Contents

References

[1] The Sniper Multi-Core Simulator

[2] O. Tange (2011): GNU Parallel - The Command-Line Power Tool

[3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, The SPLASH-2 Programs: Characterizaion and Methodological Considerations, Proceedings 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995, pp. 24-36

[4] Bailey DH, Barszcz E, Barton JT, et al. The Nas Parallel Benchmarks. The International Journal of Supercomputing Applications. 1991;5(3):63-73. doi:10.1177/109434209100500306

[5] John L. Hennessy and David A. Patterson. 2017. Computer Architecture, Sixth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[6] S. M. Londono and J. P. de Gyvez. Extending Amdahl's law for energy-efficiency. 2010. International Conference on Energy Aware Computing, Cairo, Egypt, 2010, pp. 1-4, doi: 10.1109/ICEAC.2010.5702300.

[7] DW Wall. Limits of Instruction-Level Parallelism. 1991. Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems. 176-188.

[8] Jared Stark, Paul Racunas, and Yale N. Patt. 1997. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30). IEEE Computer Society, USA, 34-43.

[9] Tomasulo, R.M., 1967. An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Dev. 11 (1), 25-33.

Benchmark
`splash2-ocean.cont`
`splash2-radix`
`splash2-barnes`
`npb-is`
`npb-cg`

Reservation Station Entries
2
4
8
16
32
48
64
98
128

Table 1: Configuration parameters and values swept in the experiment.

1 Intro

- Describe Tomasulo's algorithm and what is uses to work at a high level.
- mention use of reservation stations
- mention sweeping reservation stations
- briefly describe overall results.

Tomasulo's algorithm is a computer hardware algorithm that allows for out-of-order (OOO) execution of instructions with the use of multiple functional/execution units.

dynamically determining when an instruction is ready to execute and renaming registers to avoid unnecessary hazards.

register renaming is provided by reservation stations (RS), which buffers the operands of instructions that are waiting to issue and each associated with an executing funcitonal unit. An RS fetches and buffers an operand as soon as it is available, which eliminates the need to read the operands from registers. Waiting instructions designate the RS that will provide their inputs/operands. When successive writes to a register overlap in execution, only the last one updates to the register, As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming [5].

reduces read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) hazards. RAW is avoided by waiting for inputs to be available through the common data bus (CDB) that delivers computed inputs before executing instructions, and WAW and WAR are avoided with register renaming [5]

improves the parallel execution of instructions that may otherwise stall due to those hazards under other kinds of architectures.

2 Experimental Setup

Simulations ran for an x86 architecture simulator, Sniper 7.3 [1]. Since this experiment looked to sweep instruction-level parallelism (ILP), each simulation was configured with 1 core to isolate the feature of multiple instructions simultaneouly in flight. The same default configurations were set in `gainestown.cfg` and `rob.cfg`. Those worth noting for purposes of this experiment are:

window size of 128;

commit width 128, default in `rob.cfg`. Reasonable width is 4, but due to simulations taking an excess of 24 hours and running, we decided the large default commit width would yield results in reasonable time for the given purpose.

L1, L2, and L3 cache sizes (64 KB, 256 KB, and 8192 KB, respectively, each using 64 byte blocks);

Figure **??** visualizes the topologies for all simulations since cache sizes remained constant.

Nine different reservation station (RS) entries across five benchmarks were swept, for a total of 45 simulations (see Table 1). The different configurations were simulated with three `splash2` benchmarks (`barnes`, `ocean.cont` and `radix` [3]) and two NAS parallel benchmark (`npb`) (`is` and `cg` [4])[1]. These were chose for the range of simple to complex memory access patterns. Two benchmarks from `nbp` were also included to show the effects of increasing ILP for applications that would benefit from more thread-level parallelism (TLP).

The workloads are briefly described as follow:

splash2-barnes : The `barnes` application implements the Barnes-Hut method to simulate interactions of systems of N-bodies (particles, galaxies, etc.) in 3D.

splash2-ocean.cont : The `ocean` suite of test studies large-scale ocean movements based on currents, and uses 4D array grids and a red-black Gauss-Seidel multigrid equation solver.

splash2-radix : The `radix` suite uses an iterative radix sort algorithm that generates histograms and has each processor permute array index keys, a process that depends on processors communicating in order to determine keys thorough writes.

npb-is : The NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB) are a set of benchmarks tuned for highly parallel workloads. The `is` kernal performs a sorting operation that is important as "particle method" code (ex. simulations of mechanics (solid, fluid, etc.) as discrete "particles"), testing both integer computation speed and communication performance. This benchmark excludes floating point arithmetic.

npb-cg : This benchmark uses a conjugate gradient method that computes the smallest eigenvalue of a large, sparse symmetric, positive definite matrix. It tests irregular, long distance communication and employs unstructured matrix vector multiplication.

[1]Originally picked `ua`

All the simulations ran concurrently using bash script(s) and GNU `parallel` shell tool [2], and post processing of the data were handled with python (v2.7) and bash scripts (included separately). Simulations ran on a python virtual environment and in a detached `tmux` session, due to long duration of the experiments. Sniper provided data processing tools used were: `gen_topology.py`, `cpi-stack.py`, and `mcpat.py`.

3 Results & Analysis

4 Conclusion

5 Appendix: Raw Post Processed Data

5.1 benchmark

5.1.1 Power Results

5.1.2 CPI Stacks