Approaching Cache Coherence in Shared-Memory Multiprocessor Systems

Amitabh Yadav
M.Sc. Computer Engineering
Delft University of Technology
Student No. 4715020
Email: A.Yadav-3@student.tudelft.nl

Shivanand C. Kohalli
M.Sc. Embedded Systems
Delft University of Technology
Student No. 4751116
Email: S.C.Kohalli@student.tudelft.nl

Umeer A. Mohammad
M.Sc. Embedded Systems
Delft University of Technology
Student No. 4748549
Email: U.A.Mohammad@student.tudelft.nl

***Abstract*—The concept of Cache Coherence is the main aspect of functionally-correct faster-performing multicore processors. Coherence protocols can be software, hardware or a combination of both. The snooping based protocols are faster and efficient but they do not scale for large core count. Whereas, the directory-based protocols scale to large counts but are complex to analyse and implement, architecturally. This paper discusses some of the modern (recent) implementations of both snooping and directory based protocols for cache coherence. The modern implementations employ the traditional architectural approach with proposed solutions of optimisations and improvement. The effectiveness of these protocols is studied based on different requirements in varied multicore platforms, such as embedded system and heterogeneous SoC; and a critical comparison of these protocols is presented.**

I. Introduction

Today, the chip makers such as Intel, AMD and IBM are making chips with multiple processor cores. These chips are variously called chip multiprocessors (CMP), multicore chips, and many-core chips [1]. Multicore chips are designed with shared memory to simplify parallel programming. Cores have their own (multi-level) cache that allows them to fetch data faster. This causes a problem of incoherent data. Cache coherence is maintaining uniformity of shared memory data stored among CMP core caches. This lies at the core of correct functionality of shared memory multicores and thus, modern computer architectures must address cache coherence and design problems.

Cache Coherence can be approached by two methods: snooping and directory-based.

In the snooping method, each cache monitors the address lines for access to memory locations that they have cached. Snooping protocols are faster but they do not scale for large core counts. They are of two types: update-based protocols and invalidation-based protocols. Of these two, the invalidation based protocols are faster and more robust. Cache Coherence protocols such as MSI (Modified-Shared-Invalid), MESI (Modified-Exclusive-Shared-Invalid) and Dragon protocol are used in shared memory CMP architectures help assure correctness of shared data cached within each processor. For example, PowerPC755 from IBM implements MEI protocol, Intel IA32 supports MESI and AMD64 supports MOESI (Modified, Owned, Exclusive, Shared, Invalid) protocol. ARM, in 2011, proposed the AMBA 4 ACE for coherence handling in SoCs. [1]

In directory-based protocols, a common directory stores the shared data and this directory maintains the coherence between caches.The directory-based coherence protocols are scalable for multiple cores but are more complex to analyse, validate and implement. The complexity of directory-based protocols is due to the directory indirections that require multiple controllers to interact through the on-chip interconnect. This further adds an overhead to the directory based protocols. [4] [5]

With semiconductor technology reaching to its absolute miniaturisation levels, multicore processors provide an extension to further stretch performance improvement of computing systems. Cache Coherence is an absolute necessity in all multicore processor systems. Apart from homogeneous general-purpose multicore processors, CMP are developed for various forms of computing platforms such as Embedded Systems, System-On-Chip (SoC), and Graphics Processing Unit (GPU). And maintaining cache coherence in CMP systems increases the need for better protocols and addressing the elevated power consumption due to these protocols.

The implementation strategies and modern architectural optimisations of traditional protocols of snooping and directory-based are discussed. Various parameters for coherence research arise with different multicore platform; and factors such as area and energy efficiency is also considered along with performance. This aim of this paper is to discuss some of the numerous solutions to achieve coherence of shared data by comparison of the solutions and speculate a model for future multicore systems with larger caches and more number of cores.

This paper investigates both the cache coherence methods i.e. snooping and directory-based method by performing a study on five research papers. Section I Introduction, introduces the concept of cache coherence, its types and the research question. Section II discusses different proposed solutions to cache coherence on different multicore platforms. Here, various methods of optimisations on traditional coherence protocols are discusses. Next Section III, presents a critical comparison of these different methodologies and finally we conclude our work in Section IV.

