Variable Forwarding Cache Coherence for Chip Multiprocessors

Amit Roy[1], Supriya Vadlamani[1], T S B Sudarshan[2]
[1]Student, Computer Science & Information Systems Group,
[2]Assistant Professor, Computer Science & Information Systems Group,
Birla Institute of Technology and Sciences, Pilani
{amit_roy, supriya, tsbs}@bits-pilani.ac.in

Abstract

Caches in Chip Multiprocessors (CMPs) are organized as private L1 caches or large shared L2 cache or both. Most of the recent researchers have focused on architectural and circuit techniques to increase performance.

Variable Forwarding Cache Coherency combines the advantages of private caches and shared caches, i.e., low latency of L1 and miss rate of shared L2. This paper proposes a methodology to improve performance of the system by using variable forwarding cache coherence technique in CMPs.

1. Introduction

The ever-increasing levels of on-chip integration have enabled phenomenal improvements in computer system performance. The perpetual market pressure for improved performance is driving designers to build shared memory systems with large numbers of processors in them.

Chip-multiprocessors (CMPs) are gaining popularity both in small devices as well as the high end servers. Data sharing amongst the processors is an inherent characteristic, which demands data consistency. Typically, data consistency is maintained by cache coherency protocols.

Cache coherence implies maintaining data consistency amongst the caches of a multiprocessor system so that no data is lost or overwritten before it is transferred from a cache to the target memory. When multiple processors with separate caches share a common memory, it is necessary to keep the caches in a state of coherence by ensuring that any shared operand that is changed in any cache is changed throughout the entire system (one of the simplest of which is explained by the following diagram).

In a CMP, communication is typically carried out through cache coherence actions across the private caches. Cache coherence actions involve tag lookups and off-chip memory accesses which are power intensive processes.

The primary objective of the project is to improve the performance by incorporating adaptive caching into the existing cache structure for a multiprocessor system.

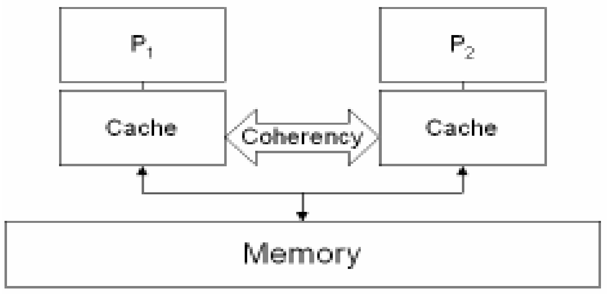

Figure 1

In the proposed implementation, based variable forwarding technique performance improvement is achieved by storing a Single Instance of a data and forwarding the data to the neighboring caches instead of replacing, thus reducing the off chip traffic as well as increasing the lifetime of data.

2. Related Work

Chip-multiprocessors (CMPs) have gained a lot of interest as a micro-architectural style for future microprocessors for high-performance systems as well as low-power systems. Much of the attention of recent research focuses on architectural and circuit techniques for increasing performance and reducing on-chip processor energy consumption via techniques such as coarse-grain coherence tracking [5, 6], token coherence [7], cache partitioning [3, 4], system bus optimizations [9], speculative-snoop power reduction technique [2], filtering remote snoop requests [10] and reducing switching activity [2]. Cache partitioning [4], (both static and dynamic) though being a good way of maintaining compositionality, consumes a huge amount of power. Off-chip accesses are one of the costliest operations in terms of power, especially because of the wastage of energy due to tag look ups on a global cache miss (due to cache miss or upgrade). [1] explains the concept of cooperative caching by dividing the data in a CMP into single instances and multiple instances. And subsequently applying either Token Coherence or 1-chance forwarding for updates on the data. Our strategy increases the longevity of single instance data by applying variable forwarding and adding a victim cache, thus reducing the number of off-chip accesses.

3. System Architecture

The basic multiprocessor architecture for implementation is described in Figure 2 each of which consists of: four nodes: processor, L1 cache, the interconnection network and additional hardware: Singleton Lookup Buffer (SLB) and Data Analysis Unit (DAU).

The SLB and DAU are as shown in Figure 3. The SLB keeps track of all the single instances of data used by its corresponding processor. The DAU comprises of a controller, a directory table and a memory element. The directory structure is intended to store details of all the data elements within the system. The memory element serves as a Victim Cache. A high-speed bus acts as the Interconnection Network.

