Guiding the optimization of parallel codes on multicores using an analytical cache model

Diego Andrade, Basilio B. Fraguela, and Ramón Doallo

Universidade da Coruña, Spain
{diego.andrade,basilio.fraguela,ramon.doalllo}@udc.es

Abstract. Cache performance is particularly hard to predict in modern multicore processors as several threads can be concurrently in execution, and private cache levels are combined with shared ones. This paper presents an analytical model able to evaluate the cache performance of the whole cache hierarchy for parallel applications in less than one second taking as input their source code and the cache configuration. While the model does not tackle some advanced hardware features, it can help optimizers to make reasonably good decisions in a very short time. This is supported by an evaluation based on two modern architectures and three different case studies, in which the model predictions differ on average just 5.05% from the results of a detailed hardware simulator and correctly guide different optimization decisions.

1 Introduction

Modern multicore processors, which can execute parallel codes, have complex cache hierarchies that typically combine up to three private and shared levels [2]. There is a vast bibliography on the subject of improving the cache performance of modern multicore systems. Several works have addressed this problem from the energy consumption perspective [9, 6]. Other works try to enhance the cache performance in order to improve the overall performance of the system [3, 5, 8].

The Parallel Probabilistic Miss Equations (ParPME) model, introduced in [1], can estimate the cache performance during the execution of both parallelized and serial loops. In the case of parallelized loops, this model can only estimate the performance of caches that are shared by all the threads that execute the loop. This paper presents the ParPME+ model, an extension of the ParPME model to predict the effect of parallelized loops on private caches as well as in caches shared by an arbitrary number of threads, this ability enables the possibility to model the whole cache hierarchy of a multicore processor. The evaluation shows that the predictions of our model match the performance observed in a simulator, and that it can be an useful tool to guide an iterative optimization process.

The rest of this paper is structured as follows. First, Section 2 reviews the existing ParPME model and Section 3 describes the ParPME+ model presented in this paper. Then, Section 4 is devoted to the experimental results and finally Section 5 presents our conclusions and future work.

2 The Parallel PME Model

The Parallel Probabilistic Miss Equations Model [1] (ParPME) is an extension of the PME model [4] that can predict the behavior of caches during the executions of both parallelized and sequential codes. The scope of application of this model is limited to regular codes where references are indexed using affine functions. The inputs of the ParPME model are the Abstract Syntax Tree (AST) of the source code and a cache configuration. The replacement policy is assumed to be a perfect LRU. This model is built around three main concepts: (1) A **Probabilistic Miss Equation (PME)** predicts the number of misses generated by a given reference within a given loop nesting level. (2) The **reuse distance** is the number of iterations of the currently analyzed loop between two consecutive accesses to the same cache line. (3) The **miss probability** is the probability of an attempt to reuse a cache line associated to a given reuse distance.

Depending on whether the studied reference is affected by a parallelized loop in the current nesting level, the model uses a different kind of PME. The PME for non-parallelized loops, introduced in [4], is valid for both private and shared caches as in both cases all the iterations of the loop are executed by one thread and the corresponding cache lines are loaded in the same cache, no matter if the cache is private to each core or shared among several cores.

The modeling of parallelized loop is much more challenging, as the iterations of this kind of loops are distributed among several threads. In this case, each cache stores the cache lines loaded by the threads that share that cache and one thread can reuse lines loaded previously by the same thread (intra-thread reuse) or a different one (inter-thread reuse). The new PME introduced in the ParPME model [1] only covers the situation where a cache is shared among all the threads involved in the computation. However, current architectures include both private and shared caches. Moreover, many systems include several multicore processors, and thus even if the last level cache of each processor is shared by all its cores, when all the cores in the system cooperate in the parallel execution of a code, each cache is only shared by a subset of the threads.

