EEAL: Processors' Performance Enhancement Through Early Execution of Aliased Loads

Abhishek Rajgadia
CADSL, Department of Electrical Engineering,
IIT Bombay, India
abhiraj171@gmail.com

Newton
CADSL, Department of Electrical Engineering,
IIT Bombay, India
newton@ee.iitb.ac.in

Virendra Singh
CADSL, Department of Electrical Engineering,
IIT Bombay, India
viren@ee.iitb.ac.in

ABSTRACT

Due to increasing speed gap between DRAM and CPU, improving the performance of memory instructions has become very critical. Load and Store instructions account for almost 20-30%. It indicates that memories have become a bottleneck in CPU's performance. Conventionally load instructions, being more critical than stores, are to be executed as early as possible. Load-Store Queues (LSQs) are used for early execution of aliased loads by forwarding them results of the executed stores. Generally memory instructions, ready to be issued from Reservation Station (RS), have their source registers available however their memory addresses are not yet known. Hence, RS cannot detect aliasing loads/stores. We propose to insert a 2nd RS after the address computation stage in the memory pipeline. The 2nd RS helps in taking data forwarding decisions based on the computed addresses of the memory instructions. If the address of a load instruction aliases with preceding store, data can be directly forwarded to it. Such forwarded loads can bypass address translation and cache access stages, leading to their early execution. Our evaluation on a 4-wide Out-of-Order (OoO) core shows that the proposed architecture outperforms the baseline architecture by up to 7.3% (avg: 2.2%) with just 0.1% area overhead and 0.05% power overhead.

Keywords

Aliased Load-Store Address Pair; Reservation Station; LSQs

1. INTRODUCTION

Since the invention of microprocessor in 1971, there has been an exponential improvement in its performance. However, in the last decade as frequency scaling has saturated due to power constraints, the pace of performance improvement of processors has slowed down [1]. DRAM's speed has also not improved much to keep the pace with CPU. Bridging the performance gap between the CPU and DRAM has become immensely important to enhance the performance of single-threaded applications.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

GLSVLSI '17, May 10-12, 2017, Banff, AB, Canada

© 2017 ACM. ISBN 978-1-4503-4972-7/17/05...$15.00.

DOI: http://dx.doi.org/10.1145/3060403.3060445

A single-threaded application's performance can be improved executing independent instructions OoO. However, due to *true* data and memory dependencies present in the application, most of the times CPU remains idle waiting for these dependencies to get resolved. There are two types of true data dependencies that occur in a program. First is *register dependency* which occurs when an instruction writes to a register and subsequent instruction reads from the same register. Such kind of dependencies are detected at decode stage by examining the register operands of the instructions. Second is *memory dependency* which occurs when a store instruction writes to a memory location and later a load instruction reads from the same location. Such dependencies cannot be resolved at decode stage because access-addresses of memory instructions are not known while decoding. Instruction decoding of memory instruction only provides information about the registers and immediate offsets which are used at a later stage of pipeline for address computation. The addresses are calculated in memory stage of the pipeline only after instructions are issued, followed by address translation and memory access. Therefore, the instruction scheduler has no information if a load instructions can be scheduled before a preceding store instruction. If the load instruction is scheduled prior to an aliasing preceding store, it may read wrong data. Such memory-order violation leads to performance penalty. Compilers, with the capability of statically analyzing the entire program, can at times provide help in detecting aliasing loads/stores. However, even compiler cannot provide much help when the data structures used are pointers.

To avoid memory-order violations, instruction scheduler can apply a conservative approach where load and store instructions are executed inorder, however it will be a very pessimistic approach. The *load* instructions will be forced to wait till all the preceding stores are committed even if they are not dependent on the *stores.* For maximum performance, an OoO processor must detect such false dependencies and issue *load* instructions at the earliest [2]. However, at the same time it should also avoid memory-order violations with any preceding aliasing stores.

Conventionally, memory dependence predictions [3] are used which give information to the scheduler about the load instructions that are safe to schedule. Memory dependence prediction is a technique which allows load instructions to execute OoO. A memory dependence predictor uses past behaviour of load instructions to predict if a particular load instruction will alias with any prior stores. If not, the load is scheduled at the earliest. In case of incorrect prediction,

memory-order violation will occur, which is detected at a later stage of the pipeline and the state of processor is recovered with a penalty of few cycles. However, memory predictors rely on past behaviors, which may change.

Our paper offers the following contributions:

1. We identify the scope of improvement in the performance of aliased loads by their early execution.

2. We present a simple yet effective micro-architecture for memory instructions to forward data from *in-flight* stores to the corresponding aliased loads.

