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

Justin Havas, Grant Mak
justin.havas@duke.edu, grant.mak@duke.edu

Duke University
ECE 552: Advanced Computer Architecture I

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.

Introduction

Data slack is defined as 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. Data slack rises from the fact that many modern processors aim for reliability. As a result, modern processors have large timing margins, in order to account for possible variation from environmental variables like temperature and voltage fluctuations and for any workload size for the functional units inside the processor. If these variations were not accounted for, then the work performed inside a pipeline stage could take more than one clock cycle, disabling the possibility for a fully synchronous design. Thus, performance and efficiency are traded off in order to increase reliability.

Data slack is particularly hard to manage, as it is affected by parameters like operands, data type, and instruction type. Additionally, since newer ISAs try to increase bandwidth per fetched instruction, data paths must be larger and more complex, and have a lot of variance between their best and worst case scenarios. Compilers can make use of these complex data paths in some situations, but when they are not being used, they still contribute to critical timing, increasing data slack. Data slack must be located and recycled in order to increase bandwidth.

Older methods of dealing with data slack include timing speculation, specialized data-paths, and operation fusion. Timing speculation aim to reduce slack by increasing clock frequency and lowering voltage. It tracks the frequency of timing violations and predicts critical

instructions, so that they can better be scheduled, and compute in less time. However, the design overheads from implementing timing error detection and recovering has a significant effect on performance. Specialized datapaths are used in processors for repeated, hot code. These data paths are not unique, and thus are not helpful in the general case. Operand fusion tries to fuse, and complete multiple independent instructions in the same cycle. However, optimizing the instruction flow using this technique creates significant design overhead, while unoptimized code provides limited opportunity. On the other hand, ReDSOC is able to avoid all of these issues.

ReDSOC Design

ReDSOC dynamically finds and aggressively recycles data slack to the maximum extent possible. It does this by first identifying data slack producing instructions. Then, it attempts to remove or recycle this data slack by allowing dependent, data slack consuming instructions to begin execution at the instant the data slack producing instruction has completed execution. In addition, ReDSOC is timing non-speculative, and does not change voltage or frequency when accelerating operations. ReDSOC also conserves data slack across any sequence of operations, so all of it is available to recycle and share with other operations. Finally, adjacent operations do not need to fit into single cycles, nor do operations necessarily have to be scheduled or rearranged in order to observe the benefits of ReDSOC.

ReDSOC can be broken up into two main pieces. The first is the ability to recycle data slack via a transparent-flow based data bypass network between the ALU functional units in the core. The other is performing slack aware instruction scheduling by optimizing wakeup and select logic in order to properly utilize the recycling execution mechanism.

Recycling Slack via Transparent Bypass Network

In order to allow dependent, data slack consuming instructions to begin execution at the instant the data slack producing instruction has completed execution, we must use transparent data flow with flip-flops. By “transparent”, we mean that any ALU functional unit is given the ability to receive the data output from any other ALU functional unit. The data flow itself is handled by a bypass network that feeds from the output of a functional unit to the input of other functional units. We specifically only incorporate this network for integer ALU functional units since we know that an operation within those can complete in less than a cycle. Other multi-cycle operations like floating point and memory do not benefit from this transparent execution and so their functional units are left unchanged.