3 The ParPME Plus model

This section introduces the main contribution of this paper, the ParPME Plus (ParPME+) model, which extends the existing ParPME model to enable the modeling of all the levels of a multicore cache hierarchy, including both the shared and private ones. Rather than adding more types of PMEs and extending the method to calculate the miss probability, our approach relies on transforming the source code to analyze in order to emulate the behavior of the threads that share a given cache level. Then, the transformed code is analyzed using the original ParPME model.

Figure 1, shows the general form of the aforementioned transformation. The top part of the figure shows the original loop, and the bottom part shows the transformed one that is associated to the representation used to analyze the behavior of a cache shared by a subset of ts_i threads. The index variable of the parallel loop that is being modeled can be used in the indexing functions of one or more dimensions of one or several data structures. The first step of the transformation eases the identification of which parts of a data structure are referenced by the iterations assigned to a subset of the threads that share the cache. With this purpose, each dimension of an array that is indexed by the index variable of a parallelized loop i multiplied by a constant

```
...
#pragma omp for schedule(static,bi)
for(i=0; i<Mi; i++)
  ...
  array[...][i*k][...];
  ...
```

```
...
#pragma omp for schedule(static,1)
for(i1=0; i1<tsi; i1++)
  for(i2=0; i2<Mi/(ti*bi); i2++)
    for(i3=0; i<bi; i3++)
      ...
      array[...][i2][i1][i3*k][...];
      ...
```

Fig. 1. Generic transformation to model cache shared among a subset of threads. Original code (top) and transformed code (bottom)

value k is split into three dimensions. If the split dimension has size N_i, the three resulting dimensions, defined from the most to the least significant one, have sizes $N_i/(t_i \times b_i \times k)$, ti and $b_i \times k$ respectively, t_i being the number of threads among which the iterations of loop i are distributed and b_i is the block size used to distribute the iterations of the loop among the threads. Since the product of the sizes of these three new dimensions is equal to the size of the original dimension, this modification changes the way the data structure is indexed but not its size or layout. In the case that N_i is not divisible by $t_i \times b_i \times k$, the most significant dimension must be rounded up to $\lceil N_i/(t_i \times b_i \times k)\rceil$, slightly increasing the size of the data structure by up to $t_i \times b_i - 1 \times k$ elements. For big values of N_i this will not affect significantly the accuracy of the predictions.

The second step of the transformation modifies the parallel loops so that (a) indexing functions can be generated for all the new dimensions defined and (b) such indexings give place to access patterns identical to those of the original code in the considered cache. For this latter purpose the transformation of each parallel loop i must take into account a new parameter, ts_i, which is the number of threads that share the cache that is being modeled at this point out of the t_i threads that participate in the parallelization of the loop. This transformation replaces each considered parallelized loop i of M_i iterations with three different consecutively nested ones of ts_i, $M_i/(t_i \times b_i)$ and b_i iterations respectively, where the first one is the outermost one and the last one the innermost one. Out of them, only the outermost one is parallel, being each one of its ts_i iterations executed in parallel by a different thread, while the two inner ones are executed sequentially by the same thread. This transformation also implies using the loop indexes of these loops for indexing the new dimensions defined in the first step. The mapping is always the same. Namely, the most significant dimension is indexed by the index of the middle loop, the next dimension is indexed by the index of the outermost (and parallel) loop, and the least significant dimension is indexed by the innermost loop.

The new code, or actually, its new representation, is almost equivalent to the original, replicating the same access patterns and work distribution, with a single critical difference. Namely, it only covers the iterations assigned to the subset of ts_i threads of interest, i.e. the ones that share the cache we want to model at this point, instead of all the t_i threads. Notice that this strategy assumes that if several threads share a cache, then they get consecutive chunks of the loop to parallelize. While this may not be always the case, this is a common and very desirable situation, as this tends to increase the potential cache reuse and reduce the overheads in case of false sharing. In those situations in which these benefits do not hold, for example when there is no possible reuse between the data used by different threads, the assumption is irrelevant, as the cache behavior should be the same no matter the sharing threads get consecutive chunks of the loop or not.

