POSTER: PLUM: Static Parallel Program Locality Analysis under Uniform Multiplexing

Fangzhou Liu
fliu14@cs.rochester.edu
University of Rochester
Rochester, New York

Dong Chen*
jameschennerd@gmail.com
National University of
Defense Technology
Changsha, China

Wesley Smith
wes.apps.2014@gmail.com
University of Rochester
Rochester, New York

Chen Ding
cding@cs.rochester.edu
University of Rochester
Rochester, New York

Abstract

Data movement has a significant impact on program performance. For multithread programs, this impact is amplified, since different threads often interfere with each other by competing for shared cache space. However, recent de facto locality metrics consider either sequential execution only, or derive locality for multithread programs in an inefficient way, i.e. exhaustive simulation.

This paper presents PLUM, a compiler solution for time-scale locality analysis for parallel programs. Experiments demonstrate that the prediction accuracy is 93.97% on average. PLUM is the first tool that analyzes data locality for parallel programs during compile time; in addition, it provides an approach for efficiently studying the representative interleaving pattern for parallel executions.

CCS Concepts • **Software and its engineering** → *Source code generation.*

Keywords Static analysis, Locality, Multithread

1 Introduction

Efficient utilization of fast memory like cache is a significant factor in performance and energy consumption; it is especially impactful in programs running in parallel, as modern architectures often share memory resources among cores.

One avenue for developers and system engineers to mitigate the cost of data movement involves measuring data locality. Recently, several locality models like reuse distance [5] and footprint [4] were developed. However, most existing locality techniques are designed for sequential programs, and it's infeasible to adapt them for parallel programs [2]. This

*This work was done when the author was a grad student at University of Rochester.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6818-6/20/02.

<https://doi.org/10.1145/3332466.3374510>

is due to the inherent non-deterministic property of thread interleaving: with fixed program input, repeated executions might generate different memory access sequences.

In this paper, we address this problem statically. However, a compiler can only extract thread-local memory access orders from the program flow. This raises the question: *can we derive the whole memory access sequence from the thread-local sequences?* Enumerating all possible interleavings is expensive: considering 2 threads with each thread has a length- n access traces, the number of possible interleavings would reach to 2^{2n} . Even with a small trace, $n = 150$, the number will become $1E+90$, larger than the number of atoms in universe. We develop a more light-weight solution called PLUM, which extracts and computes locality information through a pre-defined interleaving pattern, such as uniform interleaving which best simulates the round-robin scheduling used by practical systems. PLUM is the first all-timescale static analysis for multi-thread programs. We believe that PLUM is a good starting point for this research field.

2 Design and Implementation

PLUM is built on Static Parallel Sampling (SPS) [1], a tool that derives miss ratios accurately and efficiently during compile time. SPS first collects program control flow information during compile time and then uses it to compute data reuse statistics for sampled iterations. These reuse interval (RI) distributions will be further utilized by RI based locality models to predict the miss ratio curve. Compared with SPS, PLUM adds two components to handle thread interleaving: iteration space to thread mapping and interleaving simulation.

2.1 Iteration Space Distribution

PLUM assumes that only the outer-most loop will be parallelized and the Iteration Space (IS) is distributed evenly. The thread-local iteration spaces are obtained by evenly splitting the loop bounds for all outer-most loops. Identical to OpenMP static scheduling, if the iteration space is fully divisible by the thread number, the thread local iteration space partition size will be the same among all threads. Otherwise, threads with smaller id will be assigned with more iterations.

2.2 Interleaving Generation

To achieve 1-1 uniform interleaving, each reference (the leaf node in Figure 1) in the abstraction tree will be enclosed by a

Figure 1. Tree abstraction for 3-points stencil program.

ThreadNode (dark pink node in Figure 1) that enumerates all threads, thus enforcing 1-1 uniform interleaving. We use tuple $(A[i], \text{ref}_{A,i}, t_k)$ to represent array A accesses index i from reference $\text{ref}_{A,i}$ at thread t_k . Considering the above 3-point stencil program running in 2 threads, the memory accesses in sequential execution will follow the order “ $(A[0], \text{ref}_{A,0}), (A[1], \text{ref}_{A,1}), (A[2], \text{ref}_{A,2}), (B[0], \text{ref}_B) \dots$ ”, which will map to “ $(A[0], \text{ref}_{A,0}, t_0), (A[\frac{N}{2}], \text{ref}_{A,0}, t_1), (A[1], \text{ref}_{A,1}, t_0), (A[\frac{N}{2} + 1], \text{ref}_{A,1}, t_1), \dots$ ” after interleaving.

