RWT: Suppressing Write-Through Cost When Coherence is Not Needed

Hao Liu, Clément Dévigne, Lucas Garcia, Quentin L. Meunier, Franck Wajsbürt, Alain Greiner

RWT: Suppressing Write-Through Cost when Coherence is not Needed

Hao Liu, Clément Dévigne, Lucas Garcia, Quentin Meunier, Franck Wajsbürt, Alain Greiner
Laboratoire d'Informatique de Paris 6 UMR 7606
Université Pierre et Marie Curie UPMC – Sorbonne Universités,
4 Place Jussieu, 75252 Paris Cedex 05 France
Email: lucas.garcia@polytechnique.edu, *firstname.lastname*@lip6.fr

***Abstract*—In shared-memory multicore architectures, handling a write cache operation is more complicated than in single-processor systems. A cache line may be present in more than one private L1 cache. Any cache willing to write this line must inform all the other sharers. Therefore, it is necessary to implement a cache coherence protocol for multicore architectures.**

At present, directory based protocols are popular cache coherence protocols in both industry and academic domains because of their reduced coherence traffic compared to snooping protocols, at the expense of an indirection. The write policy – write through or write back – is crucial in the protocol design.

The write-through policy reduces the bandwidth because it augments the write traffic in the interconnection network, and also augments the energy consumption. However, it can efficiently solve the false sharing problem via write updates. In this paper, we introduce a new way to reduce the write traffic of a write-through coherence protocol by combining write-through coherence with a write-back policy for non coherent lines. The baseline write-through used as reference is a scalable hybrid invalidate/update protocol.

Simulation results show that with our enhanced protocol, we can reduce at least by 50% the write traffic in the interconnection network, and gain up to 20% performance compared with the baseline write-through protocol.

***Index Terms*—System-on-Chip; Many-core Architecture; Shared Memory Programming; Hardware Cache Coherence; Network-on-Chip; Write-Through; Write-Back; Released Write-Through**

I. Introduction

The problem of cache coherence protocols in shared-memory multicore architectures is still an active domain of research. Thanks to technology advances, we can put more and more cores in one single chip [1], making it impossible to use bus as an interconnect in the architecture. Instead, most multicore or manycore architectures now use networks-on-chip (NoC) [2] as interconnect [3], [4]. With a NoC, snooping coherence can only come at the price of many broadcasts; and if this approach has been experimented (e.g. in [5]), directory-based coherence protocols are the most commonly used in multicore architectures.

One of the most popular cache coherence protocol, the write-back MESI [6], supports directory based implementations, like the MOESI protocol in AMD architecture Opteron [7], and the GOLS protocol in Intel architecture Xeon Phi [8]. The MESI protocol uses the write-back policy to avoid propagating writes to the lower levels of cache hierarchy. However, it can lead to a performance diminution when a write targets a line not present in the L1 cache: this is because the write-back policy implements a *write allocate* policy which needs to fetch the missing line before performing the write. During this time, the processor is blocked. On the contrary, a write-through based protocol always sends the write command to the lower level of cache hierarchy, and thus is not blocked in case of miss. However, this increases the write traffic and consumes more energy than a write-back policy.

This work extends currently submitted work on the TSAR architecture [9]. The coherence protocol used in this architecture, called DHCCP (Dynamic Hybrid Cache Coherence Protocol), is a directory-based write-through protocol whose coherence traffic scales with the number of cores. However, DHCCP has two main disadvantages:

- First, the high number of writes from L1 caches to L2 caches limits the number of processors per cluster
- Second, these writes induce a high power consumption

Our objective is to reduce this write traffic by modifying DHCCP, while keeping the advantage of write-through coherence.