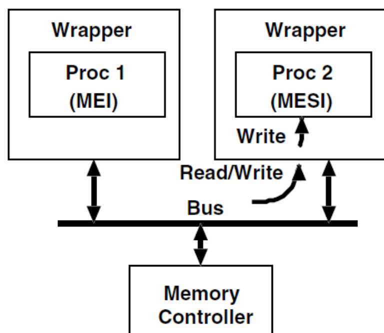

Fig. 1. Method to remove shared state [1]

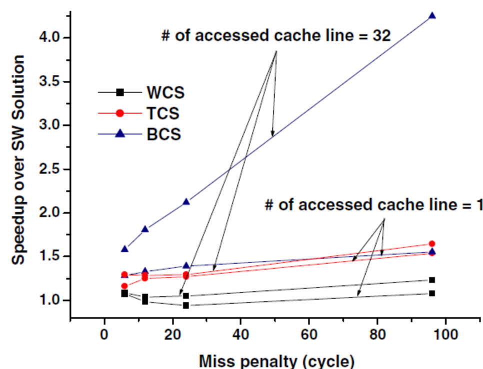

Fig. 2. Results according to miss penalty [1]

II. Description of the Evaluated Solutions

This section discusses the different solutions for cache coherence in different multicore platforms such as embedded systems and SoC. These are modern (recent) implementation and optimised implementation to get better performance in a multicore system. We describe the individual theory, implementation methodology and performance on different benchmarks.

A. Cache Coherence in Heterogeneous CMP [1]

In high performance workstations employing homogeneous processors, a straightforward approach of integrating the processors on a single bus is employed. This is easy as the bus protocol and the cache coherence protocol are compatible. However, specific computing capability is required in SoC due to integration of heterogeneous processors with different instruction set architectures integrated on a single chip. In such applications, one general purpose processor or a digital signal processor (DSP) is not sufficient in managing the entire system. In this solution, the incompatibility of bus interfaces and cache coherence protocol is addressed by proposing such a hardware/software methodology. This approach is based on the invalidation-based protocol for cache coherence.

In heterogeneous CMP systems with distributed shared memory, a directory based cache coherence can be used. Directory based protocols can address the inter-cluster issues of coherence. However, snoop based protocols are unable to address the intra-cluster coherence as coherence and bus protocol for each processor in heterogeneous CMP is distinct. In SoC, a design concept of wrapper is proposed that can help address the problem of incompatible bus protocol.

This solution studies integration of invalidation based protocols by simplifying the approach to a two-processor platform. The processors with different cache coherence protocols need to restrict the use of entire protocol states to become compatible with each other and only the states that are common are implemented. For example, if we integrate two processors with MEI and MESI protocols, the coherence protocol should be MEI. The paper further discusses the protocol integration methods for four major protocols such as MEI, MSI, MESI and MOESI with varied cases.

Here we describe the MEI with MSI, MESI, or MOESI. In this case, the proposed method is the remove the shared state. Figure 1, depicts the method by using *wrapper*. The transition to the shared state occurs when a read transaction is observed on the bus by the snoop hardware of cache controller. We can remove this shared state by simply converting the *read* operation to a *write* operation. This should be done inside snooping processors wrappers. And for the memory controller to access the correct memory, it must be informed of the operation.

For Example, MESI protocol. In this protocol, there are 3 ways to reach the S state. These are: (1) I to S, when read miss occurs and shared signal is active, (2) E to S, when snoop hardware sees read operation for clean cache line, and (3) M to S, when snoop hardware sees read operation on dirty (or modified) cache line. To remove the S state, the wrapper deasserts the S state by informing the snoop caches of writes for read operations, thus removing S state completely. So, (1), (2) and (3) cannot occur. Using the same approach, the MSI and MOESI are also reduced to MEI.

