Project Part 3 – A Case for Cache-Core Decoupling in CMP

Rangeen Basu Roy Chowdhury

Introduction

Multi-threaded applications running on a chip multi-processor (CMP) exhibit different types of communication patterns. In some applications, each core most frequently communicates with its neighboring cores [2] leading to a lot of coherence traffic being exchanged between these two entities. Examples of such applications are ocean_cp, lu_cb, blackscholes and barnes from the Parsec and Splash2x benchmark suites. With stock MESI directory based protocol, the directory acts as one level of indirection between the two neighboring cores. Eliminating this extra indirection might reduce the runtime for these applications. Providing direct connectivity between neighboring L1 caches is a way to eliminate this indirection.

In a chip multiprocessor, cores can be designed and floorplanned in a way to allow fast connectivity between L1 caches of neighboring cores, essentially doubling the cache capacity while using only a modest amount of extra logic. The caches still remain relatively simple and only a few additional transient states are added to the coherence protocol. The michroarchitecture is similar to Cache-Core Decoupling (CCD) proposed by Rotenberg et. al. [1]. While the original CCD mechanism tries avoid compulsory cache misses when a process migrates to a neighboring core and primarily focusses on multiple single threaded workloads, the approach can be generalized to increase cache capacity and reduce miss latency. The original CCD also does not deal with coherence issues. Although CCD can be implemented in a 2D layout, 3D die stacking can result in a much more efficient floorplan [1] and lower latencies for cross-core cache accesses.

The goal of this project is to design the cache-core decoupling mechanism in the context of a CMP that deals with multi-threaded applications and handles coherence issues correctly. If implemented correctly, it is also possible to accelerate single threaded programs by effectively increasing the L1 cache capacity of a core. If a core's neighbor is turned off or is idle, the neighbor's cache can be used as an extension to the core's own cache.

Design

The floorplan of the two neighboring cores are done in such a way so that the L1 data caches of the two cores are almost adjacent to each other. These two caches can then be connected using low latency wires, through appropriate buffers if required. The pair of cores can then be laid out in a mesh as a single entity thus creating a regular floorpaln for the entire CMP. NICs can still be placed in the cores in such a way that the on chip network maintains its regularity. Figure 1 gives a rough floorplan of the system. Using 3D stacked dies makes it even easier to connect cores without the need for two different core floorplans. The cores that are

vertically aligned, can be connected using a large number of face-to-face wires. Rotenberg et. al. present and analyze such layouts for 2D and 3D dies in [1].

Each core in the pair can access the other core's caches on a miss in its own cache. The core that initiates the request is called the primary core and its cache is called the primary cache. The neighboring core in the pair is called the secondary core and its cache is called the secondary cache. One can add an additional port in the secondary cache to process CCD requests or the CCD requests and the primary cache's own requests can be multiplexed onto a single cache port. In this project, a single port was used to service both CCD and primary requests. The CCD requests were given higher priority in this project. The design was implemented using the GEM5 full system simulator and evaluated using various parallel benchmarks.

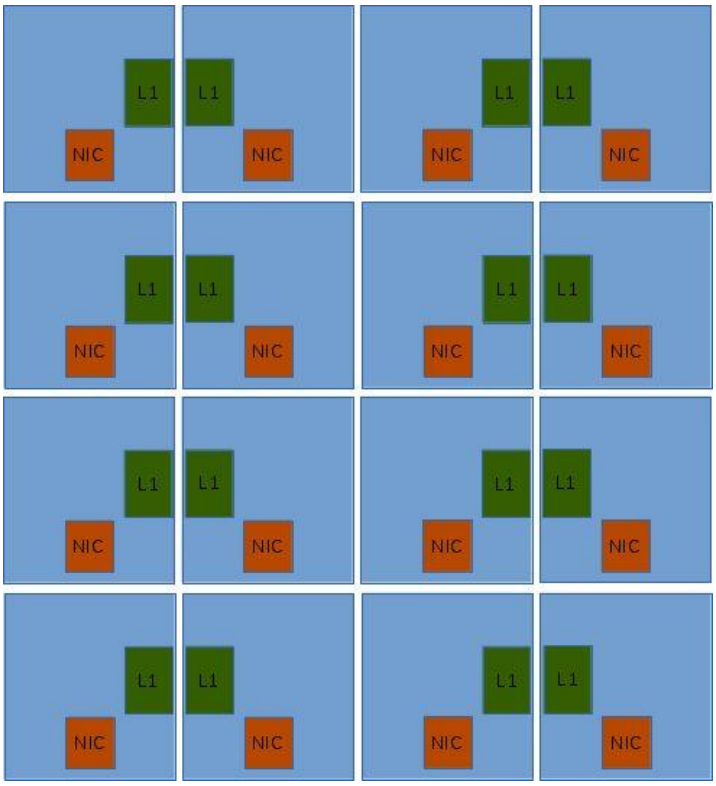

Figure 1: Rough floorplan of the CCD enabled CMP

GEM5 Implementation

The CCD access paths were implemented using two new Message Buffers, one each for CCD Request and Response. The SLICC compiler was modified slightly to set correct attributes for these message buffers. The two buffers are CCDRequestFromNeighbor and CCDResponseFromNeighbor. Each cache also contains a pointer to its neighbor's CCD buffers