To achieve this, we propose a modified version of this coherence protocol, called RWT for *Released Write-Through*. The idea of RWT is to enable coherency only when it is necessary. This necessity will be determined fully in hardware, with a per line granularity in the L2 cache. Thus, a coherent line will use the same version of DHCCP as presented in [10]. However, for lines which have only one copy and thus do not need coherence, a write-back approach is used. This technique allows to reduce the traffic related to writes for the majority of the lines which are present in a single cache, while keeping a scalable coherence protocol for shared lines. A line can then be in one of two states:

- Non-coherent write-back
- Coherent write-through

We can notice that the state change can only occur from non-coherent state to coherent state. The state of a line is reset when it is evicted from the L2 cache.

The rest of the article is organized as follows: section II discusses related works; section III presents the mechanism of RWT; section IV presents the simulation environment and the experimental results; finally, section V concludes.

II. Related Work

Cache coherence protocols have been studied for a long time. In traditional multiprocessor systems in which processors

shared a bus, the MESI protocol and its variants have proven to be the most efficient [11]. However, this problem recently regained in interest with the arrival of manycore systems, which replaced buses interconnects with NoCs of varying topology, offering a higher bandwidth, but a higher latency. A possible answer to the problem is to use software coherence and let programmers invalidate shared data when needed [12], or use different programming paradigms like message passing. However, it is now more and more accepted that hardware supported on-chip coherence will be part of future multi- and manycore chips [13]. Yet, the question of the coherence protocol remains open, and the write-through strategy could be an actual answer for manycore systems because of NoCs characteristics: [14] shows that a write-through invalidate coherence scheme can perform as well as a write-back MESI one over a NoC, while being simpler to design; besides, the commercially available Tilera architecture [4] uses a write-through protocol between tiles.

Other approaches try to reduce network latency in case of a write miss. Token Coherence [15], [16] is a technique which allows to avoid indirection at the L2 level by broadcasting requests on all the mesh, while detecting race conditions and guaranteeing correctness by a token mechanism. This is an optimistic approach using potentially failing requests, enabling a fast common case. The DicoCMP protocol and its variants [17], [18] store the sharers in both the L1 and the L2 caches. The approach used consists in that a requesting writer L1 cache must send invalidations directly to the other L1 cache owning a copy instead of the L2 cache (home tile), at the price of tracking the list of sharers for each line in the L1 caches.

While these approaches are interesting, they put efforts to minimize write-miss latency, which is almost non-existing with write-through protocols, since the latter just need to propagate writes to the lower level of the memory hierarchy without blocking the requesting L1 or processor.

[19] presents a mechanism including a policy manager able to detect when a cache powers up/down, allowing a write-back policy when a single cache is activated, and a write-through policy otherwise. While this work uses a hybrid policy like RWT, it has a cache granularity which does not fit well with manycore systems. At the opposite, RWT allows several active caches to access memory in write-back mode, provided the caches access different pieces of data.

Finally, [20] describes a hybrid technique for cache coherence, combining a coherent strategy with a non coherent one. The main difference with RWT is that this approach is based on the operating system for detecting that data is shared; this in turn involves to handle TLB misses by software – while TSAR allows a hardware handling – and restricts the strategy granularity to pages, whereas RWT allows a per line granularity.

III. Principles of RWT

A. Baseline Write-Through

The write-through coherence protocol DHCCP used as baseline is based on the following idea: when the number of copies in L1 caches is below a certain threshold, an explicit list of the sharers is kept in the L2, while above this threshold, only the number of copies is known. The explicit copies are stored in a L2 data structure shared among all lines. Thus, when the number of copies is low, a multicast update strategy is used upon receiving a write, whereas when the number of copies is high, a broadcast invalidation is sent to all L1 caches. In the following, this protocol is considered as a write-through coherent protocol and is not fully described, as this article focuses on the write-back strategy for non coherent lines, and the coupling with the write-through coherent lines. Finally, despite the fact that we only considered DHCCP as a baseline, the principles of RWT could be applied to any directory-based write-through coherent protocol.

