Automatic Code Optimization of Application Programs to Improve Performance of on Chip Cache in Multi-Core Systems

[1]Narendrasinh V. Limbad, [2]Dr.S.M.Shah
[1]ME scholar, [2]Professor
Computer Science and Engineering Department
Government Engineering college, Sec-28, Gandhinagar
Gandhinagar, India
[1]limbad.narendra09@gmail.com, [2]Sanjay shah r@yahoo.com

***Abstract* - The purpose of this paper is to propose code transformation techniques on the application program subjected to multi-core system (especially data and loop intensive application). So that the performance of the on-chip shared cache can be improved by converting the successive reuses of the same data elements by several computations (iterations) of loop nest into data locality. In this thesis we will propose the code mapping strategy to map the loop iterations to the several computing cores in such a way that each iteration pair are mapped to the cores which are having shared cache between them. So that successive reuses by these iterations of the loop nest can have the data locality. Our mapping strategy ensures that if two iteration are have very frequent data sharing then they should be mapped to the cores which are sharing their cache at the higher level in the hierarchy (means, the layers close to the cores) so data access latency can be minimized. Here our mapping strategy also considers the physical organization of the target multi-core machine. We also proposing customised scheduling strategy that will schedule the iterations mapped on each core. As mapping ensures only the placement of the iterations to the cores that are having the shared cache at any level in the hierarchy, this does not guarantee that iterations which shares the data will found that data to be shared when reuse actually takes place. It's the execution order (schedule) of the iterations within the iteration set mapped to particular core will guarantee that if two iteration are having shared data access between them, they use that data in closer proximity (means scheduling strategy specifies the execution order of each iterations assigned to particular core). So that the successive access can find that sharable data in cached.**

I. Introduction

There are very less research done towards cache optimization for multi-core systems[1][2]. Most of the code transformation schemes[3][6] proposed till now is intended for single core machine. Such optimization is nothing but keeping the most frequent data items (code fragments) local to the cache which is closer to the computing core. But in the case of multi-core architectures cache optimization is quite complicated as there is need of sharing of cache at various levels between the several cores. For this reason we need sophisticated scheme of mapping and scheduling of application programs to different cores by taking into account synchronization and data dependency issues.

Hence, our motive and goal is to develop the mathematical framework that illustrates cache mapping and scheduling strategies which ultimately improves cache performance in multi-core systems. And this mathematical model can be integrated with several major common compiler optimizations during compiler construction.

II. OVERVIEW OF THE APPROACH

For our loop intensive code, we apply our cache topology-aware code transformation scheme[5], which consists of two steps, namely, mapping and scheduling. Mapping determines how loop iterations are distributed across the cores, while scheduling decides the execution order of loop iterations assigned to each core. Both these steps take into account intra-core and inter-core data access and reuse patterns as well as the on-chip cache hierarchy of the target multi-core architecture.

In particular, the mapping component further contains two sub-steps: iteration-to-virtual core mapping and virtual core-to-physical core mapping. The former one employs a concept of virtual cores to represent a "virtual architecture" that has the same number of levels of caches as the target (physical) architecture, but infinite number of cores and cache components at each level, and determines an iteration to- core mapping such that data reuses with different reuse distances (represented by reuse vectors in our work) can be exploited at different layers of the cache hierarchy. The core idea behind this step is that, data reuses with shorter reuse distances[4] should be exploited in the higher level caches that have faster accesses, as they exhibiting more frequently during program execution and usually take place in a shorter period of time, whereas data reuses with longer reuse distances can be exploited in the lower levels of caches that have larger capacities.

Once the iteration-to-virtual core mapping is done, the second sub-step employs a folding function to map infinite virtual cores to limited physical cores by considering the data dependences across the cores. As a result, a loop nest is parallelized. After computation-to-core mapping, the scheduling component takes the iterations assigned to physical cores and determines an execution order for them, with the goal of tuning the timing of data sharing (reuse) among the cores so that data reuse can be converted into locality during program execution. Both our mapping and scheduling strategies are formulated and implemented in a linear algebraic framework.

