Cache Hierarchy for 100 On-Chip Cores

Mohamed Zahran
Department of Electrical Engineering
City University of New York
(mzahran@ccny.cuny.edu)

Abstract

The increase in the number of on-chip cores, as well as the sophistication of each core, place significant demands on memory systems. We need a high bandwidth, low-latency, high-capacity, and scalable memory systems. The current memory system hierarchy, does not scale beyond few tens of cores.

The of this abstract is to shade some light on an alternative to develop a new memory hierarchy that can scale to serve hundreds of on-chip cores, have tolerable access time, overcome capacity versus accuracy dilemma, and manage the off-chip bandwidth.

Keywords: CMP, cache hierarchy, scalability

1 Introduction

Compared to uniprocessor systems, CMPs are putting much more pressure on the memory. First, each core can have different working set, leading to more conflicts and interference if sharing caches with other cores. Second, in order to sustain high throughput, each core is expecting the data/instruction to arrive in a timely fashion, so high memory access time is not acceptable. Third, in most real-life applications, the working set of each application is big, to the extent that the sum of the working sets of applications executed simultaneously on-chip will be always much bigger than the available caches, leading to higher miss rate. Increasing cache size results in increasing access time. Finally, because the number of pins of a chip does not scale with the same rate as the number of transistors on chip (i.e. the number of cores) [1], we will face bandwidth problem for the off-chip access. A solution is needed to deal with the above problems.

2 Traditional Solutions Will Not Work

Traditional solutions, such as increasing cache sizes or using longer hierarchy will not scale beyond few on-chip cores. First, the *horizontal* growth, that is, the increase in number of inputs to the cache, is not scalable. To increase the number of inputs to the cache, we can either increase the number of ports per bank, and/or increase the number of banks. The increase in number of ports per bank will lead to a tremendous increase in area, access time and power consumption. For example, increasing the number of ports of a bank from two to four, doubles the power dissipation, increases the access time by around 40%, and the power consumption by 120%, for a 2MB cache, with 70nm technology. And since we are talking about hundred cores per chip, then increasing the number of ports is not the answer. On the other hand, increasing the number of banks results in decrease in access time, but as we keep increasing banks, the decrease in access time start tapering off. Increasing the number of banks, causes the power consumption to steadily increase by around 50%, while the area increase is much smaller.

Second, the *vertical* growth, that is, increasing the length of the cache hierarchy by adding more levels, is not scalable too. As we keep increasing on-chip cores, these cores must share an on-chip cache at some level in the hierarchy, in order to keep coherence protocol on-chip, and avoid the off-chip access. However, all the cores cannot share a single cache, due to the horizontal unscalability. In order to overcome this, we will allow every few cores to share a cache, then every few caches to share another higher-level, and so on. Although this scheme looks feasible, it suffers of several drawbacks. First of all, if several cores will be sharing a cache, then which cores to choose? Especially that we do not know yet the different workloads that will be running on those cores. If each of the cores sharing a cache is running a memory-intensive program, in a multiprogrammed environment, then there will be a severe contention around the shared cache ports. Moreover, if a multithreaded workload is assigned to cores sharing a cache very high in the hierarchy, for example at level 5 or so, then the application of the coherence protocol will be very slow, negatively affecting the performance. Finally, if the hierarchy becomes very long, then accessing the higher level will be as slow as accessing the off-chip memory, making the higher level of that long cache hierarchy useless. Therefore, the vertical increase in cache is not a good solution for hundred on-chip cores.

We propose a new scheme: distributed shared cache.

3 Distributed Shared Cache

We propose to use a group of caches not assigned to any specific core. The whole group of caches, distributed shared cache (DSC), appears as a big scalable cache.

3.1 Main Idea

DSC is not used as L1 cache, but each core has its own private hierarchy, and then they all share DSC, as the highest level in memory hierarchy. The main purpose behind this arrangement is for the lower level cache hierarchy, that is, L1 and possibly L2, will act as bandwidth filter, providing high bandwidth for their own cores, while shielding higher levels from it.

DSC will act as the common shared medium between all cores. As the number of on-chip cores increases, the number of caches in DSC also increases.

The interconnections between caches forming DSC can be easily impemented, and is scalable. This is due to two main advances in technology. The first is the usage of network on-chip (NoC). NoC is a scalable approach for on-chip interconnects, which comes very handy when the number of cores increases. Moreover, the delay taken in routing is not affecting the whole system performance, because DSC is in the highest level of the hierarchy.

The other advance in technology that makes NoC even more efficient is the 3D Chip Multiprocessors. As the number of caches forming DSC increases, the two-dimensional plane designs will not be feasible, due to the large chip area occupied by the caches. Even if we can afford large 2D area, the distance between the farthest caches will require large number of cycles. Therefore, 3D is an attractive solution for the interconnect, for many reasons. First, it is more scalable. Second, the distance between vertical layers is very small, making the routing latency between two vertical caches communicating together negligible. Third, 3D chip design reduces the average number of hops between communicating caches, and hence, reduces the total access time of DSC. Finally, with 3D chip design, there is more freedom in placing cores, therefore, cores, and their private cache hierarchy, can be placed near caches forming DSC.

