2011 International Conference on Parallel Architectures and Compilation Techniques

Beforehand Migration on D-NUCA Caches

Javier Lira[1], Timothy M. Jones[2], Carlos Molina[3] and Antonio González[1,4]
[1] *Dept. of Computer Architecture, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain*
[2] *Computer Laboratory, University of Cambridge, CB3-0FD Cambridge, UK*
[3] *Dept. of Computer Engineering, Universitat Rovira i Virgili, 43007 Tarragona, Spain*
[4] *Intel Barcelona Research Center, Intel Labs - UPC, 08034 Barcelona, Spain*
javier.lira@ac.upc.edu, timothy.jones@cl.cam.ac.uk, carlos.molina@urv.net and antonio.gonzalez@intel.com

***Abstract*—Determining the best placement for data in the NUCA cache at any particular moment during program execution is crucial for exploiting the benefits that this architecture provides. Dynamic NUCA (D-NUCA) allows data to be mapped to multiple banks within the NUCA cache, and then uses data migration to adapt data placement to the program's behavior. Although the standard migration scheme is effective in moving data to its optimal position within the cache, half the hits still occur within non-optimal banks. This proposal reduces this number by migrating data beforehand.**

I. Introduction

An important issue to be addressed for CMP NUCAs is to determine which is the most suitable location for data blocks in the NUCA space. This is not an easy problem to solve as different data blocks can experience entirely different degrees of sharing (i.e., the number of processors that share them) and exhibit a wide range of access patterns. Moreover, the access pattern of a given data block can also change during the course of program execution. Therefore a static placement strategy which fixes the location of each block for the entire application will perform sub-optimally. To address this issue, D-NUCA [1], [2] allows data to be mapped to multiple banks within the NUCA cache, and then uses *data migration* to adapt its placement to the program behavior as it executes.

Existing data migration policies are effective in concentrating the most frequently accessed data in the NUCA banks with the smallest access latency. However, a significant percentage of hits in the NUCA cache are still resolved in slower banks. This proposal complements the migration scheme with a simple prefetching technique in order to recognize access patterns to data blocks and anticipate data migration. We show that using data prefetching to anticipate data migrations in the NUCA cache can reduce the access latency by 15% on average and achieve performance improvements of up to 17%.

II. The Migration Prefetcher

Figure 1 shows the NUCA cache including the prefetcher components. There are eight prefetchers (one per core) that are located in the cache controller, which is the entrance point to the NUCA from the L1 cache. A prefetcher consists of the *prefetcher slot (PS)*, which stores the prefetched data, a structure to manage the *outstanding prefetching request* and

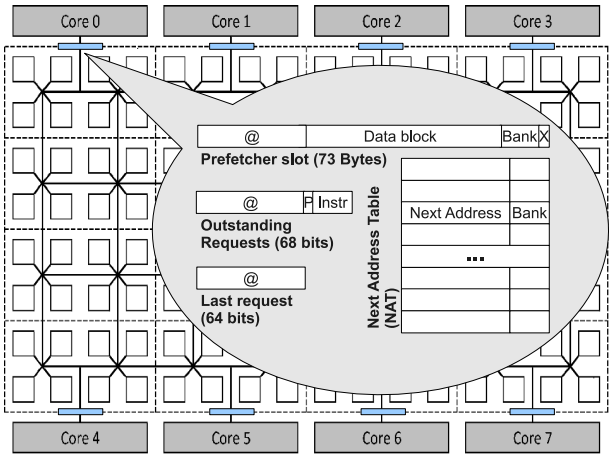

Fig. 1: Baseline architecture with the Migration Prefetcher.

the *last request* sent from the L1, and the *Next Address Table (NAT)*, which keeps track of the data access patterns. For each address requested, the NAT stores the next address to be accessed.

The prefetcher manages memory requests to the NUCA cache and keeps track of data access patterns. When a known pattern starts, the prefetcher predicts the next address and then prefetches it into the PS, and therefore closer to the requesting core. If the prediction was correct then the latency of the second access would be reduced since it would hit in this structure instead of a far-away bank.

When a request from the L1 cache arrives to the prefetcher, it can be resolved in three ways: 1) If it hits in the PS, the prefetcher sends the requested data to the L1 cache. This is the optimal situation because the memory request was serviced in the minimum amount of time due to the previous prefetch. 2) If the PS does not have the requested data, it still can hit on an outstanding prefetch request. Although the memory request will take longer than in the optimal case, there can be a reduction in the access latency in this situation. 3) If the request misses in both structures then it will be forwarded to the NUCA cache, just as if the prefetcher was not there.

At the same time as checking the PS, the NAT is updated by storing the current address in the entry for the last request, then updating the last request field with the current address.

