Technical Perspective: Imperative or Functional Control Flow Handling: Why not the Best of Both Worlds?

Bill Howe
University of Washington
billhowe@uw.edu

There is a tension between an imperative style for control flow that has been shown to be easier to use, especially for novices, and a functional style for control flow that better exposes optimization opportunities, thereby making the optimizers more capable. The authors of "Efficient Control Flow in Dataflow Systems: When Ease-of-Use Meets High Performance" propose Mitos, a program rewriting framework that achieves the best of both worlds by borrowing program analysis concepts from compilers and lifting them to the distributed dataflow regime. Dataflow systems require significant data movement during processing, which can be highly redundant and wasteful in the context of iteration: naive execution plans can reprocess the same massive dataset on each iteration, and iteration $i+1$ must wait until iteration i is finished. The authors design a mechanism for labeling each intermediate result with its execution path, allowing the system to simultaneously manage complex branching situations while also implementing efficient processing via loop pipelining, all by reasoning about and comparing execution paths.

Mitos uses an intermediate representation that adapts static single assignment form (SSA) to the distributed computing reg-ime: Each variable is assigned only once; subsequent assignments to a variable are replaced with a new version. A single dataflow graph is produced by first applying simple syntactic preprocessing: unchaining sequences of calls into separate lines, then wrapping scalar assignments as bags to unify scalar and collection processing. Then, the SSA transformation divides a program into basic blocks. This process is straightforward, except that a variable assigned in two different branches (e.g., in the if-block and the else-block) will have different versions, only one of which is correct. In a compiler, the SSA transformation uses move instructions to implement ϕ-functions that make this decision — functions that determine which version of a variable to use in the presence of branching.

Mitos implements ϕ-functions and addresses a number of other control flow tasks through an elegant design: each set of intermediate results (bag) is labeled with the execution path that produced it. An execution path is a sequence of basic block labels. By reasoning about these paths, Mitos can allow a local physical operator on a single machine to process multiple branches simultaneously, as directed by the control flow messages it receives. For example, Mitos can select the latest iteration to produce a value of variable x by inspecting the length of the prefixes that end in x, and Mitos can determine which bag to work on by comparing the current path with the bag identifier.

Control flow decisions are broadcast to all machines independently of dataflow. To provide fault tolerance, Mitos allows one distinguished control flow manager (the coordinator) to decide when snapshots of basic blocks should be taken, and broadcasts these decisions to all machines who record and recover snapshots asynchronously.

The performance results are striking: multiple orders of magnitude improvements can be realized over Spark (which has no way to avoid reprocessing loop-invariant data) and Flink (which has no loop pipelining), while scaling better as well.

Previous approaches to iteration and control flow for distributed systems tended to be system-specific, such as HaLoop [2], or so general as to resist straightforward implementation in common systems, such as Blazes [1]. While some systems, including Naiad [3], offer powerful and general optimizations for iteration, they require programming in a functional style that can limit uptake and be incompatible with popular systems.

The authors' research strategy is apparent: to adapt ideas from a relevant area (compilers), and ensure interoperability for your optimizations through very general intermediate representations. Mitos is designed to work with any dataflow system that meets two modest requirements: a node in the dataflow graph must be capable of arbitrary stateful computation, and engine must support arbitrary cycles. These two requirements are met by a number of dataflow engines, including Flink, Naiad, Dandelion, and TensorFlow.

1. REFERENCES

[1] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *2014 IEEE 30th International Conference on Data Engineering*, pages 52–63. IEEE, 2014.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The haloop approach to large-scale iterative data analysis. *VLDB J.*, 21(2):169–190, 2012.

[3] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.