Project Proposal
Souradip Ghosh (souradip@cmu.edu)
Nikhil Agarwal (nikhilag@andrew.cmu.edu)
15-745

Compiler-Based Load-Store Ordering for a General-Purpose Dataflow Architecture

URL: https://souradipghosh.com/15745

Project Description: Our objective is to build compiler passes that enforce load-store (or general memory operation) ordering at the IR level of the compiler. This work is fundamental to an existing research direction of building a general-purpose low-power dataflow architecture that heavily utilizes compiler-based techniques in lieu of extra hardware. In such an architecture, we envision the compiler to handle memory operation ordering so that applications can run correctly on the dataflow architecture. Using the compiler's output, the hardware will enforce the ordering of memory operations via explicit data dependences.

Our project can be split into two components:

1. We look to build an *analysis and algorithm that generates a directed graph representing ordered memory operations in an application* (given any conservative memory analysis or pointer analysis output). The vertices of this graph would be the operations themselves (loads and stores) and the edges represent the precedence/order of those operations (specifically -- between loads and stores, stores and loads, and stores and stores). We would like this graph to have a minimal number of edges that represents load-store ordering, and we would like to prove that our algorithm/analysis is correct.
2. To generate precedence/ordering edges, the compiler must rely on existing memory analysis to determine if the memory locations accessed by these operations alias. This requires alias analysis, array dependence analysis, etc. to provide us with that information. Unfortunately, common out-of-the-box analysis techniques (i.e. in LLVM, etc.) are very quick and very conservative. Consequently, this component of our project will *look at more robust memory analysis frameworks in research* (ones that perform several pointer analysis, interprocedural alias analysis, array dependence analysis, etc.) that produce more accurate results (i.e. ones that can disprove memory dependences more readily and more definitively). Using these frameworks, we hope to *reduce* the number of precedence edges in our ordering graph.

We will evaluate our compiler passes on a set of common linear algebra kernels (DMV, DMM, SMV, SMM, DConv, SConv), a small set of graph kernels (BFS, DFS), and a small set of sorting kernels. Correctness will be determined by proof and/or by simulation on the envisioned hardware.

- *75% Goal*: Build an algorithm for generating the ordering graph. Explore one research framework on memory alias analysis and integrate it with our compiler pass.

- *100% Goal*: Build an algorithm for generating the ordering graph with a minimal set of precedence edges. Have a framework/reasoning for correctness and minimality. Explore multiple frameworks on memory alias analysis, array dependence analysis, etc. and integrate it with our compiler pass.
- *125% Goal*: Build an algorithm for generating the ordering graph with a minimal set of precedence edges. Formally prove its correctness and minimality. Explore multiple frameworks on memory alias analysis, array dependence analysis, etc. and integrate it with our compiler pass. Do a detailed evaluation on how/where these frameworks from research reduce the precedence/ordering edges generated in the graph.

Plan of Attack:
- Week 1: Literature review. Start building the algorithm. Test on applications with simple control flow.
- Week 2: Finish working on the algorithm. Test on applications with more irregular control flow, nested loops, etc.
- Week 3: Switch gears to exploring frameworks from research (see below on some suggested) and understand why they are better than the baseline (LLVM). Work on showing correctness/minimality of the algorithm
- Week 4: Integrate one of the frameworks into the compiler pass by replacing/augmenting the memory alias analysis provided by LLVM.
- Week 5: Finish up integration. Explore another framework. Work on formal proof of the algorithm, if time permits.
- Week 6: Wrap up, work on poster. Extra time here if we fall behind.

Milestone: Accomplish what is listed through Week 4.

Literature Search: Alias analysis in LLVM [1]. Memory dependence analysis frameworks in research [2][3]. We will need to do more research on building the ordering algorithm. We also suspect graph algorithms that identify transitivity, reachability, etc. will be needed.

Resources Needed: LLVM 12 (already set up). Everything else is assumed or provided by other members of our research lab (simulator for the dataflow architecture, etc.).

Getting Started: Some help needed with literature search, so we can look in the right direction for the analysis/algorithm. We've begun working on a version of the ordering algorithm for basic blocks (i.e. applications with no control flow).

[1] LLVM Alias Analysis Infrastructure. https://llvm.org/docs/AliasAnalysis.html

[2] S. Apostolakis, Z. Xu, Z. Tan, G. Chan, S. Campanoni, and D. I. August, "Scaf: A speculation-aware collaborative dependence analysis framework," in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 638–654. [Online]. Available: https://doi.org/10.1145/3385412.3386028

[3] Y. Sui and J. Xue, “SVF: Interprocedural static value-flow analysis in LLVM,” in Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 265–266.