B. Modifying Cleanups to Add a Write-Back Mechanism

In DHCCP, the distributed shared L2 cache directory keeps coherence information for every cache line, and this requires the inclusive property of L1 caches in L2 caches. A *cleanup* request sent by a L1 cache can happen in two scenarios:

- the line has been evicted spontaneously by the L1 cache
- the L1 cache has received an invalidation request from L2 cache and responds with a *cleanup*

The *cleanup* request does not include any data in DHCCP, because the L2 cache line is always up to date. In RWT, we need to keep this inclusive property for non-coherent lines: if a line is locally modified in a L1 cache and not present in a L2 cache, a miss in another L1 on this line will result in an incoherence when the line is then retrieved by the L2. It implies that when a non-coherent line is evicted from a L2, the L1 owning the copy must both invalidate its copy and send the up-to-date values to the L2.

When a L1 cache spontaneously evicts a dirty line or when it receives an invalidation request targeting a dirty line, the *cleanup* needs to be done in parallel with propagating up-to-date values to the L2. This can *a priori* be done *via* three different ways:

- the L1 can send a write with the new values in parallel with the cleanup, thus using two different networks (one for the cleanup and one for the write);
- the L2 can interpret the write with the new values as a response to its coherence request;
- the L1 can send the new values directly in the response to the coherence request.

The first solution has the drawback that since the two transactions use different networks, their order of arrival is not defined, adding complexity in the protocol. The second solution can actually not be achieved easily because it actually presents a risk of deadlock. For these reasons, we decided to modify the existing *cleanup* request to include a potential write-back of dirty lines; this request is called *cleanup-data*. For coherent lines and the non-coherent clean lines, we use the classic *cleanup* requests as in DHCCP – i.e. without data.

C. Non-Coherent and Coherent States in L1 Caches

The RWT protocol defines two states for a valid cache line: the non-coherent (NC) state and the coherent (C) state. This

state is determined by the L2 cache, and is sent along with a miss response to a requesting L1 cache. When a line is fetched from memory by the L2 cache, the requesting L1 cache obtains the first copy of this line, and thus is granted NC state.

When a write hit happens in a L1 cache, the strategy used depend on the line state.

- the write-through policy is chosen for a line in C state. The L1 cache line is updated and the write is also sent to the L2 cache.
- the write-back policy is chosen for a line in NC state. The L1 cache line is locally modified and the lines become *dirty*: there is no write sent to the L2 cache.

Because the write-allocate strategy blocks the requesting processor, write misses in RWT are directly sent to the corresponding L2 cache. Figure 1 models the decision diagram associated with the actions taken by a L1 cache upon receiving a processor request.