4. Proposed Methodology

DAU consists of a set of counters for every unique data element corresponding to the processor accessing it and subsequently, based on the frequency of usage, either a singleton is replicated or vice versa. Singletons can either be evicted or forwarded from a cache.

The forwarding technique adopted is variable forwarding, where in, data is forwarded to the processor where it is predicted to have a high probability of being accessed in the near future.

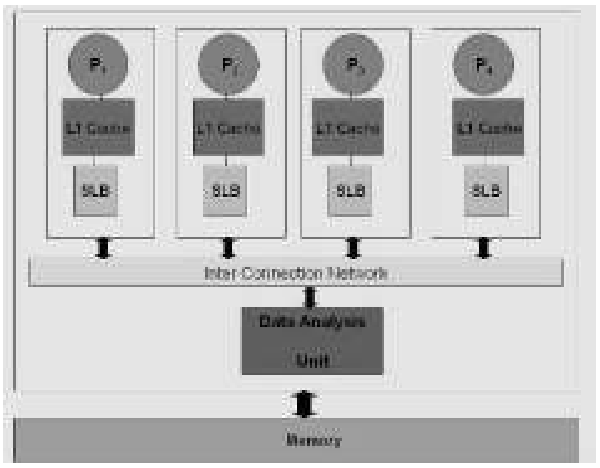

Figure 2

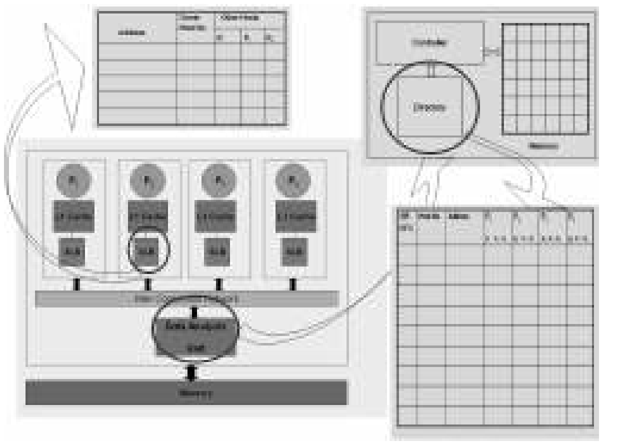

Figure 3

Predictions are made based on the access history of the data by the DAU. Data is forwarded as long as one of its counter values is greater than the threshold and there exists at least one replaceable data element; otherwise, it is evicted. Data is evicted when its counter value decrease below the threshold. On eviction, data is further stored in a victim cache. This increases the lifetime of a singleton in the system, thus minimizing the probability of having to make an off-chip memory access. Also, maintaining a singleton in the system eliminates the use of cache coherency protocol for these data elements

Replicates are overwritten with notifications sent to the hardware units. Coherency among replicates is maintained using a cache coherency protocol. The protocol intended to be used is directory protocol since the DAU can act as the directory.

5. Data Flow Design

Some of the important scenarios that define the system are described as below.

Initial State of the System: (cold start scenario) When an address is generated by one node, it will be a compulsory miss in the cache and hence will be fetched from memory. It will then be stored in the L1 cache as well as the SLB and DAU. The process can be described as follows

```
BEGIN
  Pi generates an address: $add_j$
  Do
    Cache Search
    If cache miss Do
      SLB Search
      if SLB miss Do
        DAU search
        if DAU miss Do
          Access Memory
          Get Data from Memory
          Make a new entry
          Increment Counter Value
```

```
            Save Data & address in Victim cache
            Send Data to Node
         end do (DAU miss)
       end Do (SLB Miss)
       Save Data (in cache)
       Make an entry into SLB
     end if (cache miss)
   end Do
END
```

Data Look Up in the System: Depending on the number of instances of a particular data present in different caches, the data is classified as a singlet or replicate and accordingly different policies are adopted. If it is singlet, then the processor requesting data receives a message from the DAU that contains the host id which is storing the data at that time. The processor then probes and retrieves the required data. If the data is a replicate, since multiple copies already exist, the processor receives only the data, which it stores in its L1.

