Software cache coherency support on Karlay many-core architecture

Alexy Torres Aurora Dugo
ENSIMAG 2A - SLE
alexy.torres-aurora-dugo@grenoble-inp.org

Frédéric Pétrot
Université Grenoble Alpes
frederic.petrot@univ-grenoble-alpes.fr

***Abstract*—Cache memories have been in every computer since 1980, time at which the speed of the CPU surpassed the speed of the memory. Since then, this speed gap has kept increasing continuously. With the advent of new technologies and new architectures such as many-core architectures, memories hierarchies added to the processors may not be coherent anymore, making shared data access a new challenge in terms of efficiency and production cost. As a result, in many architecture hardware cache coherency protocols as we know them are no longer hardware-supported, leading to the development of software-based methods to maintain cache coherency. This article gives a comparison of different methods used in the DNA/OS operating system kernel to maintain cache coherency with a software-based approach.**

Keywords– cache coherence, many-core, algorithm, protocols, computer architecture

I. Introduction

DNA/OS is a SMP [1] based operating system kernel executed only in kernel mode[1]. Lightweight and well structured, it was made to be easily ported on multiple architectures. Already ported to ARMv7, MIPSIV, x86, Sparc V8 and Microblaze, DNA/OS is being modified to run on the new Karlay MPPA-256 Bostan architecture. The kernel has already been ported on the many-core architecture, but due to the lack of cache coherency support, data caches have been deactivated.
The K1 processor architecture is being developed by KALRAY and has the particularity to be a part of the many-core architectures. The chip is composed of sixteen clusters attached to a NoC. Each cluster contains 16 processors running at a frequency of 400MHz to 800MHz, and each processor accesses memory through a L1 data and instruction cache of 8KB[2]. There is no hardware support for cache coherence inside the cluster. The cluster also contain a 2MB L2 cache. In this article, we will only show the result obtained by loading the kernel and the applications directly in this cache, thus referring to the L2 cache as "main memory".

II. Preliminary work

A. Cache writing policies

Two cache writing policies exist. They define the way data are written in the memory and handled in the cache.
The WRITE-THROUGH access scheme allows an immediate data write back into the lower memory level once a data has been modified by a program. In a multiprocessor system, this policy minimizes the number of cache incoherence cases but may induce latency on multiple successive memory accesses.
The WRITE-BACK access scheme copies the data back to memory only when an evicted cache line is marked as dirty. In a coherent multiprocessor system, it mush also copy back the data when an other processor wants to access a data which is marked as dirty in an other cache.
The K1 architecture allows the user to set the cache writing policy. To simplify the work described in this article, we chose to set the cache writing policy to WRITE-THOUGH since this scheme induce less cache incoherences than the WRITE-BACK scheme[3].

B. Cache consistency models

Memory consistency models or schemes define the access scheme to the memory by a component. The memory access order will depend on the selected scheme. Multiple models exist but we will only describe the most important ones in this article.
SEQUENTIAL CONSISTENCY is one of the strongest memory model with the STRICT CONSISTENCY model. In this scheme, each processor has an exclusive access to a part of the memory. Even if data writes by processors do not have to be seen instantaneously by other processors, they must occur in a sequential order which should be the same for all the processors.
PROCESSOR CONSISTENCY scheme forces data writes done by two different processors on the same location to be seen in the same order than the actual writes order. Two writes done on a different location, however, do not need to be seen in the actual order by all the processors.
WEAK CONSISTENCY enforce data reads to be finished only at synchronization points. Data writes must be visible by all other processors only when necessary at synchronization points. Process scheduling, however, should be done sequentially and the order should be the same for all the processors.
RELEASE CONSISTENCY model defines a lock based data access scheme. Before each data write or read, the processor must have gone through all synchronization entry points. When reaching the last synchronization release points, all data accesses (reads/writes) must have been finished and visible by all the processors. The synchronization order is defined by the PROCESSOR CONSISTENCY model.

[1] Symmetric multiprocessor

ENTRY CONSISTENCY enforces all accesses (reads/writes) to a memory chunk to be finished and visible to the processor before it acquires the memory. With this scheme only one processor has access to a certain memory chunk.