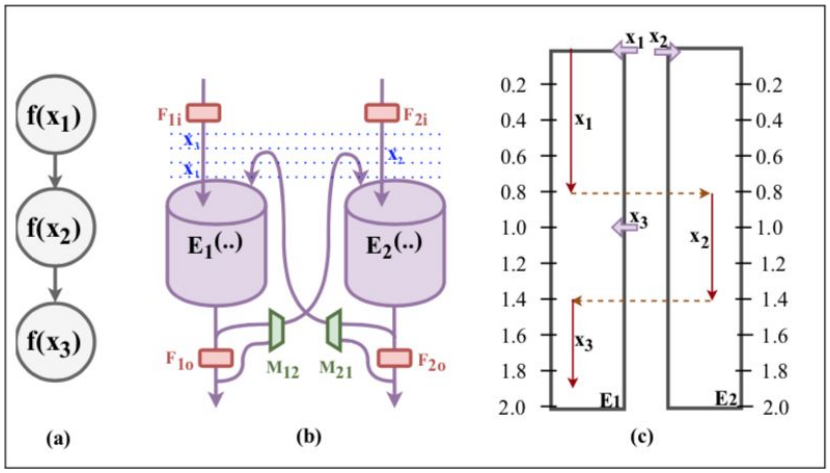

Figure 1: Data Slack Recycling Design and Example

Figure 1(b) depicts a simplified design of the transparent data flow bypass network with two functional units labelled E_1 and E_2. There are four total flip-flops going into and out from functional unit E_1 (F_{1i} and F_{1o}) and E_2 (F_{2i} and F_{2o}). Lastly, each functional unit has a multiplexer that acts as the transparent bypass network from one functional unit to the other. For instance, M_{12} goes from the output of E_1 to one of the inputs of E_2, and M_{21} goes from the output of E_2 going to the input of E_1. The inputs of these multiplexers are before and after F_{1o} and F_{2o} in order to bypass a transparent value located before the flip-flop or forward the output of the functional unit clocked by the flip-flop. For ReDSOC to properly recycle, the transparent input is bypassed whenever data is required to flow through non-clock boundaries. Otherwise the output of the flip-flop is used for forwarding between functional units.

An example of the execution of this mechanism can be seen by observing the diagrams provided in Figure 1(a) and (c). Figure 1(a) depicts the dependency chain of three ALU operations all of which will produce data slack during their computation. x_1 is the parent of operation x_2 and x_2 is the parent of operation x_3. Figure 1(c) then demonstrates how these operations would execute through the ReDSOC design. For this example, we will assume operation x_1 takes 0.8ns, operation x_2 takes 0.6ns, and operation x_3 take 0.5ns which are all operation latencies less than the clock cycle of 1ns. At time zero, both operations x_1 and x_2 are brought to their own functional unit, E_1 and E_2 respectively. Operation x_1 is the grandparent in the dependence change and has all of its dependencies satisfied so it begins computation at time zero and finishes at time 0.8ns. Operation x_2 waits until time 0.8ns when it can receive the transparent value given from the bypass network. x_2 can then begin computation at 0.8ns and will finish at 1.4ns. A new cycle begins at 1.0ns which clocks the output of operation x_1 into flip-flop F_{1o} and continues feeding it through the bypass network for operation x_2's computation. At this same instant, operation x_3 is brought to functional unit E_1 and is awaiting the result from operation x_2 before beginning computation. Finally, at time 1.4ns, operation x_2 finished computation and its transparent output is bypassed to the input of functional unit E_1 so that operation x_3 can begin computation at 1.4ns and finish its computation at 1.9ns. Thus, by the

time of the next cycle at 2.0ns, the outputs from operation x_2 and x_3 are stored into the flip-flop for any possible forwarding and the two functional units E_1 and E_2 are freed up for their next operations.

As a result of this example, we are able to see that ReDSOC provided the ability for three instructions to complete within 2 clock cycles instead of traditionally 3 clock cycles. The latency savings that this mechanism provides become more and more significant when there are millions of high slack ALU operations being run during a program. Therefore, the execution latency of the instructions decreased while the utilization of the functional unit computation increased.

Slack Aware Scheduling

As seen in the previous section, the benefits of transparent data flow bypass networks and flip-flops can only be utilized when opportunistic, slacking instructions are scheduled in an order that allows dependent instructions to reach functional units at the same time as the parent. This means that the scheduling logic needs to be made aware of the potential early completion of operations within their clock cycle. The scheduler also needs to be aware that when a data slack producing operation is expected to produce slack, a slack consuming operation needs to be assigned early enough so that the consumer can begin execution immediately after the producer finishes.

One of the ways ReDSOC adds this awareness is by adding Slack Tracking capabilities to the scheduler. These capabilities entail calculating and keeping track of an operation's completion time based on the execution times and data slack producer's completion times. These allow the scheduler to decide whether or not there is a slack opportunity available. If one is available, it will then attempt to assign a data slack consuming operation to its own functional unit and adjust the control signal to the functional unit's bypass network to allow transparent data flow.