A case study of this implementation is presented using commercially available embedded processors: PowerPC755 (supports MEI protocol), Intel486 (supports modified MESI protocol) and ARM920T (no cache coherence support). In the former case, wrappers are needed for read operation conversion. And conversion of protocols between the processor buses and ASB is also done by wrapper, applicable in both cases.

To test this method, simulations were performed using micro-bench programs that runs one task on each processor with Best-Case-Scenario (BCS), Typical-Case-Scenario (TCS) and Worst-Case-Scenario (WCS). Figure 2 shows the results, a performance vs miss penalty (memory access time). As miss penalty increases, performance difference also increases in favour of this methodology.

Using commercial embedded processors, results were obtained by simulation. 57 % performance improvement was observed for low miss penalties and a whopping 324 % performance improvement was observed for higher miss penalties.

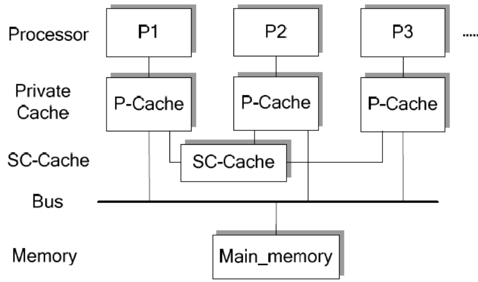

Fig. 3. Architecture of CMP with SC-cache [2]

B. Cache Coherence Protocol with SC-cache [2]

A new kind of cache coherence protocol has been introduced for CMP, called CSC (Coherence with Share Coherence-Cache) and is based on read-through and write-back mechanism. This leads to reduction in number of bus transactions and improves the efficiency of processor data access up to certain extent.

The architecture of SC-cache is describes its location between the private cache and the bus with small capacity. This protocol has been implemented using very small capacity private cache (typically between 4 KB to 64 KB) and the SC-cache size is further smaller (1 KB). The LRU (Least Recently Used) algorithm is used to increase the utilisation of SC-cache. It does so by replacing away the least recently used blocks.

A combination methodology of write through and write back has been adopted in this CSC coherence protocol. This included four states: Private-Invalidate (PI), Private-Exclusive (PE), Private-Dirty (PD) and Share-Shared (SS). There kinds of cache have been described: Private Cache, the current processors cache, Remote cache, refers to cache of remote processor, and the SC-cache. The architecture of CMP with SC-cache is shown in figure 3.

A description of the four states is as follows:

- Private-Invalidate (PI): The copy in the private cache of a core is not consistent with the main memory or copies in the other caches.
- Private-Exclusive (PE): The private cache copy has not been modified and is coherent with the main memory. However, other caches do not contain this copy and it is the only exclusive valid copy.
- Private-Dirty (PD): The private cache copy is the only valid copy and has been modified several times. No other copy is present in the main memory or other core caches.
- Share-Shared (SS): The private cache copy is consistent with the copy of main memory and this state is stored in the SC-cache, the memory which is shared by other cores. The processors therefore, read directly from the SC-cache.

The protocol implements two more states: shared and non-shared. Shared means remote cache has the copy of the request, non-shared is not having a copy. Further the functioning of this protocol works on two kinds of requests: Processor Read and Processor Write. The state transition diagram of the CSC protocol is shown in figure 4.

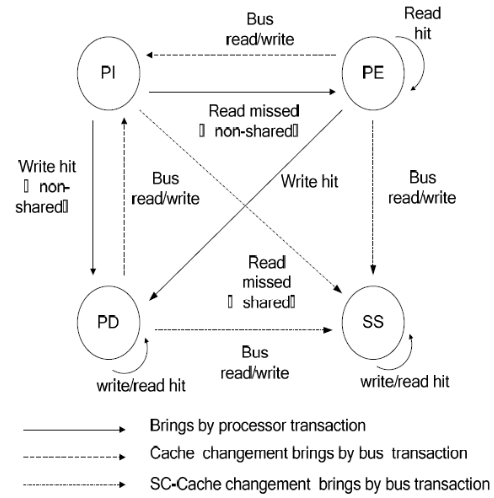

Fig. 4. State Transition Diagram of Local Cache and SC-cache [2]

To verify the correct working of the protocol, SimOS, a machine simulation environment is used. The cache section of this hardware and software section of this environment is modified and CSC protocol is realised. To measure performance, the CSC protocol is simulated against the MESI and Dragon protocol using the SPEC95 benchmark. The mgrid, applu and apsi programs are used. Execution time was measured to analyse the performance of the 3 protocols on the three benchmarks.