III. COHERENCY PROTOCOLS

Most of the protocols used to ensure cache coherence are based on hardware support or, when non existent, on software support. The following sections describe the most used approaches to ensure cache coherence.

A. Hardware based protocols

1) Snooping protocols: These protocols are based on a bus interconnection between different processors. The snooping cache protocols use the cache controller of each processor connected to the bus. The controllers watch the traffic occurring on the bus and modify their state according to a finite state automaton. For instance, when a data is modified en written back in main memory, each controller will check if the data was present in their cache. If so, depending on the protocols used [2] the corresponding cache line will be invalidated or updated. The protocols may be different depending on the cache writing policy used by the processors.
These protocols are relatively simple thanks to the use of the shared bus connection that acts as a broadcast mechanism. However the method is not adapted to many-core architectures because of the use of the bus itself. Connecting multiple cores/processors to a shared bus divides the available bandwidth by the number of components to connect. Indeed, it will become inefficient to share this bandwidth by more than eight or sixteen processors.

2) Directory based protocols: Directory based protocols maintain coherency using main directories. These memories keep track of the different states (dirty, shared, exclusive, ...) of a cache line. Depending on the protocol, one or multiple directories may be used to create redundancy in order to minimize the access time to these memories. During each memory access (in case of cache-miss), the processor that makes the access has to read the directory to know where the data is stored. Then, depending on the protocol, the processor will have to invalidate a line of cache, modify or add the sate of the cache line in the directory or simply access the data.
Directory based protocols ensure a great scalability[4] in the system. Each processor must be linked to the directory but multiple bus can be used, which addresses the snooping base protocols bus bandwidth issue. When the number of cores to connect to the the directory is over sixty-four, the directory access latency[3] adds too much overhead to the method to be efficient.

3) Distributed directory based protocols: Coming from the generic directory based protocols family, distributed directory based protocols use multiple directories called banks to decrease the storage charge. Each bank has its directory containing the cache lines state concerning the nearest processors. This allows to use cache locality property and decrease directory access overhead time. However, this protocols family enforces using multiple memories and increases the hardware cost of the protocols. It is possible to lower the memory size impact by using filters to predict data existence in the directory [4].
DHCCP[5] and SPACE[6] protocols are a part of the distributed directory based protocols. They also add shared memory access pattern detection mechanism to decrease directory access time.

4) Hierarchical directories based protocols: Using different level to represent data share level. A binary tree architecture is the most used structure to store data. Each leaf is a processor and internal nodes directories. Each directory only contains cache lines information concerning caches related to processors present in its sub-tree. Searching the cache line information is accelerated thanks to this property. One directory is divided into multiple directories which reduce memory usage by avoiding memory redundancy. This method also reduce the charge sustained by one directory. Finally some protocols use data redundancy methods to allow more efficient cache line information searches. The tree structure and the data redundancy address the concurrency management overhead and allow these protocols to be fully scalable to more than 128 processors.
The SCD[7] protocol uses this architecture to ensure cache coherence.

5) Private/Shared classification based protocols: Most of the current researches lean toward these protocols. The classification allows to manage cache line with more precision and avoid false invalidation. A generic classification scheme is given :

- Private: the data is only present in one cache.
- Shared: the data is present in multiple caches.

In this case cache coherence is only maintained for shared data. This removes overhead resulting from false invalidation when a data is supposed shared but is actually only used by one processor. A software-hardware hybrid implementation, combined to the ability to manage shared and exclusive data allows to get efficient results[8] but may be complex to design[9]. It is also important to note that some protocols have a page granularity instead of the usual line granularity management. Protocols such as VIPS[10] and SVIPS[11] use these principles.

[2] MSI, MESI, MOESI, ...
[3] The directory needs to be accessed by one processor at a time.
[4] Bloom filters have been used for this purpose.