1089-795X/11 $26.00 © 2011 IEEE
DOI 10.1109/PACT.2011.38

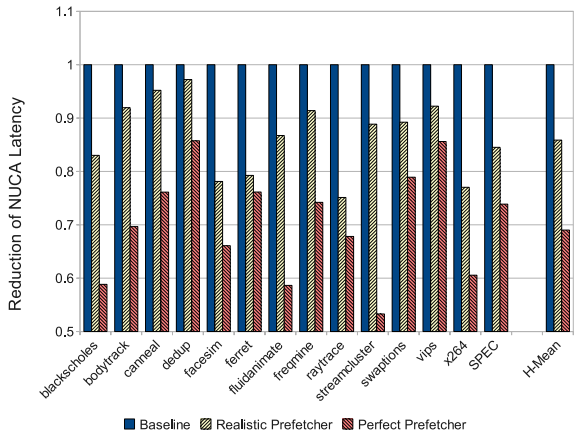

Fig. 2: Reduction of NUCA access latency.

If a pattern starting with the address of the current memory request exists in the NAT then the prefetcher submits a prefetch request and updates the outstanding request field.

The prefetcher also includes one extra bit per NAT line to represent the confidence counter and a comparator to determine whether the NAT line should be updated.

A. Lookup in NUCA

In addition to predicting the next address, we also allow the prefetcher to predict the position in the NUCA cache that this data will be. We experimentally observe that related data move together among bankclusters, therefore a significant percentage of consecutive memory requests are resolved by banks that belong to the same bankcluster. Based on this observation, for each line in the NAT we also keep the number of the bankcluster of the last responder that held the data belonging to the corresponding address. Therefore the prefetch request is sent only to one bank instead of to all candidate banks where the requested data could be mapped. In order to enhance the accuracy of the access scheme without significantly increasing the network-on-chip traffic we consider also sending the prefetch request to the local bank when it is not the last responder. We call this the *last responder and local* scheme.

B. Tuning the prefetcher

The prefetcher introduces several challenges that must be met in order to provide a realistic implementation, such as the size of the NAT or the access scheme used when looking for data. We have analysed these challenges and propose a prefetcher with a *one-bit confidence system*, an *NAT table size of 29 KBytes* (12 addressable bits), and the *last responder and local* as a search scheme. The total hardware overhead of the actual prefetcher is 264 KBytes (33 KBytes per core), which represents less than 4% extra hardware compared to the baseline architecture.

III. Performance Analysis

On average, performance benefits of the NUCA cache are increased by 4% when using the realistic implementation of the migration prefetcher, while using the perfect prefetcher (that always knows the location of the data and has an unlimited NAT) it obtains a 7% performance improvement. In general, the prefetcher is often able to find data access patterns in the simulated applications and exploit them for performance gains. In the parallel applications particularly, the realistic migration prefetcher achieves performance improvements of over 5% in four benchmarks (*facesim*, *streamcluster*, *x264*, and 17% in *raytrace*). On the other hand, when considering the multi-programmed environment, both versions of the migration prefetcher, perfect and realistic, outperform the same without using a migration prefetcher by 5%.

The key reason for the performance improvements is a reduction in the access latency to the NUCA cache on a hit in the prefetcher. Figure 2 shows that the perfect prefetcher reduces the NUCA access latency by 30%, on average, while the reduction with the realistic implemetation is 15%. These results show the ability of the prefetcher to recognize data access patterns and anticipate data migrations in order to increase the number of memory requests satisfied with the optimal latency. Unfortunately, the performance benefits of using this mechanism are restricted by the overall hit rate in the NUCA cache, which is small in some PARSEC applications. For example, the access latency for *fluidanimate* is reduced by over 40% by using the perfect prefetcher. However, this only translates into a 2% performance increase for this application due to its low hit rate of 3%.

IV. Energy Consumption Analysis

On average, neither of the two versions of the migration prefetcher increase the on-chip network contention. Although we would expect the prefetcher to significantly increase the on-chip network traffic due to the large numbers of prefetch requests that this mechanism sends, in actual fact every hit in the prefetcher saves up to 16 memory requests to the NUCA banks. To take advantage of this and therefore avoid increasing the on-chip network traffic, the migration prefetcher must be effective.

Considering the energy consumption, the migration prefetcher consumes about the same amount per memory access as the baseline configuration. Static energy is the major contributor to the overall energy consumed by caches, and this case is not an exception. In general we find that although the migration prefetcher introduces 264 KBytes to implement the required structures, it achieves a significant enough reduction in dynamic energy consumption in the on-chip network to counterbalance these overheads. The migration prefetcher achieves the highest reduction in energy consumption per memory access in applications where it also obtains the highest performance benefits, like *facesim*, *raytrace* or *x264*.

References

[1] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Procs. of the 37th International Symposium on Microarchitecture*, 2004.

[2] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Procs. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.