for queuing requests and responses to these queues. The message buffers are connected properly using the protocol python script. The primary cache controller initiates a CCD request, which is processed by the secondary cache controller and a response is sent. The primary cache controller then takes appropriate action once the response is received. Each message buffer has a minimum 1 cycle latency to enqueue and each cache controller has a single cycle response time resulting in a round trip latency of 4 cycles. This round trip latency can be reduced with better microarchitectural design by using fall through queues instead. Such queue must be implemented as Ruby does not provide one out of the box and can be undertaken as future work.

Protocol Modifications

Only one transient state was added in the L1 cache along with a few new message types and events. These are explained in detail in the following tables. The primary cache controller first checks for a cache hit in the primary cache and if the block is not present in the primary cache, a CCD access to the secondary cache is initiated. A cache block is allocated to hold the transient state NP_CCD while waiting for the CCD access to complete. If the CCD access to the secondary cache completes successfully, the data is sent to the CPU sequencer and the cache block is de-allocated. On the other hand, if the CCD access is unsuccessful, a GETS request is issued to the memory hierarchy to bring the block into the cache.

Name	Comment
CCD_LD_M **CCD_LD_E** **CCD_LD_S**	These requests are sent to the neighboring core when trying to access the neighbor's cache. Since we already look up the state of the cache block in the neighbor's cache (S, E or M), we send the appropriate request. When the request is processed at the neighboring cache, if the state is still the same, an Ack is sent back. Otherwise a Nack is sent.
CCD_ACK **CCD_NACK**	These are responses sent from the neighboring cache to the requesting cache to indicate a success or failure to read the correct data correctly from the cache.

Table 1: New Message Types

State	Description
NP_CCD	This state is reached when a CCD Load request has been made to the neighboring core and the state machine is waiting for the Ack/Nack to come back before taking suitable action.

Table 2: New Transient state in L1 Cache Controller

Event	Comment
CCD_Access	On a load from the sequencer, the L1 cache controller looks up the tag state of the cache line in both caches. In parallel to tag lookup, the cache controller checks the tag array of the primary cache and if it misses in the primary cache and the neighbor possibly has the line the appropriate state, a CCD_Access event is generated. As a response a CC_LD request is queued up in to the CCD queue. (**P.S:** The protocol can be modified such that a lookup in the neighbor's tag array is not necessary. A CCD access can be issued in parallel to a GETS request and additional transient states can be used to complete the requests correctly.)
CCD_Load_S **CCD_Load_E** **CCD_Load_M**	These events are a result of the secondary cache receiving a CCD_LD request from its neighbor. Appropriate transitions happen and either an Ack or a Nack is sent to the requesting cache.
CCD_Ack **CCD_Nack**	These events are a result of response from the secondary cache.

Table 3: New Events in L1 Cache Controller

L1 State	Event	Action	Next L1 State
NP/I	CCD_Access	● Allocate cache block triggering a replacement if necessary ● Send CCD_LD_* request to secondary cache and wait for Ack/Nack	NP_CCD
NP_CCD	CCD_Ack	● Send data to Sequencer (CPU) and deallocate cache block	I
NP_CCD	CCD_Nack	● Send GETS to fetch line from next level	IS
S	CCD_Load_S	● Send CCD ack along with data block to requesting neighbor	S
E	CCD_Load_E	● Send CCD_ACK along with data block to requesting neighbor	E
M	CCD_Load_M	● Send CCD_ACK along with data block to requesting neighbor	M
Any Other State	CCD_Load_*	● Send CCD_NACK. If this situation happens, it means the state of the cache block has changed during the course of CCD access.	Hold State

Table 4: New transitions in L1 Controller

Evaluation

Four benchmarks shown in table 5 were chosen based on their observed communication pattern as presented in [2]. Barnes, ocean_cp and lu_cb were evaluated with 16 processors

whereas blackscholes was evaluated with 8 processors as it didn't simulate correctly with 16 processors.

