.

[20] S. Woo, M. Ohara, E. Torrie *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995, pp. 24–36.

[21] M. Lis, P. Ren, M. H. Cho *et al.*, "Scalable, accurate multicore simulation in the 1000-core era," in *International Symposium on Performance Analysis of Systems and Software (ISPASS 2011)*, April 2011.

Copyright @ 2020 Authors