3. Aliased loads which suffer maximum memory-order violations are identified and are handled using dynamic forwarding which reduces such violations upto 90%.

Rest of the paper is organized as follows. Section II presents the motivation behind our work. Section III elaborates on the micro-architectural changes implemented and other relevant concepts. Section IV explains the experimental framework adopted. Section V reports the experimental results. Section VI presents related work and section VII is devoted to the concluding remarks.

2. MOTIVATION

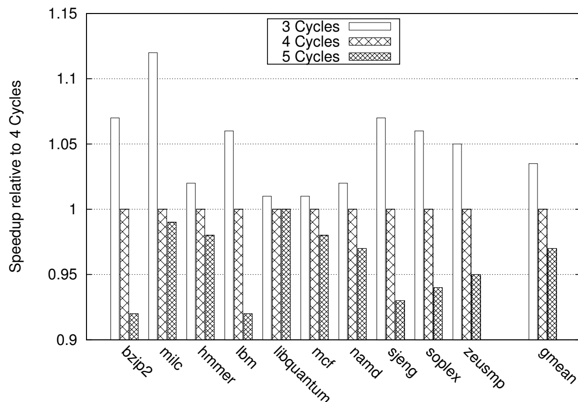

Figure 1: Effect of execution latency on performance

In load-store architecture, load and store instructions comprise of about 20-30% [4] of total instructions. Therefore, efficient execution of memory instructions is very important to improve single thread performance. Instructions do wait in RS to resolve true data dependencies and to avoid structural hazards. Memory instructions go through three stages 1) *Address Computation* 2) *Address Translation* and 3) *Memory Access*. In an OoO core only aliased memory accesses are required to be done in program order to avoid violations. Instructions should be scheduled in such a way that load instructions are executed at the earliest. One of the impediments is the load/store aliasing. A load instruction cannot be scheduled before aliased store is executed. Therefore, in order to execute load early, it is important to calculate addresses of all the earlier stores as soon as possible to check for alias.

Consider two memory instructions I1 and I2 shown below
I1: `store R3,[R2],#50`
I2: `load R1,[R2],#50`
OoO core should execute I2 at the earliest so that the dependency chain due to the load instruction I2 can get resolved early. I2 can only get executed if its operand R2 is ready and it does not alias with I1. However, I2 is aliasing with I1 as the memory location accessed by them is same. This aliasing can be detected only after *Address Computation* stage. Instruction I1 will be issued to address generation units (AGU) if both R3 and R1 are ready. If we issue both I1 and I2 when their source operand R2 is ready (only R2 is required for address computation in AGU), we will be able to detect aliasing earlier than conventional OoO processor.

As the *ready* memory instructions are waiting in RS, they can be scheduled such that their addresses can be calculated as early as possible. These calculated addresses can be used to detect aliasing load/store pairs. These computed addresses can be also used for better memory dependence predictions.

Memory predictors helps in early execution of load instructions only when it finds them not aliasing with previous pending stores. If it finds any pending aliased store, it executes stores first, holding the corresponding loads in waiting state. Such aliased loads can be executed earlier by forwarding data from the corresponding pending stores, if the addresses of loads and stores are computed earlier (to detect aliases). Loads with forwarded data do not need to access caches and Translation Lookaside Buffer (TLB) thus reducing number of cache and TLB accesses. This will also reduce stalls due to TLB misses and cache misses which are significant. However, in the conventional pipeline loads/stores have to wait in the primary RS before proceeding further to *Address Computation* stage until either their operands and data are ready or the memory predictor predicts that they will not alias. In the conventional pipeline loads can forward data from executed stores in LSQ, however stores which are in-flight are not utilized for such forwarding. For in-flight stores, neither memory predictor nor LSQ helps in early execution of corresponding aliased loads.

Memory instructions which are waiting in primary RS can be scheduled directly for *Address Computation* stage as calculated addresses can be efficiently used for finding aliases. Memory instructions with their computed address can wait in another RS (2nd RS) before their dependencies can be resolved. However, creation of 2nd RS will add an additional delay in the execution of memory instructions.