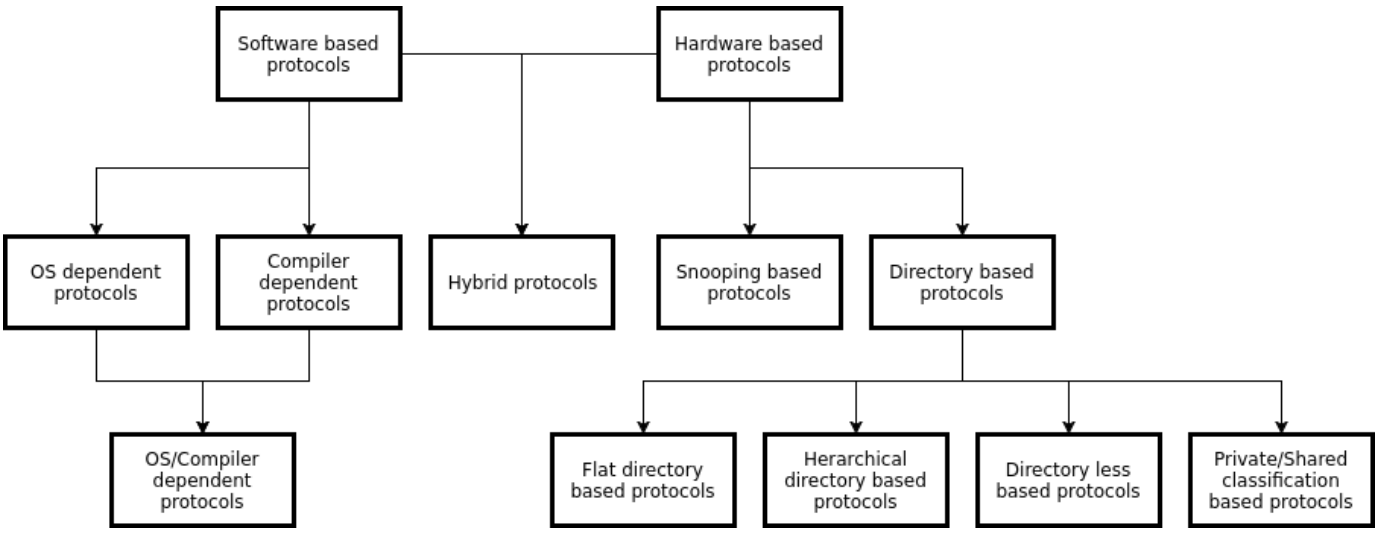

Fig. 1. Cache coherence protocol hierarchy[12]

B. Software based protocols

Multiple works have been done on software based cache coherence protocols. Most of the solutions designed are inspired by hardware protocols, taking the hardware finite state machine to the software level in order to manage the cache. The compiler and operating system are usually involved in the protocol. However using software protocol requires special access to the memory (un-cached access) and optimal design to avoid the software processing overhead. SVIPS is an example of these transformation from hardware to software. Software protocols usually use the page granularity to enforce coherence between multiple data, thus based on highly intensive operations such as page comparison or dictionary search. Moreover the memory usage overhead of these methods may be an issue on some architectures. The main memory overhead comes from the copy of a page in cache, one is modified and the other copy will allow to compare the modifications done since the page was retrieved in the cache. Comparison is usually done on cache eviction or when modifying shared/dirty cache lines. In our case, the K1 cluster architecture has only 2MB that we can use to run user software and DNA/OS. As a result, using these memory-expensive protocols is not possible.

C. Hardware/Software protocols

In order to take both advantages of the software and hardware protocols, "hybrid" protocols have been designed. Flask[13] is an example of such a protocol which uses snooping methods to ensure coherence aside with directory based methods to address the scalability issues of the snooping based protocols. It is also frequently possible to see protocols being able to vary their granularity [5] to increase their performances.

D. MPPA-256 constraints

The K1 architecture is composed of 16 clusters. Our work only concerns one cluster. Each cluster is composed of 16 processors with 8KB level 1 data/instruction cache. The level 2 cache is referred as the main memory in which DNA/OS and the user software are injected. This memory has a capacity of 2MB which is the most difficult constraint to overcome when designing the cache coherency protocol. The number of processors in the cluster being relatively high, the protocol should also be scalable.