4 Experimental results

The model has been validated on two Intel platforms with very different features: a i7-4790 (with 4 physical cores), and a Xeon 2699 v4 (with 22 cores). The experiments try to prove that the model can accurately predict the cache performance of codes on real cache hierarchies, and that it can guide cache memory optimizations. With this purpose, we have built our experiments around three case studies: (1) The loop **fision** technique is applied to a code (2) The loop **fusion** technique is applied to a code. (3) A matrix multiplication implemented using the 6 possible loop orders (ikj, ijk, jik, jki, kij, kji).

The ParPME+ model is used to predict the number of misses generated by these codes in the different levels of the cache hierarchy. These predictions are used to calculate the memory cost (MemCost) associated to the execution of each given code, that is, the number of cycles it spends due to the cache hierarchy. This calculation has been made using the same approach followed in [3]. The predictions of the model have been compared to the observations made in the SESC cycle-accurate simulator [7] in order to assess their accuracy. The difference between both values is expressed as a relative percentage The average difference across all the experiments is just 5.05%, the accuracy thus being remarkable, from the 0.2% obtained for the **fusion** benchmarks in the Intel i7, to the 11.6% obtained for the **fision** benchmark in the same platform.

As a second step, the optimization choices made by the model are compared to those made using the guidance of the actual execution time. Figure 2 summarizes the results for the fision. The figure is composed of two columns with two graphs each. Each column represents the results for one of the two platforms. The number of threads used for the experiments is equal to the number of physical cores available in each platform. We have made sure that each thread is mapped to a different physical core using the affinity feature of OpenMP 4.0. The iterations of the parallel loop have been distributed cyclically among the threads in chunks of 16 iterations. This chunk size has been used in all the experiments of this section. The top graph in each column shows the memory performance predicted by the model for the versions without fision (fisionorig) and with fision (fisionopt) for different problem sizes. The bottom graph of each column make the same comparison using the average execution times of ten executions. These times and the model predictions lead to the same conclusion, this optimization is successful in both the i7 and the Xeon.

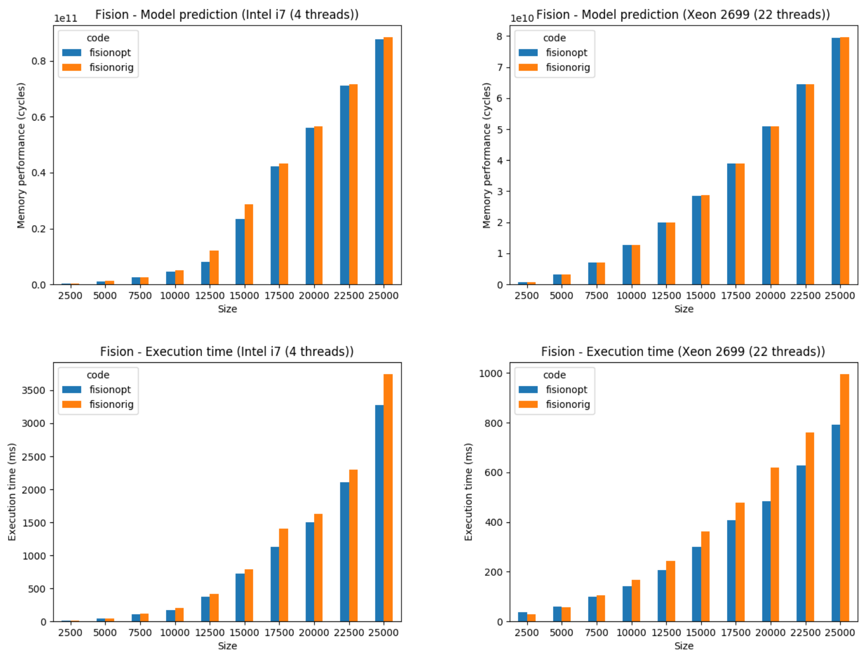

Fig. 2. Results for fision

The results for the fusion case study are shown in Fig. 3, which has the same structure as Fig. 2. In this case, the model also leads to the right decision, which is to apply the optimization in all the platforms and for all the problem sizes.