Other ways ReDSOC manages slack awareness include attempting the decrease the latency of slack awareness using speculation. Eager grandparent wakeup speculatively wakes up grandparent operations so that parent and child operations could be issued in parallel. Selection logic must be skewed in order to incorporate this speculative wakeup, so non-speculative operations are prioritized over speculative ones. However, we decided not to implement the speculative aspects of ReDSOC in our SimpleScalar implementation, so eager grandparent wakeup and skewed selection logic were not incorporated.

SimpleScalar Implementation

SimpleScalar is a system software infrastructure used to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification. Within the tool set, sim-outorder is a trace-driven simulation of a superscalar, out-of-order processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. This was the main tool we used from SimpleScalar in order to simulate the effects of ReDSOC.

Since sim-outorder is a trace-driven simulation, we focused on reflecting the ReDSOC design through timing. Therefore, instead of implementing multiplexers in order to forward

results from one functional unit to the other, we would construct the operation latency of an instruction recycling data slack in order to reflect such forwarding. There were certain key sections within the sim-outorder code that were used in order to implement ReDSOC: ruu_dispatch() and ruu_issue().

RUU Dispatch

ruu_dispatch() is a function that is called during every cycle of the simulation and is in charge of dispatching instructions from the fetch stage into the register update unit. It is in this stage where the scheduler decides whether to add an instruction into the ready queue in order to be issued in the next pipelined stage. Before implementing ReDSOC, the scheduler would only add an instruction into the ready queue when it is determined to have all its operands ready or no dependencies.

In order to implement ReDSOC within the scheduler, we needed to allow the scheduler to add instructions with dependencies into the ready queue as well, so that a dependent instruction and its parent can reach their own functional units at the same clock cycle and go on to recycle data. Thus, the scheduler was given code to add instructions with dependencies into the ready queue based off a couple of conditions.

The first condition was that the instruction we were observing had a parent already in the ready queue or the event queue. This was an important condition to satisfy since instructions that are capitalizing off of recycling data slack need to be able to know who their parent is so that their operation and issue latency could be properly computed. This condition is also important because a dependent instruction needs to be assigned to a functional unit before the parent instruction gets to the writeback stage or else it will be too late to capitalize off the parent's data slack. This is because once at writeback, the parent instruction is able to propagate its output values to its dependent instructions in the dispatch stage which takes away the original dependency in the dependent instruction.

The other condition was that the instruction with dependencies and its parent both needed to use an integer ALUs for their functional units. This was important since ALU functional units can consistently perform operations under a clock cycle, and thus, provide opportunities for recycling data slack.

RUU Issue

ruu_issue() is a function that is called during every cycle of the simulation and is in charge of issuing instructions inside the ready queue to available functional units. If an instruction from the ready queue is able to be assigned to an available functional unit, the instruction will be removed from the ready queue and added to the event queue which keeps track of the order in which functions are written back in the write back stage after execution.

In order to implement ReDSOC within the issue logic, we needed to impact the issue latency and operation latency of dependent instructions capitalizing off recycling data slack. Issue latency is the number of cycles in which a functional unit should be marked as unavailable. In order to reflect recycling data slack with regard to issue latency, we made instructions recycling data slack have an issue latency equal to the issue latency of the parent plus the issue latency of the dependent instruction. This was determined based off the

assumption that integer ALU operations will not perform faster than six tenths of a cycle. With such an assumption, we know that the dependent instruction recycling off of data slack that is brought to a functional unit at the same time as its parent will have to hold on to the functional unit for an additional cycle before completing execution and making it available.

We also needed to reflect the effects of the ReDSOC design in the operation latency of dependent instructions. Operation latency, before the addition of ReDSOC in SimpleScalar, was defined as the number of cycles it takes for an instruction to finish execution. This operation latency is used to sort the instructions in the event queue so that instructions get written back in the write back stage based off the time they finish execution. In order to accommodate the ReDSOC implementation in SimpleScalar, we needed to allow operation latency to be defined by less than a cycle since integer ALU operations create data slack by finishing execution in less than a cycle. Therefore, the operation latency within SimpleScalar was adapted to be the number of tenths of a cycle it takes for an instruction to finish execution.

Once operation latency was changed to dealing with tenths of a cycle, we were able to add logic to approximately calculate the time at which dependent instructions capitalizing off data slack should be finished executing. This operation latency for dependent instructions was defined to be the operation latency of the parent instruction plus the operation latency of the dependent instruction.