IV. DNA/OS PROTOCOLS

To compare software based protocols, we used the modular structure of DNA/OS to implement different protocol schemes and run the benchmark programs on the Bostan MPPA-256 from KALRAY. Each protocol ensures cache coherence at cluster level. The write policy selected for the comparison is a write-through policy. Every time a data is modified, the cache line containing the data will be put in a write-buffer present in the K1 architecture and eventually written to memory, without freezing the processor.

A. Lock based protocol

This protocol is based on the entry/release memory consistency model. Each time a lock is acquired, the kernel will invalidate all the L1 data cache lines of the processor that acquired the lock. During the lock release process, all the L1 cache lines will be purged in the main memory [6]. This process is done by purging the write-buffer.
The main limit of this method is that it does not ensure coherence with programs allowing race condition on data manipulated by multiple processors concurrently.

B. Trace analysis

Based on the work of Cunha [3], that advocates to simply invalidate a shared line prior to reading it, knowing that the write will eventually reach memory, the code of DNA/OS was modified to maintain cache coherence in the kernel. Each

[5] chunk, page, line

[6] The cluster L2 cache

time a potential cache coherence violation is detected in the execution trace of the kernel, a cache line invalidation/purge instruction has to be added to the source code.

C. Software buffer based cache coherency protocol

The protocol we implemented is inspired by the method proposed by *J. Cai et A. Shrivastava*[14].
To ensure cache coherence, we rely on three elementary operations during program execution: lock acquisition, data writing and that lock release. As previously stated, this method will not ensure coherence in a program that allows race conditions on data during its execution. The protocol scheme is described in FIGURE 2.

- Lock acquisition: before acquiring the lock, the system will invalidate the cache line that contains data modified by other processors before the lock is actually acquired.
- Data writing: when a data is modified by the processor, a WRITE-NOTICE is created and inserted in the write-notice buffer. The following process will be further described.
- Lock release: before releasing the lock, all cache lines containing dirty data will be purged to the main memory.

Two data structures have been created to register dirty data addresses that need to be purged when the lock is released. These structures are also needed during the lock acquisition process in order to check whether a cache line has to be invalidated or not.

- The write-notice is an entry containing the address of the modified data. The structure also contains an invalidation bit vector. The vector allows the protocol to keep track of which processor has already invalidated the data in its cache since the last modification. This method avoid falses invalidation during lock acquisition process. The write notice structure is a node in a linked list, allowing to store them in the memory.
- The write notice buffer is contained in the main memory and accessed in an uncached way. This is not an actual entity but a linked list header pointing to the first write notice.

1) Data writing detection: To create a write notice each time a data is modified in the cache, we need to detect data writing during execution. The first method to achieve this is to modify the compiler to add a call to the write-buffer creation function after each memory writing. The second way is to ask the user to add this call to the function each time a data is modified in the program.
One may also want to detect data writing without intrusion in the user's code. This is possible thanks to the MMU[7]. Each core of a cluster has its own MMU which should allow the kernel to detect every data access and decide to create a write notice when it is needed. This method, however, is not realizable on the K1 architecture.
Un-cached data writing need to be registered too (to invalidate the data if needed) but will not require cache line purge when releasing the lock.

[7]Memory management unit.