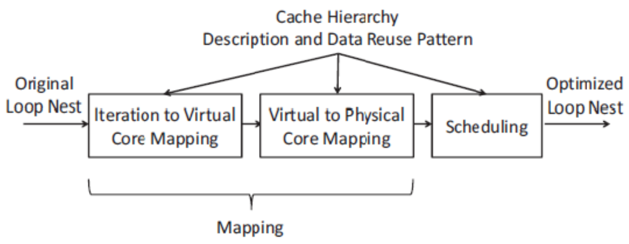
Cache Hierarchy
Description and Data Reuse Pattern
Original Loop Nest
Iteration to Virtual Core Mapping
Virtual to Physical Core Mapping
Scheduling
Optimized Loop Nest
Mapping

Figure 1: Mapping and Scheduling approach for Loop nest[5].

Few Code Mapping Related Abstractions

Iteration Vector: For l- level loop nest, we will represent every loop iterations in the form of vector of l-dimension. This creates l-dimensional vector space[7]. And dimension within the iteration vector can take the values from Lb to Ub (where Lb and Ub are Lower bound and Upper bound of the loop iterations). For example, we can represent iteration I of l-dimensional loop nest as iteration vector I = (i0,i1,i2…,il-1)T.

Reuse Vector: We describes reuse vector for every pair of iterations which are involved in data sharing (reuse). These reuse vectors describe that how frequently and in which pattern the data get reused between pair of iterations. Reuse vector is obtained from substracting given pair of iteration vectors. Means if I1 and I2 are two iteration vectors representing a given iteration then *Reuse vector*[5][8] r = I2 – I1 (provided that I2 is lexicographically greater than I1).

Reuse vector is the primary abstraction used for data locality optimization. For a given loop nest, we use R = (r1,r2,r3…,rq-1), called the reuse matrix of q different reuse vectors, to represent all its reuse vectors, where each column ri is a reuse vector identified in this loop nest. Clearly, from a performance point of view, we expect smaller reuse vector then a larger one (in a lexicographic sense on their absolute values). This is because small reuse vectors indicate shorter reuse distances, which means we have higher chances for catching the reused data in the cache.

Core Vectors: We represent every core in our multi-core architecture by a vector called Core Vector[5] to tag each core with multiple levels of caches. We represent each core using a vector that is composed of the core ID followed by IDs of all the caches it accesses in a top-down order (i.e., from the fastest/smallest cache to the slowest/largest). For example, in figure 1.1(c) the core vectors for core 0 and core 1 are c0 = (0, 0, 0, 0)T and c1 = (1, 1, 0, 0)T , respectively. If an architecture has s levels in its cache hierarchy (including cores as a separate layer), the core vector for core k (0 <= k < n), where n is the total number of cores, would be ck = (ck0, ck1,….,cks-1)T , where ck0 is the ID of core k and ck1,…,cks-1 are IDs of the caches that core k accesses at each level.

The motivation behind the use of core vectors is that, when two iterations (or threads or computations) are assigned to two cores with core vectors ci and cj (0<=i; j < n), respectively, by comparing the two core vectors, one can determine how these two iterations share cores and caches at different layers of a hierarchy, and therefore further decide whether the data reuse between these two iterations could potentially be exploited in the cache hierarchy, and if so, at what level. For example, if two iterations are mapped to two cores c0 and c1 in Figure 1.1(c) with core vectors (0, 0, 0, 0)T and (1, 1, 0, 0)T , the data reuse between these two iterations can be exploited in the L2 and L3 caches connected to these two cores, since the two core vectors have the same L2 and L3 IDs. As another example, if two iterations are mapped to two cores c0 and c4 that have core vectors (0, 0, 0, 0)T and (4, 4,2,1)T , the data reuse between these two iterations cannot be exploited in any cache in the hierarchy, since the two core vectors have no common entries, which means the corresponding two cores do not share cache at all.