3.2 Advantages of DSC

DSC has many advantages. First it is scalable. The cumulative number of ports in DSC is larger than any traditional cache. For instance, if we have DSC with only nine caches, with two ports each, then the total number of ports is 18. We cannot build a traditional cache with 18 ports. The large cumulative number of ports implies high bandwidth. This is because the access time of DSC is equal to the access time of any single cache plus the routing time. Since each individual cache is relatively small with small number of ports, then the access time is small. When caches are assigned to a core, we try to assign the nearest cache to that core, whenever possible, making the routing time also small, therefore having higher bandwidth. Fourth, unused caches can be turned off to save power. Finally, it is to be noted that all other schemes applied to enhance the access time, and accuracy of individual caches, such as NUCA [3], are totally orthogonal to DSC, and can be applied to each cache forming DSC.

3.3 Implementation Issues

Cache Assignment: Since threads are assigned to cores in almost a random fashion, cache assignment needs to be done to satisfy the following criteria. First caches must be close to the requesting core., Second,cores running threads of the same application must be assigned nearby caches, to form a *region of caches*, that can communicate easily. Since threads of the same application are assigned to cores, and retired, at run time, the problem of assigning caches to cores is similar to the problem of memory management and dynamic allocation.

Concept of Regions: There are two types of applications that can run on CMP. The first type consists of multithreaded applications. In that type, an application consists if several threads, executed in parallel, and can be assigned to cores and retired at different times. Caches assigned to cores running threads from the same application will be assigned in a way to be close to the cores, and close to themselves, whenever possible, to facilitate communication. Those assigned caches are called regions. If it is not possible to assign a cache to be close to the core and at the same time close to the other caches in the region, then preference will be given to a cache close to the core. In this case, the region will formed of caches scattered in different places. On the other hand, if we cannot find a cache close to the requesting core, we all assign the closest possible, and when another closer cache becomes available, because its core becomes idle, then the date *migrates* to the closer cache. Although the migration process may be expensive, it does not happen frequently. If the number of caches forming DSC is less than the number of cores, then we may run out of caches. In which case, we will allow regions to *overlap*. That is, a cache may have two different applications running on it at the same time. If this happens, then there must be a way to avoid conflicts. We can adapt techniques such as adaptive cache topology [5], application-adaptive cache [4], in an *on-demand* way. That is, these techniques will be enabled only if threads from different applications are assigned to the same cache. A lot of interesting ideas in cooperative caching are presented in [2]. However, they are assuming caches statically assigned to cores and accessed only by that core.

Coherence in a Region: Since DSC is used as a shared medium, then cache coherence must be taken care of. However, unlike traditional shared caches, the coherence here is simple, because it will happen at the *region level* only. So we plan to adapt the traditional directory-based coherence protocol, to have a *distributed* directory-based coherence protocol, that ensures coherence on a per-region basis. Therefore, there must be a directory for each region. The number of regions is not known before hand. We cannot have a directory for each cache, and at the same time we cannot have a centralized directory, as it will be a bottleneck. The solution to this is to divide all the DSC caches into *fixed static regions*. Contrary to the dynamic regions, which are formed at run time, and are of different sizes (i.e. different number of caches), depending on the number cores, the fixed regions are static geographical regions, of the same size. We will divide all the DSC caches into groups of the same size. Each group consists of neighboring caches. These groups are the static region. Each static region will have its own directory. When a dynamic region is formed, it can span one or more static regions, in this case, it will use the tables of all the static regions that it touches. This will ensure the correctness of the coherence protocol. A dynamic region may expand or shrink at runtime. Therefore, the directory protocol may be using different number of tables at different times.

4 Conclusion

In this abstract we briefly discussed a new proposed scheme to design cache system for 100 on-chip cores. We presented the main design, as well as several implentation issues.

References

[1] Semiconductor Industry Association. International technology roadmap for semiconductors, 2003.

[2] J. Chang and G. Sohi. Cooperative caching for chip multiprocessors. In *Proc. 20th Int'l Symposium on Computer Architecture (ISCA)*, June 2006.

[3] C. Kim, D. Burger, and W. Keckler. An adaptive, non-uniform cache structure for wire-dominated on-chip caches. In *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 2002.

[4] J-H Lee and S-D Kim. Application-adaptive intelligent cache memory system. *ACM Transactions on Embedded Computing Systems*, 1(1), 2002.

[5] J-K Peir, Y. Lee, and W Hsu. Capturing dynamic memory reference behavior with adaptive cache toplogy. In *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1998.