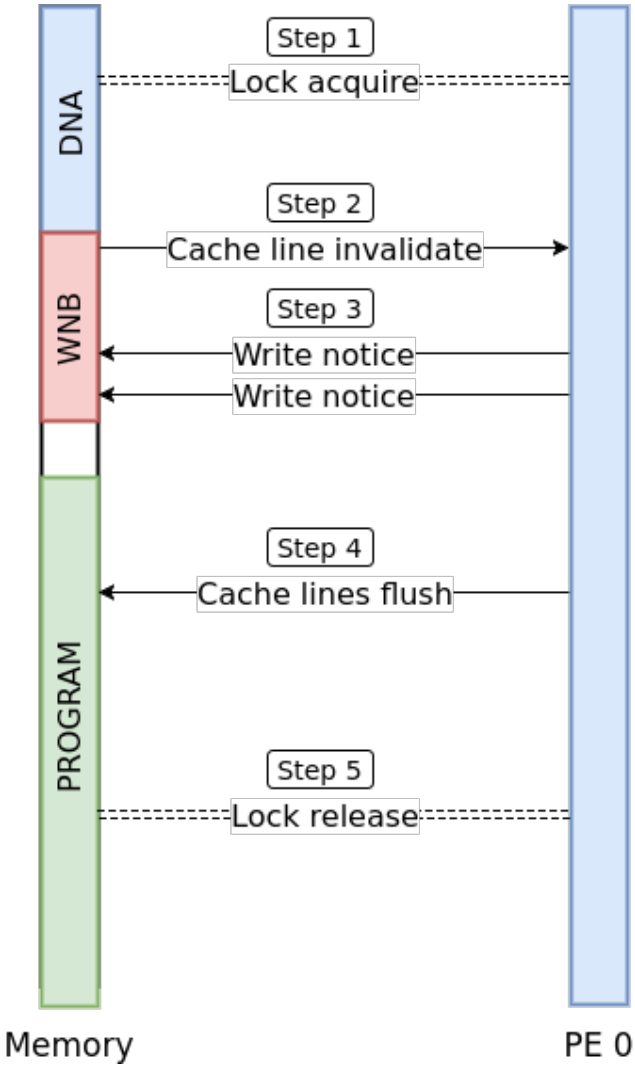

Fig. 2. Protocol processing steps

The method we selected to perform the comparison is the second one, where the user need to call the write-notice create function after each data modification.

D. First implementation of the protocol

```
struct write_notice_t
{
    uint32_t data_addr;
    uint16_t updated_cpu_mask;

    write_notice_t * next;
}
```

- DATA_ADDR is the address of the data modified by the processor.
- UPDATE_CPU_MASK is a 16 bit mask where the i^{th} is set to 1 is the i^{th} processor already updated its cache.
- NEXT is the next node of the buffer.

When booting, DNA/OS initialize the write buffer by setting the pointer to the WNB. This pointer is shared by all the processors of the cluster. We added space restriction to the buffer to limit the memory impact of the protocol. In this first version, the WNB can not exceed 128KB.

1) Lock acquire: After acquiring a lock, the processor will walk the buffer to potentially invalidate cache lines. Thanks to the K1 architecture, asking to invalidate a data which address is not contained in the cache will not invalidate any line. Thus we simply walk the buffer, asking the processor to invalidate every addresses contained in the buffer. Once the line invalidated, the processor will set the i^{th}bit of the update mask accordingly to its identification number. One of the possible enhancement to this process would be to directly update the data instead of invalidating the line as proposed in [14]. However the K1 architecture does not allow this update. Once the processor finished the invalidation process, the rest of the program is executed.

2) Cache modification: Once the processor acquires the exclusivity on the data, each cached data writing will be reported. A *write-notice* is created and added to the buffer. If the data is already contained in the buffer, its bit mask will be set to 0 far all the others processors. The *write-notice* is also put on the top of the buffer. The case when the buffer is full will be described later in this article.

3) Lock release: On lock release, the write-buffer will be purged to the memory to make all the modifications visible to the others processors. FIGURE 2 show the whole update scheme used by the protocol.

4) Write-notice buffer management: The write notice buffer is a linked list containing all the *write-notice* generated. We choose to favor memory saving instead of computing time since the memory on the MPPA-256 is limited.
Due to this lack of memory, we must limit the size taken by the buffer in main memory. To address this issue we select an arbitrary size limit of 128KB. This size corresponds to approximately 1600 *write-notice*. We also note that the bigger the write buffer is, the longer the buffer walk will be during *write-notice* generation.
The *write-notice*s accumulate in the buffer during the execution. We need to evict them when the buffer is full or when all the processors updated the data in their cache.

- Adding a Time-To-Live to the *write-notice* based on the number of acquisitions is a first solution. Removing all the *write-notice* exceeding this TLL when a processor acquire a lock.
- If the buffer exceed the size limit fixed by the user, we choose to remove the 25% oldest *write-notice* base on an LRU scheme. We use this method in the proposed version of the protocol.