The CSC cache coherence protocol was able to optimise the overhead thus upgrading the storage performance. However, there was no reduction is cache miss. The total execution time was reduced by nearly 10 % compared with MESI and Dragon. This indicates that CSC approach is more effective than traditional cache coherence protocols.

C. Energy Efficient Cache Coherence for Embedded Systems [3]

Snoop cache coherence protocols try to solve the coherence problem, however for an embedded system they can be considered to be power hungry.

Snoop-cache coherence protocol scheme does not assume anything about the application structure, and thus considers that shared data can be accessed by anyone. However, this means that for every cache miss all the caches will be probed, this causes excessive power consumption. Now, if the information regarding which shared memory is accessed by which task is known to all the snoop controllers, during a cache miss only the corresponding cache can be probed sharing the same memory region.

Traditionally, the data shared between the parallel tasks is known to the application software, but this information is not penetrated to the hardware level. At the hardware level, the memory systems handle all the references to the memory generically assuming the tasks can access all the locations. The proposed methodology makes available this data of shared

memory utilization between the parallel tasks to the hardware. Now, a snoop controller with the help of this data can refine the memory references only relevant to itself.

To get this information down to the system software, the developer during the application development must inform which of the global arrays, for example, should be treated as shared memory. This can be achieved by passing the reference of this global array and its size at the time of creation of the new thread. After this task is created, this information is thus known to the operating system. Now the memory manager which actually allocates data in physical memory classifies the shared arrays to the set of physical memory frames. It then assigns a unique ID for each shared memory region. This information is further used by the hardware for filtering the requests.

The hardware would need to store the shared memory regions utilized by its corresponding processor nodes. This information can be retrieved either from the operating system or the thread library when the task is scheduled. It can be stored in a simple register, a bit 1 or bit 0 at the ID equal to bit index, will inform whether the task accesses that shared region or no. Whenever a request is raised the snoop controller just checks the region ID with its stored register at bit index ID. If the bit index corresponding to ID is 1 it does a cache probe else it is not required.

This proposed solution was tested with the benchmarks chosen from SPLASH-2 benchmark suite The amount of energy consumed per access is measured using CACTI tool. Enerrgy savings of about 40% on an average was observed, which is quite considerable.

This paper proposes a low power cache coherence protocol for multiprocessor shared memory-based embedded systems. It utilizes the fact that in an embedded application knowledge of shared memories between tasks can be conveyed to the hardware. The proposed methodology is also cost and area efficient and can thus be a productive solution for the modern embedded applications.

D. Dynamic Self-Invalidation [4]

Directory based coherence protocols send invalidation messages to appropriate processor nodes. The processor, upon receiving these messages invalidates its copy of data and sends a response back to the directory. The idea behind Dynamic Self-Invalidation (DSI) is to reduce cache coherence overhead in shared memory CMP by removing the invalidation messages. This is possible by making processor automatically invalidate its local data copy before a conflict occurs due to another processor's access. The solution for coherence presented here is a hardware directory based write invalidate protocol. Self-invalidation helps reduce latency and bandwidth required for requests of conflicting messages.

The DSI approach implementation is carried out by responding to the cache about the dynamically identified invalidate blocks in response to a cache miss. This causes the cache controller to later self-invalidate the selected block. It is also important that this self-invalidation takes place at the appropriate time otherwise it may lead to unnecessary cache misses and thereby degrading the performance.

Two invalidate blocks identification methods are introduced: additional directory states and version numbers. Also, two techniques for self-invalidation using cache controller is presented: a FIFO buffer and selective cache flushing at synchronization.

