Optimal Thread-to-Core Mapping for Pipeline Programs

Hao Luo, Chen Ding and Pengcheng Li
University of Rochester

Abstract
Pipelining is commonly used in multi-threaded code. In pipeline programs, the computation is divided into stages that perform different types of computations. Unlike in a data parallel program, threads in a pipeline program have different behavior. Because of the asymmetry, the performance varies significantly depending on how threads are grouped to use the same shared cache. It is time consuming to find the best grouping through exhaustively testing. As the number of threads increases, testing is not scalable.

This short paper presents model-assisted search. It profiles a program to find all-thread data sharing in training executions and predicts the cache performance for all thread groupings. From the prediction, inferior groupings are removed from consideration, leaving a smaller set of choices for test. The pruning uses a concept called the miss-count poset (partially ordered set). For an 8-thread pipeline program running on a hyper-threaded quad-core processor, we show that the best grouping is 70% faster than the worst and the proposed technique reduced the cost of testing by 64%.

1. Introduction
Pipeline parallelism has been one of the most widely used models, providing programmers the ability to apply a series of operations on a stream of data. It is used in many modern applications, including image processing, web service, search engine indexing, data compression, and gene sequence analysis, and supported by many contemporary parallel programming tools, such as Intel’s Thread Building Blocks (TBB) and OpenMP.

Despite their importance and popularity, the memory performances of pipeline parallel programs are difficult to model. Two challenges arise here. The first is the asymmetry of thread behavior, and the second is the difficulty in modeling locality due to data sharing. Unlike data parallel programs, worker threads in pipeline parallel programs can operate on different processing steps. Different groupings of threads on shared resources (shared cache in particular) sometimes bring very different impact on performance. Unfortunately, due to the exponential increase in the search space, the naïve approach of exhaustive testing is not scalable in finding the optimal thread grouping.

This paper investigates a more practical model-based approach based on off-line performance tuning aiming to alleviate this problem. It consists of two phases: profiling and testing. Profiling is conducted once to discover the inherent locality of the target program and input set while testing is conducted to verify the model on the target hardware platform. Profiling mainly serves to reduce the cost of testing by filtering out the theoretically inferior groupings.

2. Model-assisted Thread Grouping
In theory, there are \(\frac{n!}{k!} \times \frac{1}{\binom{n}{k}}\) ways to group threads for a \(n\)-thread pipeline program on \(k\) homogeneous caches. In our experiments, we target on four L2 caches in the system, each of which has 2 sharers and our benchmark dedup (from PARSEC benchmark suite [1]) has 8 threads and 5 stages. In order to investigate all thread groupings on four L2 caches on the 8-thread execution, a total of 105 tests are needed in exhaustive testing.

Our model-assisted approach performs following three steps in search for the optimal grouping:

1. Profile the application to collect the shared footprint [2].
   Shared footprint is our theoretical model used to predict the locality for all thread groups on all possible cache hierarchies. Since our algorithm of modeling the shared footprint is a single-pass and hardware-independent algorithm, the profiling needs to be done only once and can be conducted under any cache configuration.

2. For the target platform, predict the miss-count vectors of all groupings and construct miss-count poset (Section 2.1).

3. Prune the search space (Section 2.2) based on the miss-count poset. Test on the remaining choices.

Due to the limitation of space, we only elaborate on the procedure of pruning search space based on the knowledge of all thread groups’ cache performance on the target cache configuration.

2.1 Miss Count Poset
The tool of shared footprint enables us to model the cache miss count of all thread groupings from a single profiling execution. With the help of shared footprint metric, a vector
of modeled cache miss counts (one entry for a target cache) can be used to describe the locality of a thread grouping\(^1\). We call this vector the miss-count vector. We further define miss-count poset as a directed graph \(G_c(V,E)\), where \(c\) represents the cache configuration, which includes the number and capacity of the target caches in the system. \(V\) is the set of miss-count vectors of all thread groupings and \(E\) is a partial ordering relations on \(V\). A miss-count vector \(P_i\) is said to be “greater than” another miss-count vector \(P_j\) if each entry in \(P_i\) is greater than the corresponding entry in \(P_j\). Notions like “smaller than” can be defined in a similar way.

---

\(^1\) The entries in the vector are sorted in descending order.

2.2 Search Space Pruning

The miss-count poset can be an effective heuristic in grouping threads and enable us to selectively pick cache-friendly thread groupings to test. The intuition behind is following. If a grouping \(A\) has miss-count vector larger than that of another grouping \(B\), \(A\) is less likely to be optimal, at least no better than \(B\), from the perspective of locality. We then say \(A\) is dominated by \(B\) in this case. We therefore have more confidence in pruning \(A\) while keeping the groupings that have good locality. We applied this method on dedup and filtered out 37 out of 105 groupings from the 8-thread run.

An optimization can further bring a larger range of filtering by introducing the notion of slack rate \(r\) \((0 \leq r \leq 1)\). Now a grouping \(A\) is dominated by grouping \(B\) only when each of \(A\)’s miss count is larger than \(B\)’s corresponding miss count lowered by a factor of \(1 - r\). This extension could introduce cycles in the miss-count poset. In this case, we can compact the cycle and pick one node in the cycle as representative. In essence, we filter out the dominated cycles and only test the representative of a cycle. Exhaustive testings showed that the optimized pruning filtered 67 out of 105 groupings while many good groupings remained as Figure 2 shows.

---

![Figure 1](image1.png)  
**Figure 1.** A subgraph of dedup’s miss-count poset. Four miss-count vectors \(A\), \(B\), \(C\) and \(D\) with slack rate \((r)\) 0.01 are shown. The variation of \(A\) and \(B\)’s miss-count vector is considered “insignificant”, within range \((1 - r, 1 + r)\). The same applies to \(C\) and \(D\). Since \(A\) dominates \(C\), only one grouping (either \(A\) or \(B\)) needs to be tested.

![Figure 2](image2.png)  
**Figure 2.** Performances of dedup on different thread groupings. The statistics are collected on an 8-core machine with two cores sharing a L2 cache. A total of 8 worker threads were launched to execute a 5-stage pipeline. The performance of each setting was reported by averaging the results obtained from 10 tests.

3. Conclusion

In this paper, we introduced an approach to optimize the grouping of asymmetrical threads in pipeline programs using the analysis and modeling of shared footprint. This model enables us to predict the cache performance in all program groupings on all cache sizes. Based on the prediction, we pruned the search space and selected a subset of groups to test. Results obtained from our experiments demonstrated that the shared footprint model can be used to predict the locality of parallel programs. More applications of this model can be exploited in other areas such as locality-sensitive thread scheduling. We leave it as future work.

References