Prerequisite:

The data whose address (add_j) is generated by a node (say Pj) is present in the system.

```
BEGIN
   Pj generates an address: $add_j$
   if cache miss Do
      SLB search
      if SLB miss Do
         DAU search
         if  DAU hit Do
           if(number  Instances(Data(addj))=Multiple )Do
             make an entry into the Pj cell of the data
              initialize the counter for Pj for the data
              return{message, data}
            end if (Multiple Instances)
            else Do
               make an entry into the Pj cell of the data
               initialize the counter for Pj for the data
                return{message, hostAddress }
            end else (Multiple Instances)
         end if (DAU hit)
       end if (SLB miss)
     end if (cache miss)
END
Pj after receiving message
BEGIN
   Check message
   if (message == multiple instances of data) Do
      save data in cache
   end if
   else Do
      save pointer in SLB
     access data  from the node corresponding to the
      pointer
   end else
END
```

Data Transfer in the System: Transfer of data from one node to another node occurs when a single instance in the system is being accessed very frequently by only one processor which isn't the host processor. In such a case, the data instance will be moved from the cache of the host processor to that of the destination processor (i.e the one that currently accesses the data frequently).

In the following pseudo code, P_{host} represents the processor that stores the data instance, P_{ref} represents the processor that is pointing to the data instance and P_{dest} represents the processor to which the data instance has to be moved to. P_{ref1} indicates the processor which is the potential destination.

Prerequisite:

Processors P_j, P_k and P_l (P_{ref}) are pointing to P_i, which holds the data. The counter parameters of P_i < threshold and one of $\{P_j,P_k,P_l\}$ > threshold.

Node: DAU

```
BEGIN
 if(($P_{ref1}$.counter>threshold)and($P_{host}$.counter<threshold))
        $P_{dest} = P_{ref1}$
        Send the destination processor's id and a pointer
        to the data to $P_{host}$
  end if
END
```

Node: P_{host}

After receiving the destination processor's id and the data address from the DAU

```
BEGIN
     Update Entry in the SLB
      Send the update to all the processors accessing the
      data
      Send the data to the destination processor's cache
END
```

Node:P_{ref}

After receiving the data address and the destination processor's id from P_{host}

```
BEGIN
     Store the data in the cache
     Update Entry in the SLB
END
```

Node: P_{dest}

After receiving the data address and the data from the P_{host}

```
BEGIN
     Update Entry in its SLB
END
```

Data Replication in the System: The DAU, on finding that a single instance is being accessed often by the host and at least one of the reference processors send a message to the host for replication. The host in turn sends a message to all the other nodes currently accessing the data, and subsequently multicasts the data.

Prerequisite:
If P_j, P_k or P_{ref} pointing to P_i and the counter parameter of either P_j or P_k crosses the threshold value along with P_i's counter value being higher than threshold.

```
Node $P_i$
 BEGIN
   Send the data pointer and data to all the processors
    accessing the single instance.
    Delete Entry in the SLB
 END
Node $P_{others}$
  After receiving the data pointer and the data
   BEGIN
         Delete Entry in the SLB
         Store Data in the Cache
   END
```

Removal & Eviction of Data in the System: The data present in one location in the L1 cache of a node can either be removed permanently from the system or it can be evicted and stored in another cache. If a data has multiple instances in the system, and it is being replaced by a singlet, then the data is deleted from the system. On the other hand, if the data being replaced is a single instance then instead of being overwritten the data is evicted and is stored in a Victim cache inside the DAU.
Prerequisite: data is evicted in Pj.

```
BEGIN
    $P_{host}$.Send( message,address) to DAU
    if (numberInstances(Data(addj)) in DAU = 1)Do
       if (number of $P_{ref}$ = 0)  Do
         evict the data from $P_{host}$
         write the data into the Victim Cache
       else
          find the next node out of $P_{ref}$ with the highest
          counter value
          copy the data to that node
          multicast message to all the $P_{ref}$ & the old $P_{host}$
          for updating their respective SLB entries
       end Do($P_{ref}$=0)
    else
      if(only 1 counter > threshold parameter) Do
          singlify the data (store the data only in this node
          and multicast messages to all the nodes that
          contains the data to delete  and update their
          respective SLB entries)
       end if
       else
         if(none of the counter > threshold parm) Do
           find the node with the highest counter value.
           singlify the data
         end if
     else if (numberInstances(Data(addj)) in DAU>1)Do
       Delete the Data
      end if
 END
```

6. Implementation

The implementation is being done in Verilog using ModelSim Simulator. The inputs for comparison used are address traces of Simple Scalar for different SPEC95 Benchmark applications. The following graph in Figure 4 & 5 shows the improvement in the on-chip hit rate.