Core Difference Vector: It defines the sharing patterns of computations (iterations) mapped to two cores. Let ci & cj be two core vectors where 0 <= I; j <n; and n is total number of cores and s is total number levels in cache hierarchy then the core difference vector[5] d,j is defined as, di,j = (d0,d1,….,ds-1)T where dt (0 <= t < s) is calculated as,

$$dt = \begin{cases} 0 & \text{if } ci,t = cj,t; \\ 1 & \text{otherwise;} \end{cases}$$

Here, our cache hierarchy in multi-core systems are of ‘tree-like‘ structure where main memory stays at the root, caches at various levels as internal nodes and cores as leaves of the tree structure. Therefore, if entry t in core difference vector contains 0 then all remaining entries (means, t+1, t+2.., ts-1) are 0.

Significance of virtual core

In, our core vector space there are certain points like (0,0,0,1)T is not the valid core vector. Hence, our core space is not continuous. To determine the proper mapping of iterations to the physical cores we uses continuous core space as our mapping matrix will map the iterations to virtual core space and then the folding function map the multiple virtual core from the core space to the fewer physical cores. As a result, a virtual architecture has a general "graph-like" structure; and any "tree-like" on-chip hierarchy can be considered as a special case of this virtual architecture. Virtual cores serve two purposes in our approach. Firstly, they provide a continuous linear space for applying parallelization/ mapping, which makes our approach easier to formulate. If virtual cores were not there, in our mathematical formulations (explained later), we would have to deal with constant bounds (e.g., number of cores), which would make implementation harder. Secondly, once we determine our mapping matrix for a virtual architecture (i.e., in terms of virtual core vectors), we can use them for any physical architecture that fits the template and customize only the folding function for each physical architecture.

III. CODE MAPPING PROCESS

Iteration to Virtual core mapping

To map every iterations of given loop nest having l - levels to virtual core architecture having s – layers there is need of mapping matrix. If two iterations I1 and I2 exhibit a data reuse between them with the reuse vector say r = I2 – I1 (assuming I1 is lexicographically lower than I2) at layer - k in the hierarchy then we should map these two iterations to the cores ci and cj such that their core difference vector becomes,

$$d_{i,j} = (\overbrace{\underbrace{1,1,1,1,1,}_{k\text{-entries}}\ldots 0,0,0}^{s\text{- entries}})^T$$

Means, two core vectors should have the shared cache at level- k; so that the data reuse get converted into locality.

Further, we can represent the core difference vector in terms of dot product of mapping matrix M and reuse vector r as,

$$M.r = (\overbrace{\underbrace{1,1,1,1,1,}_{k\text{-entries}}\ldots 0,0,0}^{s\text{- entries}})^T$$

Same for dependence vector d as,

$$M . d = (\underbrace{0,0,0,0,0,0}_{s\text{-entries}})^T$$

Here, our goal is to exploit the frequent data reuses at higher level in the hierarchy to reduce Average access latency.

Therefore, for given q reuse vectors we can have reuse Matrix R = (r0,r1,r2,…,rq-1) where r0,r1…rq-1 are the reuse vectors. Now, first we short these reuse vectors in the ascending order of their reuse distance. And then divide them into groups gi (0 <= I < s) according to separator vectors αi (0 <= I < s-1) (The calculation of αi is based on the capacity of the cache at level - i). And finally map them to the different layers in cache hierarchy by determining proper mapping Matrix M given by M = (mo, m1, m2, …,ms-1)T where, mi (0 <= I <s) are row vectors.

Equation (1) and (2) represents the iteration – physical core mapping as their RHS is the format for physical core difference vector. We know that if the entry numbered k in core difference vector is 0 then entries followed by k also 0. But, our virtual architecture is 'Graph-like' structure there may any number of arbitrary connections between the layers of cache hierarchy. For our iteration – virtual core mapping only entry numbered k should be 0.

So, we have

M.u = (x,x,x,x,0,x,x,x)T where, u is reuse vector or dependence vector

In, our mapping Matrix M = (mo, m1, m2, …,ms-1)T where, mi (0 <= I < s) are row vectors. We have m0.d = 0 if we want to map the iterations having the certain dependencies represented by d on the same core at layer 0 (core layer).