The DSI method for coherence performs very well when there is significance coherence traffic. Moreover, the DSI method eliminates both the invalidation and response (acknowledgement) messages by letting the processor nodes to get the cache block copy without updating the directory. It is presented that there is a little performance increase in most benchmark programs however, by combining DSI and weak consistency, one can get rid of about 50-100 % of the invalidate messages, improving the performance upto 26%.

The idea behind dynamic self invalidation is similar to other forms of self-invalidation: to ensure that data is present at the home node when another processor requests to access it. But, DSI differs from other because the self-invalidation is performed by the coherence protocol itself and no programmer intervention is required. The write-invalidate protocol occurs in three steps: (1) identifying the invalidate cache block, (2) invalidating, and (3) acknowledgement (if needed). The DSI, in contrast, speculates the invalidate blocks but invalidation is scheduled for a future time.

Identifying the blocks to invalidate (invalidate blocks) is the act of speculating if a particular block will be invalidated in the near future. To perform this identification, a hardware methodology is presented. One way is by the directory controller to identify the invalidate blocks by keeping a history of the data share pattern. This information can help invalidate blocks in future. Similarly, other way is by the cache controller to identify these invalidate blocks by maintaining a similarly history and using it while responding to a cache miss. The cache blocks that suffer conflicting access i.e. the blocks that might need invalidation are the main candidates for self invalidation. The implementation of the identifying blocks method is done by two methods: Additional States and Version Number. The additional state method uses four additional states to find the invalidate blocks while in version number method the directory keeps a version number for every block and increments it with every processor request.

The second step of self-invalidation can be performed by software, hardware or a combination of the two. The node that is caching must make a record of the identity of invalidate blocks for self invalidation in the near future. Two hardware methods have been presented namely, the FIFO buffer which invalidates the blocks on falling out from the buffer, and using custom hardware to self-invalidate at the time of synchronisation operation. FIFO implementation requires additional memory. In this implementation, the cache controller maintains the FIFO with identified invalidate blocks and the invalidation is performed when a new entry comes into the FIFO. Further, if synchronization operation are identified by the cache controller then the FIFO can be eliminated and entries flushed after one

or more synchronization operations.

The result of this implementation has been evaluated by performing the DSI under sequential consistency and weak consistency. The benchmarks, Barnes, EM3D, Ocean, Sparse and Tomcatv are used to measure the performance. DSI with sequential consistency had little effect on the execution time however, upto 41% of reduction in the execution time was noted under different conditions of cache size and network latency. Further, with weak consistency one can expect elimination of upto 26% of the invalidation and acknowledgement messages. Therefore, DSI is a general technique that can be implemented to software, hardware or both and for CMP system with large cache size and large number of conflicting messages proves very effective in reducing execution time.

E. Dependable Cache Coherence [5]

This solution proposes a combined approach of directory based cache coherence (DirCC) protocol and the execution migration (EM) based protocol and is referred to as dependable cache coherence architecture (DCC). In EM protocol, we have only one copy of data cached at any location in the processor and when a thread needs access to a data that is not available in the local core, the execution migrates to the memory location and continues there. DirCC ensures error resilience while EM does not require directories as no modifiable data is shared. The DirCC and EM protocols can co-exist on the same CMP and the solution presents the architectural implementation to efficiently move from one protocol implementation to another.

DirCC: The directory based protocol works by fetching the data from the main memory to local cache when the instruction stalls. This method ensures fault tolerance and exclusivity of the data. Average Memory Latency (AML) is the metric used to measure performance of the DirCC protocol and is defined as follows:

$$AML_{DirCC} = cost_{\$access,DirCC} + rate_{\$miss,DirCC} \times cost_{\$miss,DirCC}$$

Rate and *cost* primarily contribute to the AML of DirCC protocol.