Benchmark	Suite
Barnes	Splash2x
Ocean_cp	Splash2x
Lu_cb	Splash2x
Blackscholes	Parsec

Table 5: Benchmarks Used

As shown in Table 6, three out of the four benchmarks got speedups. Lu had the maximum speedup of 20% while Barnes had a speedup of only 1%. Looking at the L2 access statistics in Figure 2, CCD reduced the number of L2 requests and thus reduced the miss latency by a decent amount. Most completed CCD accesses have an issue to completion latency of 4 - 5 cycles. Table 7 shows the number of various CCD related events for the different benchmarks. Both clean and producer-consumer CCD accesses were made by the benchmarks.

simulation cycles (ruby cycles)	Base	CCD	Speedup
Barnes	621,246,302	613,380,752	1.01
Blackscholes	168,561,646	181,759,676	0.93
Ocean	601,240,835	540,691,344	1.11
Lu	312,552,730	263,690,909	1.19

Table 6: Simulation Cycles and Speedup

Although the number of L2 accesses in case of Blackscholes were lower, it suffered from larger number of L1 misses due to non-allocation of cache blocks in the L1 on a successful CCD access. On a hit in the secondary cache, the line is not brought into the primary cache from the memory hierarchy or from the secondary cache. This is clearly a design flaw and can be easily solved by modifying the protocol to bring in a cache block that hit in the secondary cache in parallel to the CCD access. This will lead to single cycle cache hits for future accesses as opposed to 4 cycle CCD access.

CCD Events	Access	Ack	Nack	Load_S	Load_E	Load_M
Barnes	58,590,940	58,590,907	33	57,819,753	144,347	626,807
Blackscholes	18,505,067	18,505,067	0	18,487,699	15,690	1,678
Ocean	9,044,181	9,044,164	17	8,806,481	100,086	137,597

Table 7: CCD Related Events. Most of the sharing is clean ('S' and 'E' states) although decent amount of producer-consumer sharing ('M' state) is also observed. Very few of the CCD requests were unsuccessful.

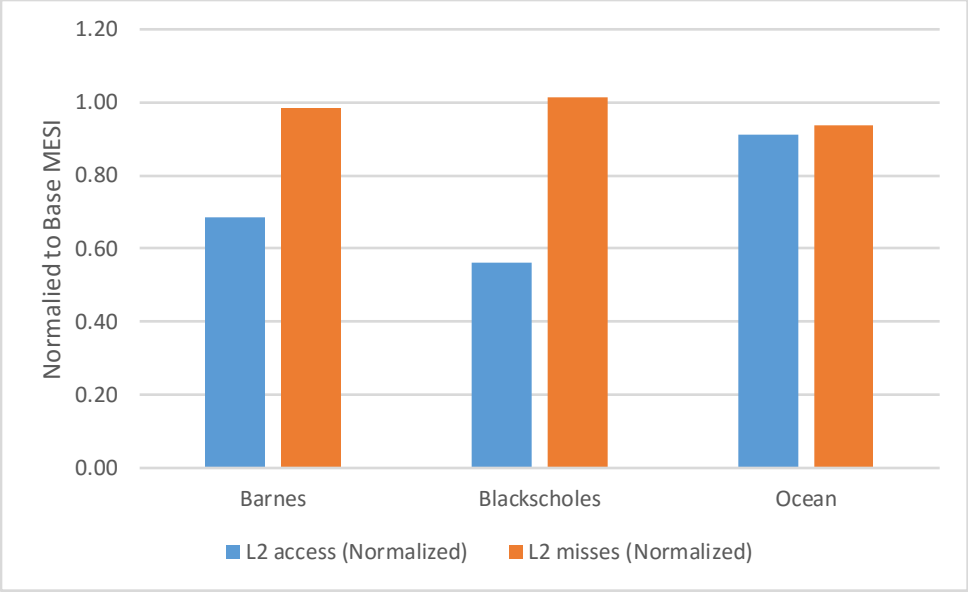

Figure 2: L2 accesses and misses normalized to the Base design. As expected, L2 accesses have been significantly reduced.

In conclusion, the CCD technique to reduce miss latency has good potential and with careful microarchitecture and protocol design, a significant benefit can be achieved. With 3D stacking, the latency of the CCD access can be reduced further leading to a even greater speedups.

References

[1] Rotenberg, E.; Dwiel, B.H.; Forbes, E.; Zhenqian Zhang; Widialaksono, R.; Basu Roy Chowdhury, R.; Tshibangu, N.; Lipa, S.; Davis, W.R.; Franzon, P.D., "Rationale for a 3D heterogeneous multi-core processor," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on* , vol., no., pp.154-168, 6-9 Oct. 2013

[2] Barrow-Williams, N.; Fensch, C.; Moore, S., "A communication characterisation of Splash-2 and Parsec," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, vol., no., pp.86-97, 4-6 Oct. 2009
doi: 10.1109/IISWC.2009.5306792