When evicting a *write-notice* the protocols needs to notify each processors that did not updated its cache. To do that, the processor detecting the buffer overflow will generate inter-processor interrupt for each processor of the cluster. When receiving an interrupt, the processor will walk the last 25% *write-notice* contained in the buffer and apply the process explained when releasing a lock. Once done, each processor will wait for a barrier. Once all the processors attained the barrier, they can executed the task they were processing before the interrupt. The FIGURE 3 describe the buffer management process.

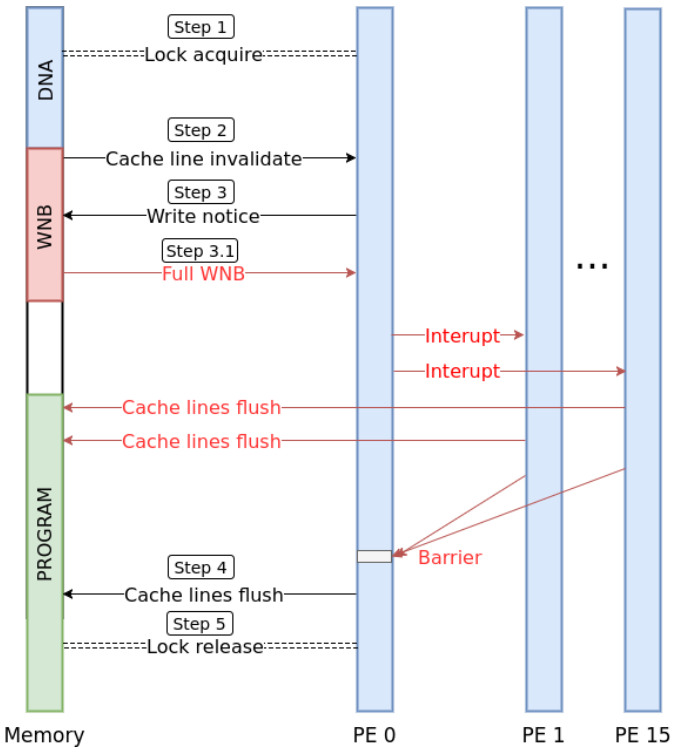

Fig. 3. Full WNB management

E. Second implementation of the protocol

In this second version, we choose an other structure to represent the *write-notice*:

```
struct write_notice_t
{
    uint32_t data_addr;

    write_notice_t * next;
}
```

- DATA_ADDR is the address of the modified data.
- NEXT is the next *write-notice* in the buffer.

The main modification of the protocol comes from the lack of bit mask to know which processor already update its cache. We do not use a public *write-notice* buffer containing up to 1600 *write-notice* but 16 private buffers. One private buffer contains up to 125 entries. With this solution we save 16 bits per *write-notice* and lessen the buffer walk time.
Further work may lead to searching how to put the *write-notice*

in the correct private buffers to avoid false invalidation of the cache and reduce the buffer walk time.
With this protocol version, each lock acquired leads to the private buffer clearance. This avoid useless processor interrupts, reducing the buffer size overflow occurrences. Indeed, if one buffer is to overflow, we only need to interrupt the buffer's owner.

V. Results

To compare the result obtained with the different protocols we developed two simple programs. The aim of these was to validate the protocols and analyze their execution.
Due to a lack of time only two programs were used to analyze our protocol. Further tests are done to compare the trace analysis approach with the lock based approach.

A. Write notice based protocols

The two first programs are:

- Mat_mult, multiplying two 100x100 matrices with up to 16 threads (one per processor). The results are showed in Figure 4.
- RGB2YUV, converting RBG format images to YUV format with up to 16 threads (one per processor). The results are showed in Figure 5.