The first mix consists of the applications Compress95, Hydro2D, Ijpeg, TomcatV and the second consist of Apsi, Perl, Turb3D and Wave5. Turbo3D and Wave5 has shown an improvement of 7 and 5 percent respectively, while the improvement for the average case is 0.4. This is due to overlapping of the address space of the two programs which reaps the maximum benefits of the proposed architecture.

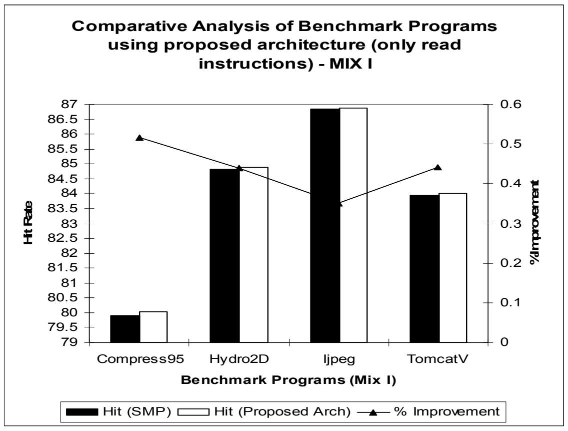

Figure 4

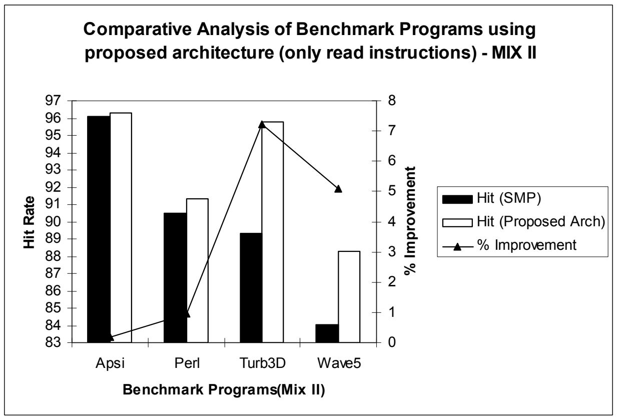

Figure 5

7. Conclusion

With the increasing need for efficiency, variable data forwarding caching stands to be a crucial design consideration. The proposed design emphasizes the maintenance of unique data elements to the maximum extent possible. Also, the lifetime of data elements are extended based on their access patterns. These techniques can, enhance performance and power efficiency if the Data Analysis Unit and Coherence Protocols maintain synergy.

Future Work includes adopting this technique for achieving power efficiency.

8. References

[1] Jichuan Chang and Gurinder S. Sohi, "Cooperative Cache for Chip Multiprocessors," Proc. of the 33rd Annual International Symposium on Computer Architecture (ISCA), 2006, pp 264-276.

[2] C. Saldanha and M. Lipasti "Power Efficient Cache Coherence," High Performance Memory Systems, Springer-Verlag, New York, 2003, pp 63-78.

[3] A.M. Molnos, M.J.M. Heijligers, S.D. Cotofana, J.T.J. van Eijndhoven, "Compositional, efficient caches for a chip multi-processor," Proc. of the conference on Design, Automation and Test in Europe, 2006, pp 345-350.

[4] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," The Journal of Supercomputing, vol. 28, issue 1, 2004, pp 7-26.

[5] Moshovos, A., RegionScout, "Exploiting Coarse Grain Sharing in Snoop-Based Coherence," Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA), 2005, pp 234-245.

[6] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking," ACM SIGARCH Computer Architecture news, vol 33, issue 2, 2005, pp 246-257.

[7] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M.K.Martin, David A. Wood, "Improving Multiple-CMP Systems Using Token Coherence," Proc. of the 11th International Symposium on High-Performance Computer Architecture, 2005, pp 328-339.

[8] Mirko Loghi, Massimo Poncino and Luca Benini, "Cache Coherence Tradeoffs in Shared-Memory MPSoCs," ACM Transactions on Embedded Computing Systems, Vol. 5, No. 2, 2006, pp 383-407.

[9] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, John B. Carter, "Interconnect-Aware Coherence Protocols for Chip Multiprocessors," ACM SIGARCH Computer Architecture News, Vol. 34, Issue 2, 2006, pp 339-351.

[10] M. Ekman, F. Dahlgren, and P. Stenström, "Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors," Workshop on Duplicating, Deconstructing, and Debunking, 2002.