The results for the matrix multiplication case study are shown in Figure 4. In this case, we have to decide the best loop ordering to perform the matrix multiplication. The outermost `i` or `j` loop is the one parallelized. The results have the same structure as in the previous cases. According to the actual execution time, the ikj and kij ordering are the best ones, and this decision is accurately taken using the model.

All the predictions of our model have been obtained in less than one second no matter the problem size. However, the execution of the real codes or the simulator took quite longer for the larger problem sizes, ranging from the 3 times longer of **padding** in the Intel i7, to the 30 times longer of **matmul** in the Intel Xeon 2680. This is a crucial characteristic of our model, as we can evaluate several optimization choices in a fraction of the time required to execute the real codes.

5 Conclusions

This paper explores the possibility of using an analytical model of the cache behavior to guide compiler optimizations on parallel codes run on real multicore systems. The existing ParPME model introduced in [1], supported caches shared by all the threads participating in the parallelization. Nevertheless, several architectures present caches that are private to one threads or shared by a subset of the cores/threads participating in the execution of a parallel application. For this reason, a first contribution in

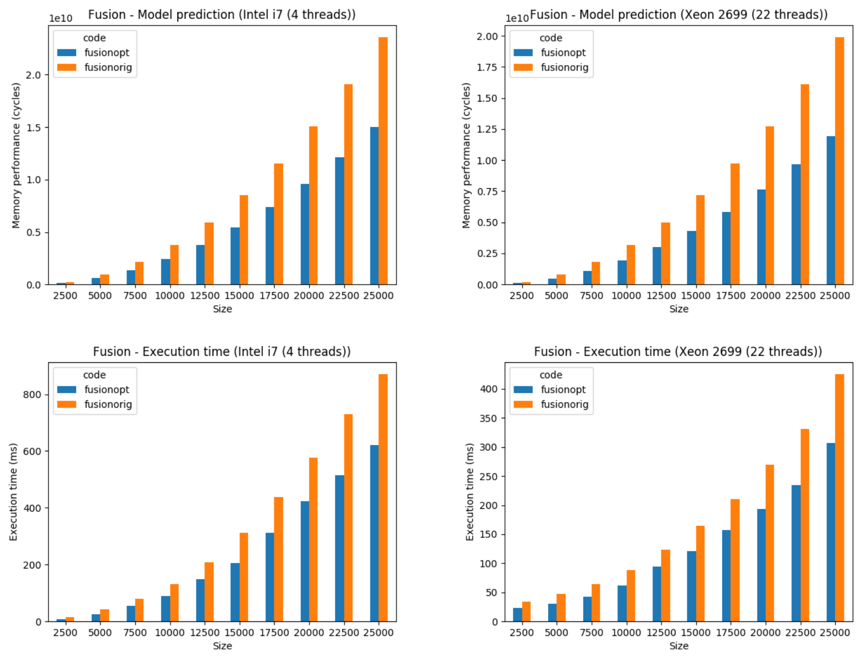

Fig. 3. Results for fusion

this paper, leading to the ParPME+ model, has been the development of a procedure which, by changing the representation of the code to analyze inside the model, allows it to also accurately predict the behavior of this kind of caches. As a result, the model can now analyze the behavior of the complete cache hierarchy of a multicore multiprocessor computer. Our experiments using three case studies and two architectures show that the model is a good guide to choose the most cache friendly optimization choice. This is not surprising, as the predictions of our model only differ by 5.05% on average from the observations of a cycle-accurate simulator. In addition, the model can provide its predictions in less than a second.

In the future, the model can be extended to model any of the missing hardware features present in some processors. It would also interesting to complement it with a CPU model.