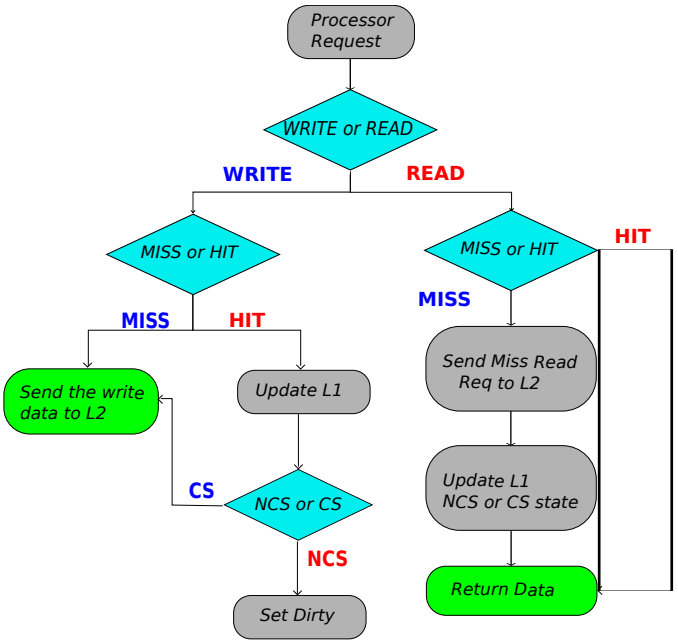

Figure 1: Decision Diagram of a L1 Cache upon Receiving a Processor Request

D. Switching From Non-Coherent to Coherent State

A line in a L2 cache is initially in the NC state, thus there can be only one copy in a given L1 cache. A key point in RWT is the detection by the L2 that a line currently in NC state ought to be in C state. This happens when a L1 cache requests a copy of (*resp.* issues a write on) a line which is in NC state and already has a copy in another L1 (it is possible that a shared line only has one copy). This triggers the L2 cache to send an invalidation request to the actual owner of the copy, which responds with a *cleanup-data* if the line is dirty, and *cleanup* otherwise (see Figure 2). When the L2 cache receives the *cleanup*, it changes the state for this line from NC state to C state, thus the L2 cache sends the copy with the C state information to the L1 cache requesting the line. Figure 2 illustrates this line state change mechanism in RWT.

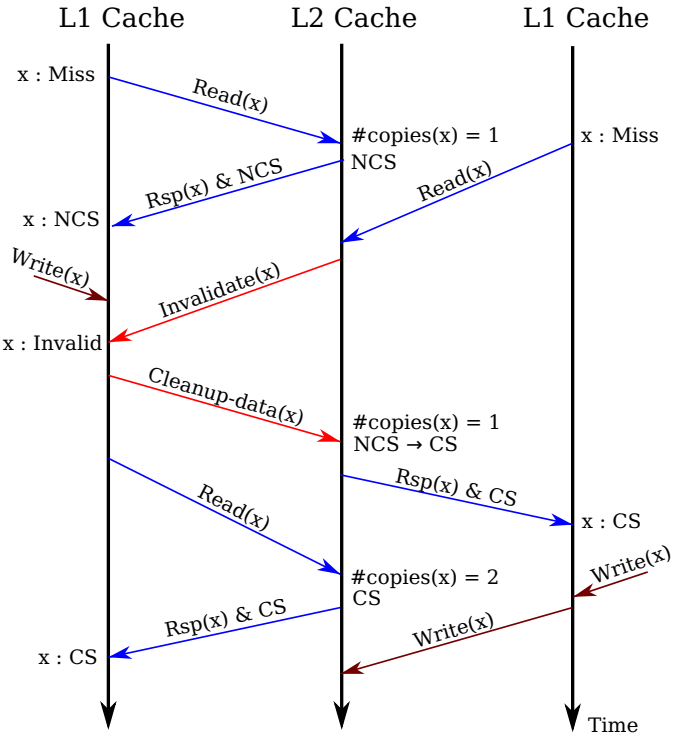

Figure 2: Line State Change Mechanism: From Non-Coherent State (NCS) to Coherent State (CS), for a Line Named x

RWT does not allow a line in C state to be switched back in NC state: the rationale behind this is that even if the number of copies goes back down to one, there is a high probability that another cache will access it in the near future since this line contains shared data. Of course, when a line is evicted from a L2 and then fetched back from memory, its status is reset to NC state. Thus, this line lives in C state until it is evicted from the L2 cache.

To avoid an unnecessary switch overhead for shared cache lines containing instructions – which are almost never modified – RWT has been implemented only for the data part of the L1 cache.

IV. Evaluation

A. Architecture

We implemented our proposed RWT cache coherence protocol at the cycle-accurate level, by modifying the models of L1 and L2 caches of the original TSAR components. The TSAR model is written using a library of cycle-accurate components called SoCLib [21], which uses the SystemC standard [22].

Figure 3 shows an overview of the TSAR architecture. It is a clustered architecture with a 2D mesh topology using a network-on-chip. Each cluster contains 4 Mips cores, a shared L2 cache, and a local crossbar connected to the router. There are separate networks for direct requests (reads and writes) and coherence requests in order to avoid deadlocks. Table I shows the configuration parameters for the TSAR architecture.

