Recycling Data Slack in Out-of-Order Cores using SimpleScalar

Justin Havas, Grant Mak
ECE Department, *Duke University*

Abstract

Modern processors favor reliability and robustness over bandwidth and performance. As such, they set their timing margins conservatively to ensure that outputs are reliably and consistently obtained. Furthermore, as instructions become more and more complex, the data paths that these instructions take also grow more complex, causing large timing differences between worst and best case scenarios. Together, these two issues create data slack, which is the portion of a clock cycle where there are unused data paths present in the circuit. Such paths include under-utilized or idle functional units, and unscheduled, dispatched instructions. In this paper, we recreate the ReDSOC design, which aggressively finds and recycles data slack in out-of-order cores. After implementing ReDSOC with SimpleScalar, we found a percent speedup ranging from 3.9% to 24.9% depending on the issue-width and the compiler configuration. This is comparable to the 5% to 25% percent speedup obtained in the original ReDSOC implementation.

Objectives

The objective of this project is to recreate and confirm the results of the *Recycling Data Slack in Out-of-Order Cores* research paper from the 2019 IEEE International Symposium on High Performance Computer Architecture. This paper proposed a way to manage data slack, which is the portion of a clock period where nothing is happening, due to unutilized circuitry paths inside the hardware. Our plan is to recreate the results using the SimpleScalar sim-outorder simulator and the GCC, GO, and Anagram benchmarks in order to observe the result of our ReDSOC implementation.

Methods

SimpleScalar simulations of the ReDSOC design were run using the sim-outorder executable. In order to stay consistent with the prior paper's processor specifications for a small, medium, and large processors, we needed to override flags in sim-outorder to closely match such specifications. The key specifications of the small processor were a width of 2, LSQ/RUU size of 8/16, and 2 integer ALUs. The key specifications of the medium processor were a width of 4, LSQ/RUU size of 32/64, and 4 integer ALUs. Lastly, the key specifications of the large processor were a width of 8, LSQ/RUU size of 64/128, and 6 integer ALUs.

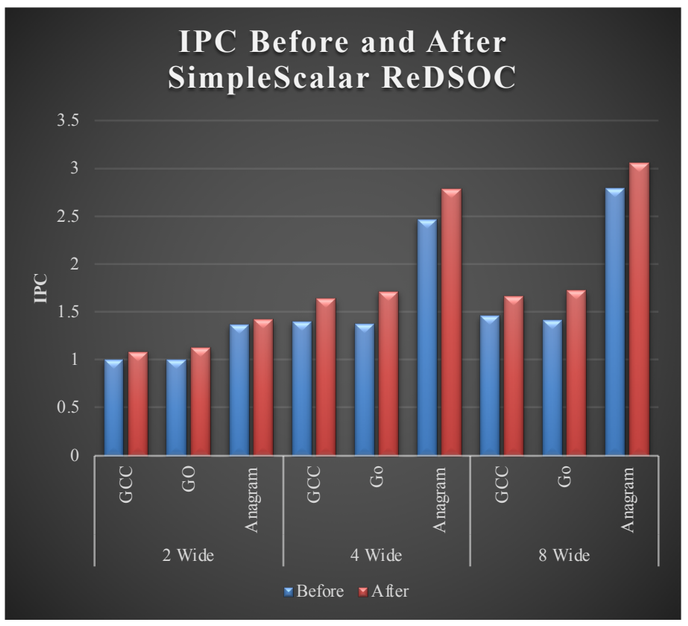
Simulations were then ran with the GCC, GO, and Anagram benchmarks. These benchmarks each have around 40-45% of their total instructions as ALU instructions. They also allow us to observe performance with different magnitudes of total instructions in the program. Anagram has a magnitude of 10 million total instructions, while GCC and GO have magnitudes of 100 million and 1 billion total instructions, respectively. These benchmarks help provide results that allow us to observe the ReDSOC implementation from different perspectives. Performance measurements were calculated based off total simulation IPC provided by the sim-outorder statistics.

