Developing a Representativeness Measurement for Program Execution with Instruction-level Visualization

Federico Cifuentes-Urtubey[1], David Koppelman[2]
[1]University of Maryland, Baltimore County, Baltimore, MD 21250
[2]Division of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803

Abstract

Processor performance can be measured by the amount of instructions it executes in parallel and in prior to when it needs the information to compute a value. Data exchange is a major factor in the latency of program execution, causing inconsistency in performance. From simple arithmetic to evaluating interdependencies between inputs to a processor, there can be unpredictable branch mispredictions that further change the speed of program execution. PSE, an instructional-level visualization tool, displays where inefficiencies in execution occur and dependencies between instruction registers, creating a system to characterize traits of a program's execution. Measuring the instructions in a way that computes their coverage over an entire execution will allow users to focus on segments where instructions cause stagnant progress.

Background

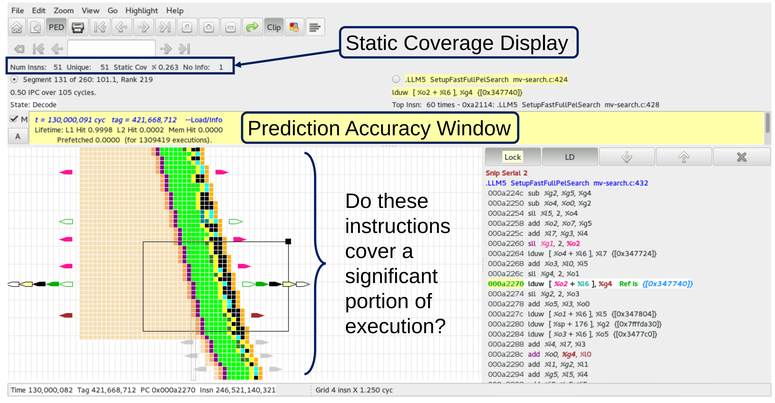
- PSE (processor simulation elucidator) is an instructional-level data visualization tool that displays steps, statuses, and cycles of instructions of a program execution.
- Registers are small amounts of storage on a processor, which are useful to access data the fastest compared to memory and cache. The number of registers on a processor depends on its architecture.
- Assembly language is a low-level language that lists single instructions line-by-line. PSE displays SPARC assembly implementations, and instructions can be arranged in various orders: chronologically, by most common instruction, and by prefetch accuracy.
- Each instruction is dependent on previous, incomplete instructions. It can depend on the register being used, a prior computed value, or a prediction from branching.

Methods and Implementation

- Using PSE to analyze the inefficiencies (i.e., cache misses, mispredictions) and memory or register information for instructions.
- Developing programs in C++ that implement conceptual steps in measuring representativeness:
  - A Representativeness score – Using a frequency table with string text to compute character coverage. It can also be used to address similarities in memory addresses, processor registers, etc.
  - Using container classes – Considers multiple instances of a target (i.e., memory address, certain instruction, branch path) to compare number of instructions between paths and an entire program execution
- Adding a static coverage feature to PSE – A percentage of static (unique) instructions in a segment that represents the amount of them versus the total number of instructions.

Visualizations

Using a *dataset* file, PSE displays an overview plot that refers to the execution rates and efficiency of a program. This is where the options to sort instructions are available. Black points portray the execution rates of each instruction (currently in chronological order), the blue points represent processor prediction rate, and other points regard prefetching (entering the processor) and committing (exiting the processor).



A pipeline execution diagram (PED) plot displays steps and memory addresses for each instruction, with the vertical axis listing single instructions and the horizontal axis measuring clock cycles in the processor. When instructions are found to be incorrect from the result of branch mispredictions, they are restarted for the correct branch and incorrect ones are doomed to be squashed (deleted).

Representativeness measurements will assist in identifying latency for appealing areas in a segment. If the visible portion of the PED is of poor performance, a representativeness measurement can encourage instruction rewriting and optimizing to reduce the lagging.

Center for Computation & Technology

Discussion

Representativeness can be measured in various ways using different data structures. The considerations are as follows:

Static Coverage – Using an integer count of unique instructions to compute a percentage of coverage within a segment of instructions.

Dynamic Coverage – Constructing an array of integers, with the size being the number of static instructions, to use as a frequency table to calculate a percentage of coverage.

Path Representativeness – Describing the jumps and branching in execution, targeting the predictions and mispredictions using a suffix tree.

Event Representativeness – Marking dynamic instructions with selected events, such as "correctly predicted" or "mispredicted". Load instructions can be marked with where the data is accessed or missed, "L1 hit," "L2 hit," "L3 hit," or "Memory hit" (cache miss). Forming a suffix tree based on the marked instruction stream will measure places where high latency occurs.

Conclusions

Organizing the instruction information based on the characteristics listed above saves time on manually searching for instructions that have slow prefetching or low efficiency to relate with its coverage over a segment or complete execution.

Future improvements will include implementing the suffix tree data structure representativeness measurements into PSE. These developments can be further applied into microarchitecture design research and hardware performance evaluation.

References

1. Koppelman, D., & Michael, C. (2014). Discovering Barriers to Efficient Execution, Both Obvious and Subtle, Using Instruction-Level Visualization. Proceedings of the First Workshop on Visual Performance Analysis (VPA) 2014, 36-41. doi:10.1109/VPA.2014.11
2. Kapoor, R. (2009). Avoiding the Cost of Branch Misprediction. Intel Developer Zone.

Acknowledgments

This material is based upon work supported by the National Science Foundation under award OCI-1263236 with additional support from the Center for Computation & Technology at Louisiana State University.