EM cache coherence: In this case the execution is brought to the data where it is originally cached, referred to as *home* of that data. The advantage of EM is that it needs no directory and hence no overheads. It efficiently makes use of spatial locality but is bound by register values for instance of using temporal locality. Further, the EM performance is bound by the number and cost of migrations. The average memory latency (AML) for EM protocol is defined as follows:

$$AML_{EM} = cost_{\$access,EM} + rate_{\$miss,EM} \times cost_{\$miss,EM} + rate_{core_{m}iss} \times cost_{migration}$$

The baseline architecture of the processor consists of processor cores having a shared address space via on-chip interconnects (figure 5). Each core has 2-level data cache hierarchy. Under private L1, DirCC manages coherence while the shared L1/L2 is used by EM coherence. The DCC enables transition between the protocols at runtime and keeps impact at the user level to minimum.

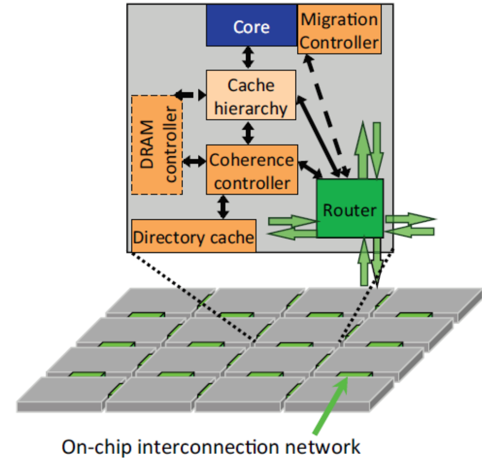

Fig. 5. Fully distributed tiled multicore [5]

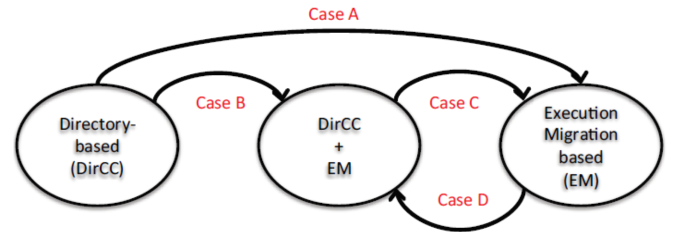

Fig. 6. Modes of operation for DCC architecture [5]

An overview of the modes of operation (figure 6) and the four transition cases are as follows:

- Case A: If the link or directory controller has a fault, transition occours to EM.
- Case B: In this case, some segments of address space is handled by DirCC and others by EM.
- Case C: If DirCC causes fault at runtime, transitions to EM.
- Case D: If migration controller is faulty and DirCC controller is working, then transition back to DirCC.
- In worst case, if both DirCC and EM are broken, OS-level reassignment of address takes place by interrupting the processor.

The transition methodology occurs as follows:

1) DirCC to EM: A special broadcast is sent to all cores by the DirCC when a page is ready for transition. When replies from all cores are received back, another broadcast is sent which updates the translation look aside buffer (TLB) to enable EM. For subsequent access the EM is implemented.
2) EM to DirCC: A cache flush of the page's address space is initiated by the migration controller of home core and a special broadcast is sent to all cores to update the TLB to disable EM. On next memory access, the DirCC takes charge.

The DCC protocol was evaluated using Graphite Simulator. The AML for SPLASH-2 LU_NON_CONTIGUOUS benchmark

that exhibits strong read/write data sharing was calculated. This gave a 1.25× advantage to EM over DirCC. Next, the SPLASH-2 RAYTRACE benchmark with read-only data sharing was use to evaluate AML. In this case there was 2.6× advantage to DirCC over EM. Thus, DCC is a novel approch to achieve better coherence without the expenses of area, power and performance.

III. Comparison

We have discusses five main solutions to achieve cache coherence in multicore platforms. These platforms were varied in architecture and implement different strategies to achieve coherence. The two main coherence protocols studied through these five implementations are snooping protocols and directory based protocols. Both protocols have the pros and cons for implementation in different architectural settings.