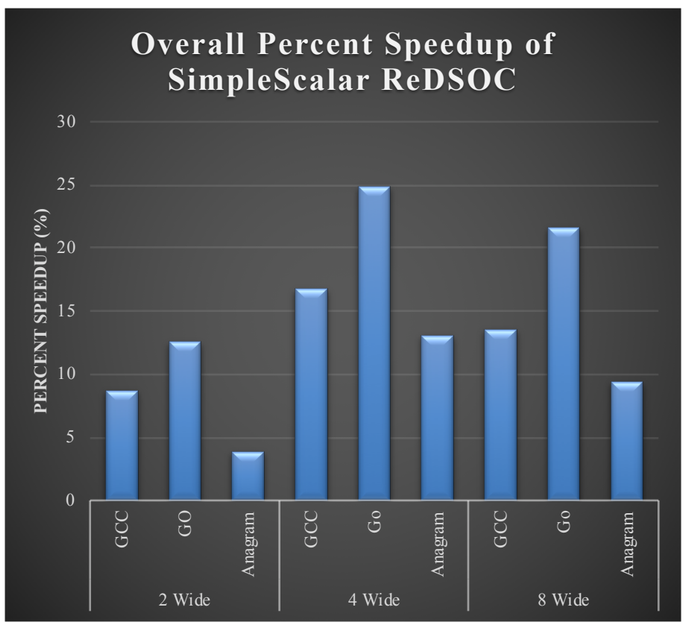
Results

The SimpleScalar simulation provided information about the IPC of a benchmark code sequence for three different compilers - GCC, GO, and Anagram. After running the simulation, we found a speedup for all possible configurations after implementing ReDSOC. When comparing the IPC before and after the ReDSOC implementation, we found an experimental range from 3.9% to 24.9%. The bottom of the range came from the 2-wide, Anagram compiler configuration, and the top of the range came from the 4-wide, GO compiler configuration. On average, we obtained percent speedup of 13.8%.

We found the experimental range of the integer ALU instruction percent speedup to be 10.3% to 99.1% - corresponding to the same configurations as earlier. These speedups were calculated using Amdahl's Law by plugging in the overall speedups determined above and the percent of total integer ALU instructions. On average, there was a speed up of 46.5% for the integer ALU instructions.

The medium or 4-wide processor configuration reported the highest average speedup of 18.2% across all three benchmarks. The large or 8-wide processor configuration reported the second highest average speedup of 14.8%, while the small or 2-wide processor configuration reported the lowest average speedup of 8.4%.





Conclusion

In analyzing the nine comparisons shown in the results sections, our experimental range across all processor configurations and benchmarks is 3.9% to 24.9% which is extremely close to the 5% to 25% range mentioned by the paper. This varying range of increase in overall performance is mainly an attribute of the varying increase in integer ALU speedup. Due to this significant average speedup, Amdahl's law helps show that increasing integer ALU performance can correspond to a significant overall performance increase even when the percentage of integer ALU instructions run are approximately 40%.

One observation that is unsettling is the variability of the performance increase gap. This gap was seen to be generated experimentally when benchmarks ran on processor configurations that did not allow them to maximize their execution throughput due to the smaller width of the processor or smaller number of functional units. This seems to generally be confirmed by increasing IPC when processor width increased. However, the IPC increase from the small configuration to the medium configuration made a much more significant impact than from medium to large configurations.

We believe this is the case because the ReDSOC implementation requires at least one dependent instruction to be assigned to its own functional unit at the same time the parent is assigned to its functional unit. This scenario is less likely to happen when there are less functional units in the core which would cause processors of smaller width or integer ALU functional units to miss out on capitalizing on data slack as often as processors with more functional units.

In general, this observation corresponds with the results of the initial paper. The benefits of slack increase with the size of the core as a larger core provides more idle functional units that can capitalize on transparent data flow. Larger cores also come with a larger RUU which allows for more data slack dependent instructions to be scheduled more aggressively. Such a scheme allows for the possibility of multiple dependency chains utilizing the benefits of recycling data slack.

In conclusion, ReDSOC has been shown to work well with long sequences of data-dependent instructions. This is especially true with sequences that contain primarily ALU operations. Luckily, this is a common trait amongst instruction sequences and modern, general purpose registers. Therefore, ReDSOC shows a lot of promise, and is a good way to increase performance in general purpose, multiple issue, out-of-order cores.

References

1. G. S. Ravi, M. H. Lipasti, "Recycling Data Slack in Out-of-Order Cores," in *HPCA* 2019
2. G. Hinton, D. Sager et al., "The microarchitecture of the pentium 4 processor," Intel Technology Journal, 2001.
3. D. Brooksand M. Martonosi,"Dynamically exploiting narrow width operands to improve processor power and performance," ser. HPCA, 1999.
4. O. Ergin, D. Balkan et al., "Register packing: Exploiting narrow-width operands for reducing register file pressure," ser. MICRO, 2004.
5. G. H. Loh, "Exploiting data-width locality to increase superscalar execution bandwidth," ser. MICRO, 2002.
6. E. Gunadi and M. H. Lipasti, "Narrow width dynamic scheduling," Journal of Instruction-Level Parallelism, 2007.