Acknowledgment

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (TIN2016-75845-P), and by the Government of Galicia (Xunta de Galicia) co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04) as well as under the Centro Singular de Investigación de Galicia accreditation 2016-2019 (ED431G/01). We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of their computers.

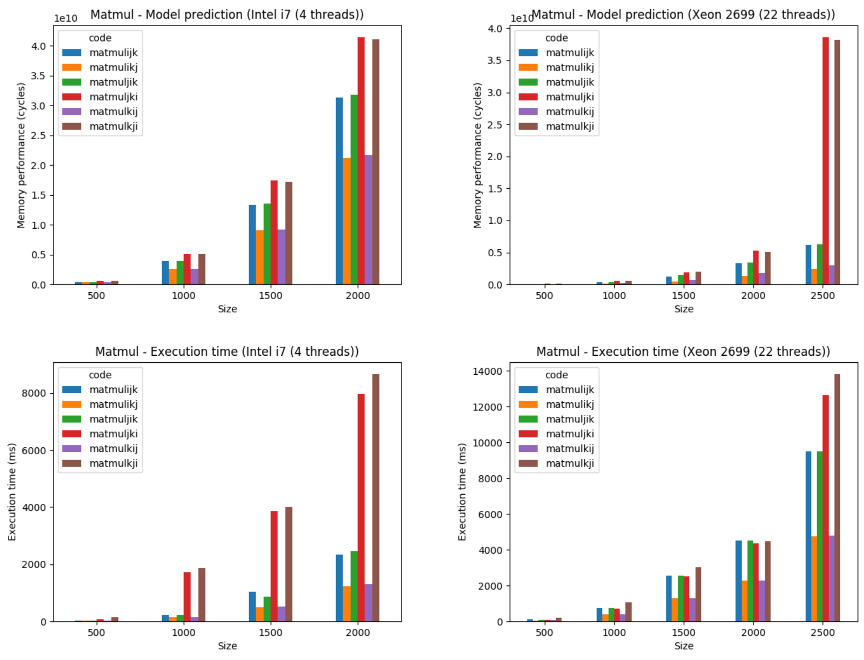

Fig. 4. Results for matmul

References

1. Andrade, D., Fraguela, B.B., Doallo, R.: Accurate prediction of the behavior of multithreaded applications in shared caches. Parallel Computing **39**(1), 36–57 (2013)
2. Balasubramonian, R., Jouppi, N.P., Muralimanohar, N.: Multi-Core Cache Hierarchies. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers (2011)
3. Fraguela, B.B., Carmueja, M.G., Andrade, D.: Optimal tile size selection guided by analytical models. In: Proc. of Parallel Computing. vol. 33, pp. 565–572 (2005), publication Series of the John von Neumann Institute for Computing (NIC)
4. Fraguela, B.B., Doallo, R., Zapata, E.L.: Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. IEEE Trans. on Comp. **52**(3), 321–336 (March 2003)
5. Ramos, S., Hoefler, T.: Modeling communication in cache-coherent smp systems: a case-study with xeon phi. In: 22nd Intl. Symp. on High-performance Parallel and Distributed Computing. pp. 97–108. ACM (2013)
6. Rawlins, M., Gordon-Ross, A.: A cache tuning heuristic for multicore architectures. IEEE Transactions on Computers **62**(8), 1570–1583 (2013)
7. Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., Montesinos, P.: SESC simulator (January 2005)
8. Schuff, D.L., Kulkarni, M., Pai, V.S.: Accelerating multicore reuse distance analysis with sampling and parallelization. In: 19th intl. conf. on Parallel Architectures and Compilation Techniques. pp. 53–64. PACT '10, ACM, New York, NY, USA (2010)
9. Zang, W., Gordon-Ross, A.: A survey on cache tuning from a power/energy perspective. ACM Computing Surveys (CSUR) **45**(3), 32 (2013)