Cache Coherence is essential for the correct functionality of the processor. The Snooping protocols are shown to be efficient in CMP platforms with less core counts such as embedded CMP and SoC. Based on [1], [2] and [3], this is because they are faster, need simpler hardware implementation resources, and their power consumption can reduced by optimisations. As demonstrated in [1], we can extend the snoop protocol to heterogeneous cores that may or may not support coherence with architectural optimisations. This is done by introducing wrappers in the platform. Conversion of read to write and/or shared signal are used in these wrappers to maintain coherence. This not only solves compatibility issues among coherence incompatible cores but also shows promising performance improvements.

The snooping protocol sees performance improvement upto 10% compared to traditional protocols such as MESI and Dragon by adding the SC-cache and minimising the overhead of storage, as shown in [2]. However, the implementation of shared coherence cache (at the expense of hardware utilisation) is not a very suitable implementation for low core count CMP. Further, the implementation of the core on the modified 'SimOS' uses small sizes for private and SC-cache to record performance improvement which does not show promising result for large sizes of cache.

Performance metric is a function of energy efficiency as shown is [3]. The cores consume more power during implementation of snooping which probes all caches for every cache miss. A software/hardware approach by making data available prior to the snoop controller solves this problem, making embedded system CMP applications energy efficient. This method is cost and area efficient, and without any significant performance loss, upto 40% of energy can be saved.

Among directory based protocols, two optimisation approaches are presented, dynamic self-invalidation [4] and Dependable Cache Coherence [5]. Both of the approaches to cache coherence aims at the same research question i.e. optimisation of coherence to get better performance. However, the approaches are radically different. [4] describes a method to reduce the coherence overhead by removing the invalidate and acknowledgement messages; while [5] describes coherence protocols (DirCC and EM) that co-exist on a CMP and perform transition depending on the 'OS-page' being processed. [4] proposes improvement in performance as high as 41% in certain cases with good results in elimination of messages. In [5], it is shown that EM performs better on benchmark with strong read/write data sharing and DirCC is better with read-only data sharing.

The DSI method is well suited with weak consistency problems. But, has very little effect on sequential consistency problem and is not very well suited in large core count CMP. It, however, is a promising method for further research. On the other hand, DCC shows very promising results with high applicability in large core count processors at minimum expense of cost and area.

IV. Conclusions

In this paper, we highlight different 'approaches' to achieve coherence in different types shared memory multiprocessor platforms. We discussed five solutions on different platform configuration implementing the two broad types of coherence protocol, namely, Snooping and Directory-based protocol.

The snooping methodology is faster with simpler implementation in CMP. It effective for low core count CMP and heterogeneous CMP platforms (such as SoC) and can further be implemented in Network processors, which can be indicated as future work in this implementation. However, it does not scale for large core count CMP and thus, directory-based protocols come into the picture.

The directory based protocol are very efficient in large core count processors but they are slow and their complex implementation make it difficult to implement. Optimisations of Dynamic Self-Invalidation, helps reduce overhead of messages in the CMP and hints to better performance. Better optimisation of having two such protocols, such as DirCC and EM, to have co-exist on a processor and transition for different processes. This helps the case for large core count CMP platforms and presents a promising solution for future multcore processors.

References

[1] Suh, T., Blough, D.M. and Lee, H.H. *Supporting cache coherence in heterogeneous multiprocessor systems.* In Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings IEEE. 2004, February, Vol. 2, pp. 1150-1155.

[2] Li, J.M., Liu, W.J. and Jiao, P. *A New Kind of Cache Coherence Protocol with SC-Cache for Multiprocessor.* In Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on IEEE, 2010, May, pp. 1-5.

[3] Dash, A. and Petrov, P. *Energy-efficient cache coherence for embedded multi-processor systems through application-driven snoop filtering.* In Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on IEEE, 2006, pp. 79-82.

[4] Lebeck, Alvin R., and David A. Wood. *Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors.* In ACM SIGARCH Computer Architecture News, ACM, 1995, vol. 23, no. 2, pp. 48-59.

[5] Khan, O., Lis, M., Sinangil, Y. and Devadas, S. *DCC: A dependable cache coherence multicore architecture.*, IEEE Computer Architecture Letters, 2011, 10(1), pp.12-15.