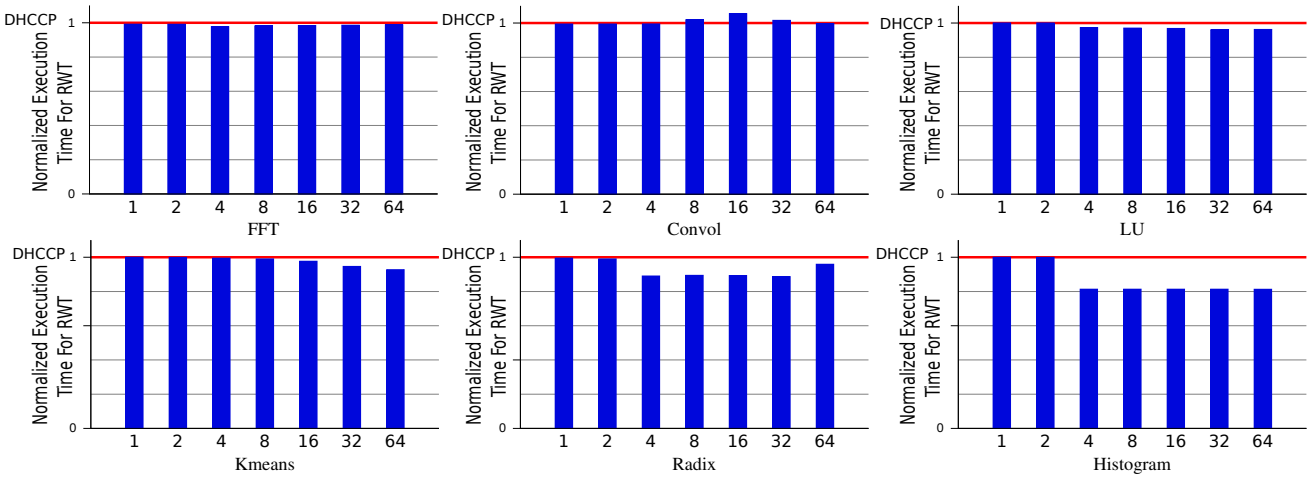

Figure 4: Execution Times (in cycles) for RWT Normalized w.r.t. Times with DHCCP

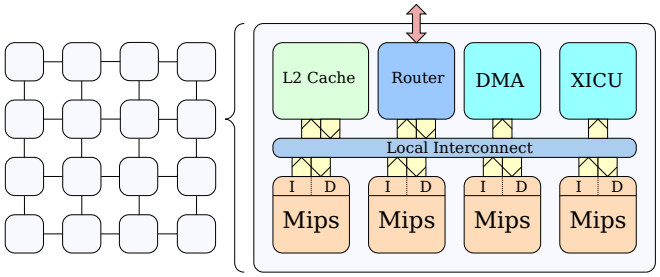

Figure 3: Overview of the Clustered TSAR Architecture used for Implementing RWT

Table I: Hardware Main Characteristics

Mesh Size	up to 4 × 4 clusters
L1 Cache Sets (I & D)	64
L1 Cache Words (I & D)	16
L1 Cache Ways (I & D)	4
L2 Cache Sets	256
L2 Cache Words	16
L2 Cache Ways	16
TLB Sets (I & D)	8
TLB Ways (I & D)	8

B. Applications

Applications used for evaluations are FFT and LU from the Splash-2 suite [23], Histogram and Kmeans from the Phoenix-2 benchmark suite [24], and Convol, which is an image filtering program performing a convolution filter in X and Y directions. Table II shows the configuration for each application.

All these benchmarks have been run over an operating system called GietVM, which is developed in our laboratory and supports the physical placement of software objects *via* virtual memory. We used this feature to spread data physically in clusters, trying to improve the locality of data accesses when possible.

Table II: Applications Parameters

Application	**Input Data**
Histogram	25 MB image (3408 x 2556)
Convol	1024 x 1024 image
Radix	262,144 keys (default)
FFT	2^{18} Complex Points
LU	512 × 512 elements
Kmeans	10,000 points

C. Measurements

Other experiments currently in review have shown the scalability of DHCCP up to 64 cores, both in performance and coherence traffic. Therefore, these experiments focus more on the comparison between RWT and DHCCP.