Methodology

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 small, medium, and large processors, we needed to set specific flags in order to closely match such specifications. The table below demonstrates the specific flags that were overridden when running the sim-outorder simulation. Flags not listed were kept at their default values.

Table 1: Overridden Flags in Sim-OutOrder Simulation

Flags	Small	Medium	Large
issue:width	2	4	8
decode:width	2	4	8
commit:width	2	4	8
mem:width	2	4	8
lsq:size	8	32	64
ruu:size	16	64	128
res:ialu	2	4	6
res:fpalu	1	3	4

cache:dl1	dl1:2048:32:1:l
cache:dl2	dl2:65536:32:1:l

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 the total simulation IPC provided by the sim-outorder statistics.

Results

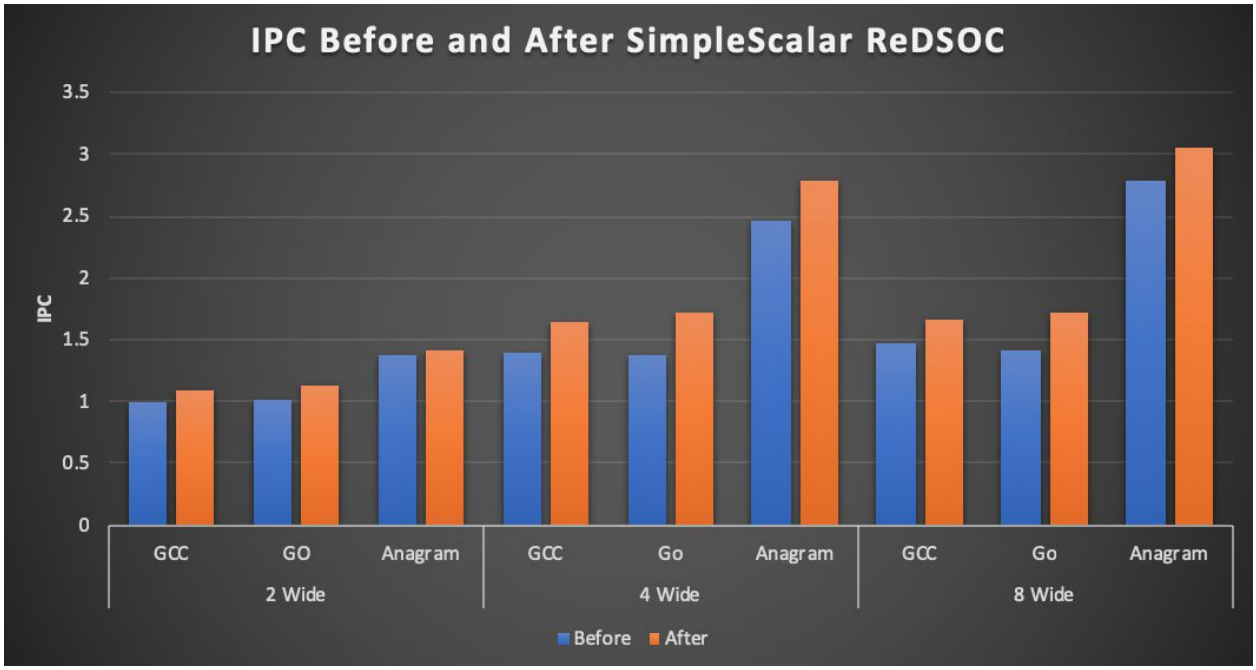

Figure 2: IPC before and after ReDSOC

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. Figure 2 shows the difference in IPC when implementing ReDSOC. 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. Figure 3 shows an experimental increase of 3.9% to 24.9% based on this difference in IPC. On average, we obtained percent speedup of 13.8%.