To avoid exhaustive enumeration, we apply sampling in PLUM. Similar to SPS, PLUM generates a specialized version of source code, known as the *sampler*, to sample reuses for each reference in the loop. The sampler first samples several iteration points from the entire iteration space and then traverses the abstraction tree starting from these sampled points. For each sampled point, it counts the number of accesses until it meets a reuse. The value of the counter is then the reuse interval¹.

3 Evaluation

This section evaluates the miss ratio curves predicted by PLUM. We compare them with ground-truth miss ratio curves derived from parallel execution traces gathered by instrumentation. We apply PLUM on 8 PolyBench/C 4.2.1 [3] benchmarks running on 8 threads. The input size is 128x128x128 for all level-3 nested loops, 1024x1024 for level-2 nested loops and 1024 for nature loops.

Accuracy Trace-based analysis will output different miss ratio curves for each execution due to the non-determinism of thread interleaving. Here we use the average miss ratio curve of 20 executions as our baseline. We apply the 1st norm to quantify the accuracy. Table 1 shows the accuracy for all 8 benchmarks. On average, PLUM predicts the miss ratio curve with 93.97% accuracy. With 10% sampling ratio, PLUM achieves 101.97x speedup on average compared to baseline.

Limitation The assumption for 1-1 interleaving fails to catch the non-determinism of parallel programs. Consider 2 threads with the same access pattern *abc*. All reuses under

¹Reuse interval means the number of accesses between use and its reuse. Considering trace “*abccba*”, the reuse interval for *a* will be 5.

Table 1. Accuracy of Miss ratio prediction. c denotes the number of cache lines. The table shows the accuracy with respect to different cache size ranges and the average.

Benchmark	Accuracy			
	< 10	$[10, 10^3)$	$\geq 10^3$	Average
stencil	89.78%	92.62%	98.69%	95.18%
2mm	88.70%	91.08%	99.92%	93.52%
bicg	77.41%	92.23%	100%	90.35%
atax	92.32%	96.53%	100%	96.01%
3mm	88.46%	90.87%	99.96%	94.34%
gesummv	87.75%	95.29%	100%	94.39%
gemm	94.99%	88.36%	99.75%	91.63%
mvt	88.23%	93.30%	97.75%	96.32%

1-1 interleaving will have interval 1, but in a possible real execution (e.g. *aabcbc*), reuse intervals could be more varied. Taking 2mm for example, PLUM predicts that 22% of accesses form reuse interval 1, while instrumentation shows 1% only. This gap causes the miss ratio for $c = 2$ cache predicted by PLUM to be 0.78, which is 17.89% inaccurate compared with the true miss ratio, 0.95.

4 Summary

This paper presents PLUM, a compiler solution that analyzes cache behaviour for parallel programs efficiently and accurately. Evaluation results on PolyBench show that PLUM achieves 93.97% accuracy on average. It is also a starting point to efficiently investigate the most representative interleaving patterns for parallel programs.

Acknowledgments

The authors would like to thank Haosen Wen and Wentao Cai for the discussion, Sylvia Si Cheng for the proof-reading and all the comments and suggestions of the anonymous reviewers. This research was supported in part by the National Science Foundation (Contract No. CCF-1909099, CCF-1717877, CCF-1629376) and an IBM CAS Faculty Fellowship.

References

- [1] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality Analysis Through Static Parallel Sampling. In *Proceedings of PLDI (PLDI 2018)*. Philadelphia, PA, 557–570.
- [2] Hao Li, Jialiang Chang, Zijiang Yang, and Steve Carr. 2017. Memory Distance Measurement for Concurrent Programs. In *Proceedings of the LCPC Workshop*.
- [3] Louis-Noel Pouchet and Tomofumi Yuki. 2016. PolyBench/C 4.2.1. <http://sourceforge.net/projects/polybench/files/>.
- [4] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. 2011. Linear-time Modeling of Program Working Set in Shared Cache. In *Proceedings of PACT (PACT '11)*. Washington, DC, USA, 350–360.
- [5] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Trans. Program. Lang. Syst.* 31, 6, Article 20 (Aug. 2009), 39 pages.