We compared the performance of RWT relatively to DHCCP, as well as the traffic in the interconnection networks, for reads, writes, and coherence requests. Precisely, the measures used to evaluate our two protocols are:

- **Execution time**: the execution times correspond to the parallel phase of each application.
  Execution time = timestamp of the last thread finishing the parallel phase - timestamp of first thread starting the parallel phase
- **Traffic**: in L2 caches, we implemented several counters to measure the *cost* of each type of request. This cost is a product between a number of flits and a distance, and is defined as follows:
  - For requests going from a L1 cache to its local L2 cache, we define that the cost is equal to the number of flits, because there is only one access in the local

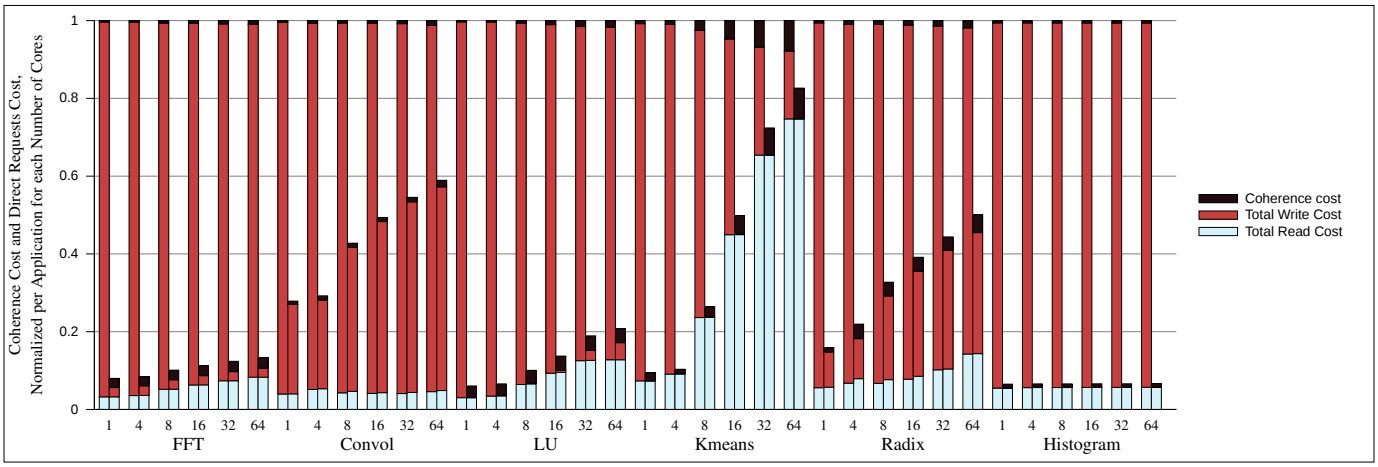

Figure 5: Comparison of Read, Write and Coherence Cost between DHCCP (Left Bar) and RWT (Right Bar), Normalized w.r.t. DHCCP total cost

crossbar.
Local cost = Nflits x 1

– For requests going from a L1 cache to a remote L2 cache, the cost includes 2 accesses to the local crossbar (an access from the local crossbar to the router and another access from the router to the local crossbar) and the hops for propagating the request from a router to another.
Remote cost = Nflits x (Nhops + 2)

The rationale behind the cost is that it should be closely related to the energy consumption, and therefore be as low as possible.

D. Results

Figure 4 shows the execution time with RWT for the 6 considered benchmarks. These times are normalized per number of cores and per application w.r.t. times on DHCCP.

For 5 out of the 6 benchmarks, RWT either has identical performances of improves them, up to 20%. Results on Convol can be explained by the nature of the application, in the sense that all the internal image buffers – 5 in total – are constituted by lines shared by different writers only (then later read by a single reader). Thus, when DHCCP only propagates writes to memory, RWT adds a coherence overhead since each line is first sent in non-coherent mode, then has its status switched to coherent, inducing a non-negligible overhead. Yet, for most applications, this case remains rare enough so that this overhead is largely balanced by RWT's overall gain.