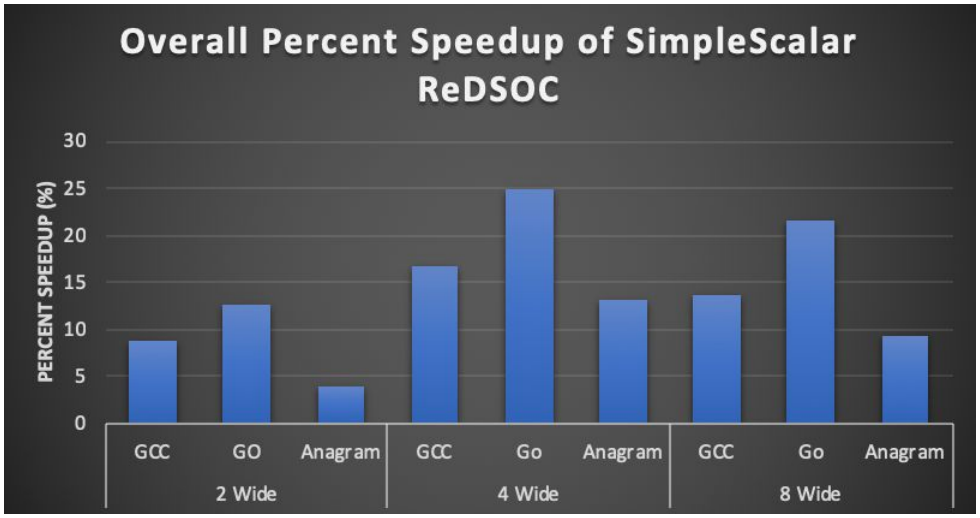

Figure 3: Overall percent speedup due to ReDSOC

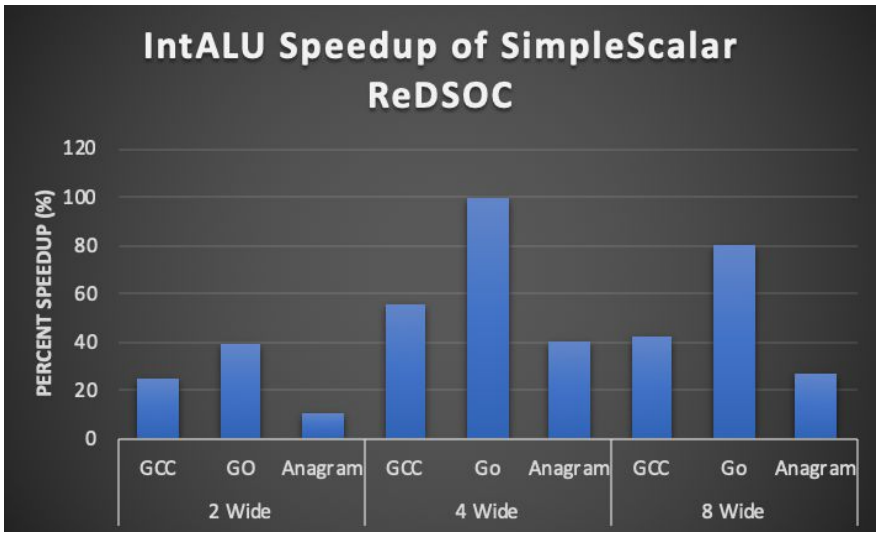

Figure 4: Integer ALU percent speedup due to ReDSOC

To derive the speedup for integer ALU instructions, we used the following variation of Amdahl's Law:

$$ALU\ Speedup = \frac{total\ \#\ of\ ALU\ instructions}{total\ \#\ of\ program\ instructions} / \left(\frac{1}{overall\ speedup} - \left(1 - \frac{total\ \#\ of\ ALU\ instructions}{total\ \#\ of\ program\ instructions}\right)\right)$$

Figure 4 shows the experimental range of the integer ALU instruction percent speedup to be 10.3% to 99.1% - corresponding to the same configurations as earlier. 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%.

Analysis

The paper from which we based our SimpleScalar implementation of the ReDSOC design noted that they calculated a range of 5% to 25% increase in overall performance across all the benchmarks they used. In analyzing the nine comparisons shown in the results sections, our results show the smallest increase in IPC came from the Anagram benchmark on the small, 2-wide processor which was 3.9%. The largest increase in IPC came from the GO benchmark on the medium, 4-wide processor which was 24.9%. Thus, 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. The experimental range of integer ALU instruction speedup was found to be 1.1 to 2. The average speedup across all benchmarks and processor configurations was 1.46. Due to this significant average speedup, Amdahl's law helps show that increasing integer ALU performance can correspond a significant overall performance increase (avg. 13.8%) 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 an increasing IPC when processor width increased.