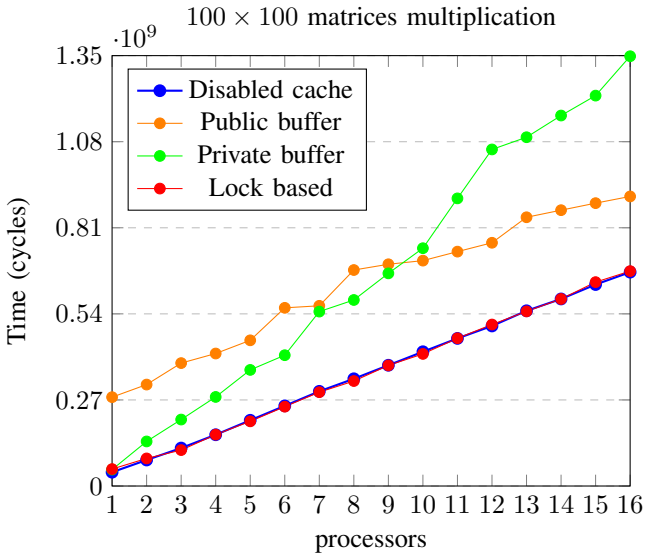

Fig. 4. 100 × 100 matrices multiplication

B. Trace analysis VS Lock based

The following results were obtained with two programs from the Splash2[15] benchmark. Due to the highly constrained environment of the MPPA-256 two other programs (OCEAN from Splash2 and ParallelMJPEG) were to be ported but remain a work in progress.
Both Water-Spatial in Figure 6 and Water-Nsquared in Figure 7 are ran 20 times. The graph represent the mean time of all the executions.

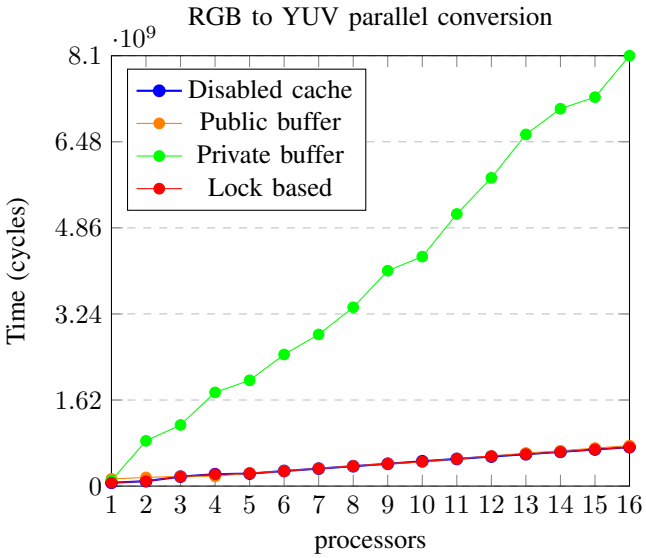

Fig. 5. RGB to YUV parallel conversion

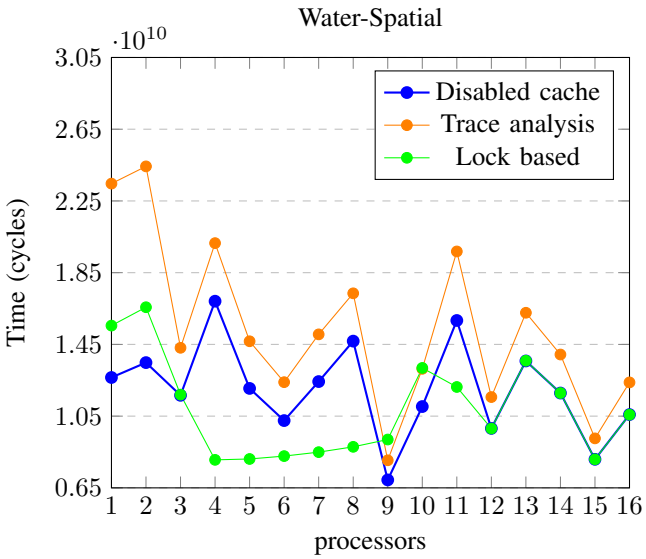

Fig. 6. Water-Spatial

VI. Discussion and further enhancements

These results show the inefficiency of the protocols. These performance issues are explained by the very specific architecture of the MPPA-256 that does not fit the DNA/OS structure well. But the tests also allowed to validate the protocols.
To address the performances issues due to the structure of DNA/OS we modified its code to limit the overhead generated by the kernel. The second set of tests realized with Water-Spatial and Water-Nsquared show better result and even slight amelioration when enabling the caches.
To address the kernel overhead issue further work may rely on new lightweight kernels specially developed for the K1 architecture. This method would allow to analyze the cache protocols efficiency more precisely.