Figure 5 shows the cost of Reads, Writes, and coherence for both RWT and DHCCP. Results are displayed normalized w.r.t. the sum of the costs for DHCCP. We can observe that all applications benefit from RWT's decrease in writes, and that the write cost is entirely removed in 4 of the 6 applications. This results in low or very low overall costs for all applications compared to DHCCP. Also, if the deletion of writes in RWT translates into a small increase in the coherence traffic (since for non-coherent lines, data is propagated in memory *via cleanup-data* requests), we can notice that the overhead in RWT's coherence cost is negligible compared to the write cost it allowed to suppress.

The rationale behind RWT is that lines do not change their state often: either they are private to a processor, which is the most common case, or they are shared once and then stay in the L2 cache long enough so that the switch cost is amortized. Private lines typically contain stack variables, but also many shared variables which are actually distributed per thread. Writes on private lines are either buffered in case of hit – which is the best case –, or propagated in case of miss – like for the write-through. To confirm that line state switches are rare events, we measured the percentage of reads and writes resulting in a line state change. Figure 6 shows the results for the same configurations as before: we can observe that this percentages for *reads* barely exceeds 3% for LU, while remaining very low for other applications: less than 1% in most cases. *Write* misses triggering a line state change are negligible. This results confirm our hypothesis that line state changes are rare and explain the good results of RWT.

These encouraging results lead us to consider integrating RWT as the main coherence protocol in the TSAR architecture.

E. Hardware Overhead

RWT adds little hardware overhead compared to DHCCP. In fact, the number of per-line bits of metadata is unchanged in the L1 cache, while it is only increased by 1 in the L2 cache for storing the NC or C state. The major overhead comes from the fact that RWT requires a cleanup FIFO in the L1 cache which contains a full cache line (64 bytes), whereas it is not necessary in DHCCP as cleanup cannot contain data. This overhead still remains negligible, meaning that RWT manages to reuse most of DHCCP hardware efficiently.

V. Conclusion

In this paper, we proposed a new hybrid hardware cache coherence protocol called RWT, which allows to reduce the write traffic while not degrading the performances of a scalable write-through coherence protocol. RWT mixes a non-coherent

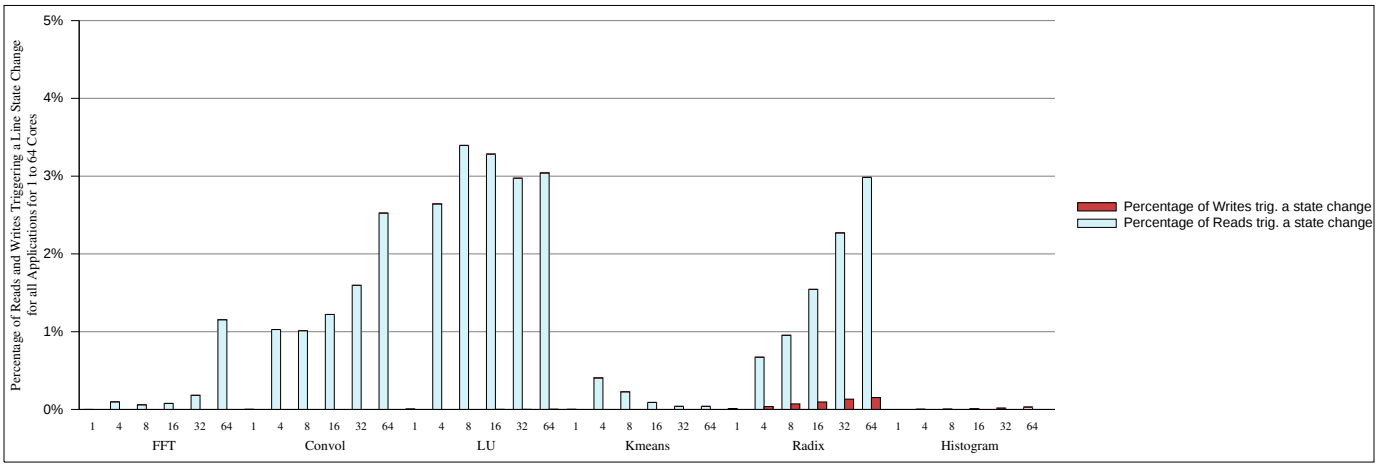

Figure 6: Percentage of Read Requests (Left) and Write Requests (Right) Triggering a Line State Change from NC State to C State

write-back strategy for non-shared lines, and a write-through coherent strategy for shared lines, the choice being made entirely in hardware.

