**Checkpoint Repair for High-Performance Out-of-Order Execution Machines**

1. **Problem**

Exception handling and branch prediction miss handling require repair mechanism that resets the machine to a previous state. Whenever an exception or a branch prediction miss occurs, a *consistent state* is stored just before the fault or just after the conditional branch instruction, or a *precise state*, will be used to restore the machine state. The proposes two repair mechanisms that restore the machine to the precise state in the exception and branch miss scenarios separately.

**B. Importance**

The out-of-order execution combined and branch prediction are two techniques for high performing, from an architectural standpoint. As demonstrated by the paper, using branch prediction and out-of-order execution effectively reduce cycles per instructions. Corresponding state recovery solutions need to be in place to counter the exception thrown during an out-of-order execution and after a branch prediction miss.

**C. Solution**

The state repair algorithms revolve around the concept of consistent state: machine state that no instructions to be issued will affect it and all the issued instructions have affected it. The algorithms utilize a number of data structures that keep track of the logical space information, which consists of states that help the algorithm to recover the memory locations and registers content of a previous checkpoint. The first algorithm used to recover from the out-of-order exception memorizes an array of a trailing backup states, which contain the potential consistent state corresponding to the instruction boundary. In the branch miss recovery, *current* logical space is written with the content from backup state just after the conditional branch and its program counter is written with the memorized alternative program counter addresses. Finally, an algorithm incorporates branch prediction miss and exception recovery by memorizing potential consistent states that could be used by both types of recovery.

Although the authors don’t provide metrics or evaluation data in the paper. But they state that the proof of correctness is available upon requests. The authors also have a lot of discussions on the factors that affect stalling, especially in Exception repairs.

**D. Comments**

For a paper about architectural algorithm, the authors did a fantastic job of making the algorithms relatively easy to understand, with all the state, data structures’ definitions and diagrams. I understood at least the general concept of the algorithm with not much prerequisite knowledge. However, if the authors could parallelly compare the algorithms of recovering exceptions and branch prediction miss, it could have been more explanatory to the readers. Since a lot of steps in the two algorithms are similar like the issue or the repair stage, directly seeing the differences between them would help understand the algorithms.

**Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches**

**A. Problem**

The prediction success rate for data dependent branch instruction has not been improving with the history-based prediction technique. The past records of branching destinations do not correlate with the future branching outcomes.

**B. Importance**

The data dependent branches have contributed more and more of the branch prediction misses and thus need to be addressed. Traditional methods of predicting data dependent branches rely mostly on some compiler output aiming to acquire the minimum operations necessary to compute the direction of the branch, or some heavy-weighted dependence graph computed on another thread. The light-weight data dependence approach defeats these techniques efficiency-wise.

**C. Solution**

Regarding a branch statement, the algorithm first initiates the generation of a dependence chain by backwards dataflow walking and collecting all instructions required for calculating a branch’s outcome. Then the chain is processed by a Dependence Chain Engine(DCE) which continuously computes the predictions in the program loop. There are multiple control modes for a data dependence chain in order to enhance the timeliness and the level of parallelism. For example, depending on the type of the branch, the chain execution could either be initiate when the predecessor initiates or finishes. The DCE also runs on a micro-architecture which contains a number of memory and ALU structures separate from core architecture, for the sake of performance, Due to their effect on the successor’s data dependence chain control mode, branch runahead separates branches into guard branches and affector branches. The key of detection of such type difference relies on the identification of merge points, which is a process done by a separate structure used to store the Wrong-Path program counters.

For evaluation, the authors use two metrics, which are Instructions Per Cycle(IPC) and Branch Mispredictions Per Kilo Instructions (MPKI). They test their models with a collection of uop instructions (SPEC 2017) which consist of arithmetic and logical operations, on four different configurations, which include the hardware with a baseline branch predictor and three Branch Runahead Predictors with register file restriction ranging from small to large. Both metrics’ results indicate a significant improvements from Branch Runahead Prediction compared to the baseline, and a positive correlation between the size of the register file and the improvement level. The authors also point out that high performance achieved by Branch Runahead might come with an unreasonable requirement of physical register file size in the microarchitecture.

**D. Comments**

The paper explains its procedure in details with examples which make it very easy to understand. It also provides a lot of analysis on the evaluation of its performance like the tradeoffs between timeliness and parallelism. However, it doesn’t mention how DCE architecture is implemented into the hardware. The readers might be curious to see how it will be added to the CPU chip and what the impact to the hardware would be.