We believe this is the case because the ReDSOC implementation requires at least one dependent instruction of a parent 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 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.

Despite the IPCs across all benchmarks for the large, 8-wide configuration being larger than IPCs for the medium, 4-wide configuration, it is interesting to note that the overall percent speedups across each benchmark between the medium and large configurations decreased. This means that in this experiment, the medium configuration was able to recycle more data slack than the large configuration. There are a couple of explanations on why this could be the case.

One could be due to the fact that the large, 8-wide configuration only contained 6 integer ALU functional units instead of 8. This is different from both the small and medium configurations since they each had the number of integer ALU functional units equal to their

processor-width sizes. This can create an impact because a smaller number of integer ALU functional units means that more data slack opportunities are unable to take place since functional units will be busy more often.

Another possibility could be that the additional costs and complexities of having a large, 8-wide configuration out-of-order processor may have combated the overall increase in the speedup from recycling data slack. For example, bypass networks grow N^2 when processors are of width N. Since the medium, 4-wide configuration is smaller and less complex relative to the large one, it would have a smaller amount of latency from complexities and be able to show higher effect from recycling data slack.

Related work

The primary related work [1] helped guide us in understanding the problem with data slack, and provided the ReDSOC solution we used as a basis for attempting to aggressively recycle this data slack within SimpleScalar. This related work provides results from a much wider set of benchmarks that include programs that contain a significant percentage of high dependence memory operations and others that contain between 30-60% high slack ALU operations. It even includes some benchmarks that involve machine learning kernels and offers ideas for increasing the amount of slack opportunities within them.

Fast ALU computations are being utilized in some Intel processors [2]. Their low latency integer ALU computation allows them to compute ALU operations in half a clock cycle. The output of these operations then use a closed loop called the ALU-bypass loop to allow dependent ALU operations to compute the other half of the clock cycle. This work relates well to how ReDSOC loops or recycles the output of an ALU functional unit to another functional unit.

Prior works have also optimized narrow data-width based execution to improve functional unit utilization [3]. This work recognizes that over half of integer ALU operations in benchmarks require 16 bits or less in computation instead of the 64 bits used to describe the entire operation. Their work obtained a speedup by allowing the hardware to recognize this and combine the data of multiple ALU instructions performing the same operation into a single instruction that would go through the ALU. This work relates to this project by identifying another cause of data slack and devising a solution in order to reduce the amount of slack.

Another related work found a way to optimize narrow data-width based execution to improve issue-width [4]. This work develops a microarchitecture called Multi-Bit-Width in order to exploit the predictability of data-widths. It then injects multiple narrow-width instructions into wires usually used to transport operands and bypass 64-bit instruction results in order to effectively increase the issue-width without adding additional wires. This work relates to this project by predicting data slack opportunities and increasing the issue-width by adding multiple narrow-width instructions instead of a single 64-bit instruction.

One more related work uses dynamic scheduling to allow instructions classified as narrow-width to execute faster [5]. It does this by placing narrow-width instructions on a fast, narrow datapath that includes fast, narrow ALUs and caches. This work relates to this project by using the scheduler in order to aid in the recognition of slack opportunities and operating accordingly.

Conclusion

This paper accurately recreated the results its predecessor. By locating and recycling the data slack, a nontrivial amount of speedup was obtained for ALU operations that take up a significant amount of instructions in programs. This showed how prevalent data slack is in many applications, and how reducing it can dramatically increase performance and bandwidth.

A modification of SimpleScalar allowed it to follow ReDSOC's method of reducing data slack. By finding the relationships between instructions, dependent instructions can be completed exactly when their parent instructions complete. When this is done iteratively over an entire instruction sequence, a substantial amount of data slack can be recycled, leading to the aforementioned 3.9% to 24.9% overall speedup.

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. G. H. Loh, "Exploiting data-width locality to increase superscalar execution bandwidth," ser. MICRO, 2002.
5. E. Gunadi and M. H. Lipasti, "Narrow width dynamic scheduling," Journal of Instruction-Level Parallelism, 2007.