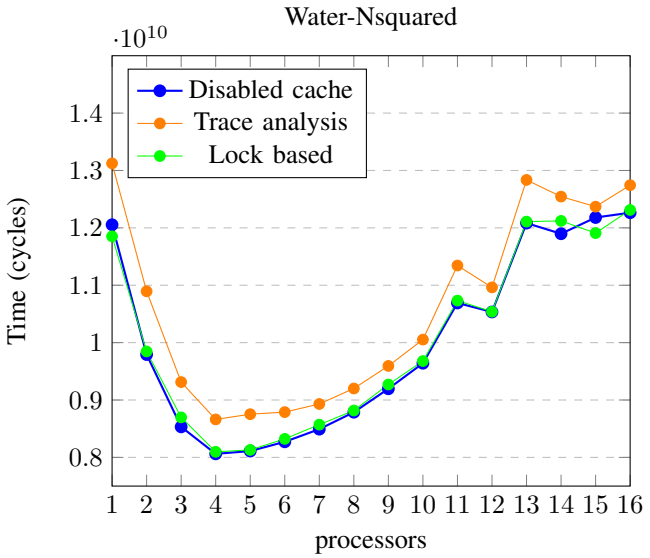

Fig. 7. Water-Nsquared

VII. Conclusion

This article presented the work achieved to develop and analyze multiple cache coherence protocols designed for the many-core architecture developed by Kalray[2]. It shows the increasing complexity of developing methods to ensure cache coherency at the cluster level and the incapacity of state of the art methods to produce efficient execution time on general purpose operating systems.

References

[1] Xavier Guerin and Frédéric Pétrot. A system framework for the design of embedded software targeting heterogeneous multi-core socs. In *20th IEEE International Conference on Application-specific Systems, Architecture and Processors*, pages 153–160. IEEE, 2009.

[2] B. D. de Dinechin, R. Ayrignac, P. E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[3] Marcos Aurélio Pinto Cunha, Omayma Matoussi, and Frédéric Pétrot. Detecting sw cache coherence violations in mpsoc using traces captured on virtual platforms. *ACM Trans. Embedded Comput. Syst.*, 16(2):30:1–30:21, 2017.

[4] Xiuzhen Lian, Xiaoxi Ning, Mingren Xie, and Farong Yu. Cache coherence protocols in shared-memory multiprocessors. In *International Conference on Computational Science and Engineering*, ICCSE '15, 2015.

[5] H. Chtioui, R. B. Atitallah, S. Niar, J. L. Dekeyser, and M. Abid. A dynamic hybrid cache coherency protocol for shared-memory mpsoc. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 3–10, Aug 2009.

[6] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. SPACE: sharing pattern-based directory coherence for multicore scalability. In Valentina Salapura, Michael Gschwind, and Jens Knoop, editors, *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010*, pages 135–146. ACM, 2010.

[7] Daniel Sanchez and Christos Kozyrakis. Scd: A scalable coherence directory with flexible sharer set encoding. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[8] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. *SIGARCH Comput. Archit. News*, 39(3):93–104, June 2011.

[9] L. I. Kontothanassis and M. L. Scott. Software cache coherence for large scale multiprocessors. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 286–295, 1995.

[10] Alberto Ros and Stefanos Kaxiras. VIPS: Simple directory-less broadcast-less cache coherence protocol. Technical Report 2011-029, Department of Information Technology, Uppsala University, November 2011.

[11] Magnus Norgren. Software distributed shared memory using the vips coherence protocol. Master's thesis, Uppsala University, Department of Information Technology, 2015.

[12] TSB Sudarshan Parvathy N, Bhargavi R Upadhyay. Cache coherence: A walkthrough of mechanisms and challenges. In *International Conference on Electrical, Electronics, and Optimization Techniques*, ICEEOT '16, 2016.

[13] *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*. IEEE Computer Society, 2015.

[14] Jian Cai and Aviral Shrivastava. Software coherence management on non-coherent cache multi-cores. In *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID 2016, Kolkata, India, January 4-8, 2016*, pages 397–402. IEEE Computer Society, 2016.

[15] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995.