Our results show that it reduces more than 50% of write traffic in average, while reducing execution time by 5% in average compared to the baseline protocol DHCCP. Further experiments need to be done on a larger set of applications, and for platforms containing more than a hundred cores. Indeed, RWT targets manycore platforms like the TSAR architecture. Yet, we are confident the significant reduction in cost brought by RWT will continue as the number of cores grows. Future work also include the implementation of some optimizations in RWT, especially the line replacement policy in the L2 cache, as we can take advantage of the small eviction overhead of non-coherent lines to favor eviction of such lines over coherent ones.

REFERENCES

[1] J. M. Rabaey, "Scaling the power wall: Revisiting the low-power design rules," *Keynote speech at SoC*, vol. 7, 2007.

[2] G. De Micheli and L. Benini, *Networks on chips: technology and tools*. Academic Press, 2006.

[3] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac: a 1000-core cache-coherent processor with on-chip optical network," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 477–488.

[4] C. Ramey, "Tile-gx100 manycore processor: Acceleration interfaces and architecture," *Tilera Corporation*, 2011.

[5] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, 2010, pp. 108–109.

[6] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *ACM SIGARCH Computer Architecture News*, vol. 12, no. 3, pp. 348–354, 1984.

[7] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the amd opteron processor," *IEEE micro*, vol. 30, no. 2, pp. 16–29, 2010.

[8] S. R. Garea and T. Hoefler, "Modelling communications in cache coherent systems," *Technical Report*, 2013.

[9] *TSAR: Tera-Scale Multiprocessor ARchitecture*, Available: https://www-soc.lip6.fr/trac/tsar, 2009.

[10] Y. Gao, "Generic cache controller for a massively parallel manycore architecture using coherent shared memory," Ph.D. dissertation, Université Pierre et Marie Curie (UPMC), 2011.

[11] M. Loghi, M. Poncino, and L. Benini, "Cache coherence tradeoffs in shared-memory mpsocs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 5, no. 2, pp. 383–407, 2006.

[12] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A case for software managed coherence in manycore processors," in *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10*, 2010.

[13] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[14] P. Guironnet de Massas and F. Pétrot, "Comparison of memory write policies for noc based multicore cache coherent systems," in *Design, Automation and Test in Europe (DATE)*. IEEE, 2008, pp. 997–1002.

[15] M. M. Martin, M. D. Hill, and D. A. Wood, "Token coherence: decoupling performance and correctness," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 182–193.

[16] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, and D. A. Wood, "Improving multiple-cmp systems using token coherence," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 328–339.

[17] A. Ros, M. E. Acacio, and J. M. García, "A direct coherence protocol for many-core chip multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 12, pp. 1779–1792, 2010.

[18] A. Ros, M. E. Acacio, and J. M. Garcıa, "Cache coherence protocols for many-core cmps," *Parallel and Distributed Computing*, 2010.

[19] P. Sassone, C. Koob, D. Vantrease, S. Venkumahanti, and L. Codrescu, "Hybrid write-through/write-back cache policy managers, and related systems and methods," Jul. 18 2013, uS Patent App. 13/470,643. [Online]. Available: http://www.google.com/patents/US20130185511

[20] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 93–104.

[21] The Soclib Consortium, *SoCLib: an open platform for virtual prototyping of multi-processors system on chip*, Available: http://www.soclib.fr, 2008.

[22] *SystemC Reference Manual*, Synopsys Inc., http://www.systemc.org.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. New York: ACM Press, 1995, pp. 24–37.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 2007, pp. 13–24.