And same way mk.r = 0 if we want to exploit the data reuse at layer k (Where r is in group gk).

Determining Row vectors:

To determine the row vectors m0 to mk for our mapping matrix we will devide them in two groups m0 and mk where, 1 <= k < s (we also take s' = min(l , s) in place of s for optimization purpose)

Here, mk.I >= 0 (legality constraint) for every k to ensure valid mapping of I to virtual core space.

Determine m0:

We know m0.d = 0 to map the pair involved in data dependency on the sane core.
Here to determine m0 we need (l-1) linearly independent dependence vectors where l is number of levels in loop nest. Since, there may be more than (l-1) dependence vectors we will choose lexicographically smallest ones. And then determine m0 by following equality constraint:

$m0.\ d1 = 0;\ m0.d2 = 0;\ldots\ldots;.m0.dl\text{-}1 = 0;$ (3)

And for the reuse vectors not used in equality (3) we have locality constraint,

$$|m0.r| <= q0 \qquad (4)$$

To ensure that even though two iterations having dependencies are not mapped to the same cores. We will try to minimize the distance between two different core by minimizing the value of q0 (calculation of q0 is based on Fourier Motzkin Elimination[9])

Determine mk:

It is same as case – 1 we will choose (l-1) linearly independent reuse vectors from group gk (lexicographically smallest). Then apply following equality Constraint to derive mk:

$mk.r1 = 0;\ mk.r2 = 0;\ mk.r3 = 0;\ldots..mk.rl\text{-}1 = 0;$ (5)

And same way for the rest of the reuse vectors which are not selected in equation (5) we have

$$|mk.r| <= qk \qquad (6)$$

IV. Virtual Core to Physical Mapping

After the determination of mapping matrix M = (mo, m1, m2, …,ms-1)T. We determine the folding function F = (f0, f1, f2,…..,fs-1) where, s is total number of cache levels; to map each virtual cores to the fewer physical cores.

V. Conclusion

In this report literature we have described the code mapping and approach for loop intensive applications in linear algebraic framework. The goal of this paper is to improve the performance of the on-chip cache hierarchy of Multi-core systems by adopting the novel code mapping approach to map and schedule the loop iterations of the given loop nest having data reuses and data dependencies among them so that we can maximize sharing of data and minimize the interferences. Our approach can be used as a code optimization mechanism with the other code transformation techniques in compiler construction of multi threaded applications intended to run on multi-core platforms.

References

[1] Jernej Barbic " Multi-core architectures" 15-213, Spring 2007 May 3, 2007.
[2] A. Vajda, Programming Many-Core Chips, DOI 10.1007/978-1-4419-9739-5_2, © Springer Science+Business Media, LLC 2011.
[3] Neungsoo Park, Bo Hong, and Viktor K. Prasanna, "Tiling, Block Data Layout, and Memory Hierarchy Performance" IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 14, NO. 7, JULY 2003.
[4] Meng-Ju Wu and Donald Yeung, University of Maryland Technical Report UMIACS-TR-2012-01 "Understanding Multicore Cache Behavior of Loop-based Parallel Programs via Reuse Distance Analysis"
[5] Wei Ding Yuanrui Zhang Mahmut Kandemir Jithendra Srinivas Praveen Yedlapalli, "Locality-Aware Mapping and Scheduling for Multicores" CGO '13 23-27 February 2013, Shenzhen China.
[6] Micahel E. Wolf and Monica S. Lam, "A Loop Transformation Theory and an Algorithmto Maximize Parallelism" IEEE Transactionson Parallel and Distributed Systems, Vol-2 No.-4, October 1991.
[7] J.A.B Fortes and D.I. Moldovan, "Parallelism detections and Transformation techniques useful for VLSI Algorithms " J. Parallel Distributed Comput.Vol-2, pp.-277 – 301, 1985.
[8] M. Wolf and M. Lam. "A data locality optimizing algorithm" PLDI, 1991.
[9] A. Schrijver. Theory of linear and integer programming. John Wiley & Sons, Inc., New York, NY, USA, 1996.