An additional cycle delay in execution of memory instruction can have serious impact on performance of processors. Our analysis on the effect of increasing or decreasing the execution latency of memory instructions by 1-cycle is done by studying 10 SPEC CPU 2006 benchmarks. The system configurations used for the study are mentioned in detail in section IV. Figure 1 shows that by decreasing the latency of execution of each memory instruction by 1-cycle, there can be improvement of up to 11.53% (average 3.45%) while increasing the execution latency by 1-cycle can cause performance loss of up to 7.57%. Thus, if we are able to decrease the execution latency of certain fraction of memory instructions, we can get significant performance gain. Improvements due to the proposed micro-architecture depends on the number of aliasing cases present in the RS. Initial study was done to find the number of such cases. Table 1 gives percentage of total instructions executed that could have bypassed the execution. As evident from our study, there is a potential for early execution of aliasing loads/stores whose dependencies can be predicted earlier in the 2nd RS to gain

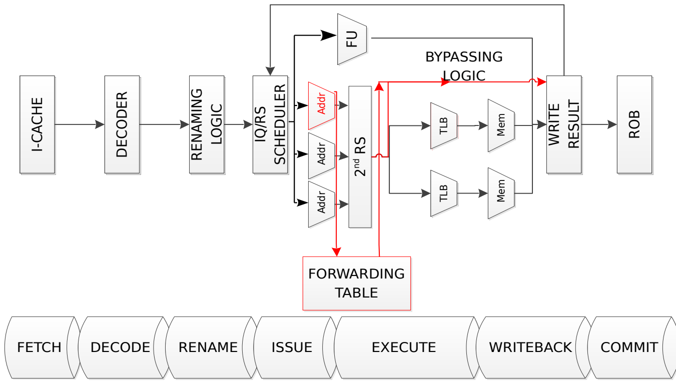

Figure 2: Proposed architecture with 2nd RS, Forwarding table and Bypass logic

performance benefits. Hence, the focus of our work is on Early Execution of Aliased Loads (EEAL).

Table 1: Potential cases for early forwarding

Benchmark	Aliasing Load/Store (%)
bzip2	2.2
mcf	1.33
lbm	0.79
milc	0.53
cactus	0.2
soplex	0.05

3. PROPOSED ARCHITECTURE

Our proposed architecture is shown in Figure 2. In order to facilitate for early address computation, we insert second RS where instructions can wait for memory access and data availability for store. There instructions from primary RS (called as RS-1) can be issued to 2nd RS (called as RS-2) as soon as operands needed for their address computation are available. After their addresses gets computed they are inserted into the RS-2. Now, these instructions present in RS-2 have their effective address known. These addresses can be used for detecting aliased loads and forwarding of data from pending store to corresponding aliased load. The advantages of RS-2 are following:

1. Since addresses are known, it can be used for forwarding data between aliased load/store pairs.
2. Due to data forwarding from stores to loads in RS-2, forwarded loads do not need to access caches and TLB thereby reducing TLB misses and cache misses.

Early Forwarding and Bypassing: In a normal OoO processor, data is forwarded from store to load in parallel to the cache access. This happens after address generation and translation. Since we have the availability of addresses of stores and loads in RS-2, we can forward data in RS-2 itself and load instruction can bypass the remaining stages of execution. The advantages of early forwarding and bypassing are as follows:

1. Loads which get forwarded data from preceding stores (called as **early-forwarded loads** from here onward) in RS-2 can bypass address translation and cache access pipeline stages. Elimination of two stages can have huge impact on performance as evident from Figure 1. It also saves power.
2. The execution units that are not used by early-forwarded loads can be used by other memory instructions. It gives an illusion of more number of execution units.
3. Memory predictor prevents issuing of loads if an aliasing preceding store is present in the RS-2 to avoid memory order violation. By early forwarding, loads do not have to suffer delay due to memory predictor.
4. Instructions dependent on early-forwarded-load can become ready early, it may have domino effect.

Forwarding Logic: Forwarding logic consists of a Forwarding table as shown in Figure 2. It has a *Valid bit*, *ROB Index*, *Data*, and *Address* per entry as shown below.

Valid	ROB Index	Address	Data

The store instructions which are issued from RS-1 are inserted into RS-2 after address computation. At the same time, the store instruction also makes an entry in the Forwarding table, indexed using LSBs of the effective address. The load instructions which are present in the RS-2, indexes into the Forwarding table using LSBs of their effective address. If the load finds a valid entry in the table at the index corresponding to it and the store is older than the load (which can be checked using ROB index), the load takes data from the table entry. When the store is issued from the RS-2, it invalidates the entry in the Forwarding table.

Dynamic Forwarding: All early-forwarded loads do not get their data from the corresponding correct stores. Correct store is the one which is the latest among the preceding aliasing stores. In case of repeated violations, it has been observed that the same static load is forwarded data from the wrong store again and again. Such cases can be identified dynamically. We can dynamically learn which loads have crossed the given threshold of wrong forwarding. This information can be communicated to the forwarding logic which will prevent early forwarding to such loads.

Bypassing Logic: The early-forwarded loads have obtained data from the Forwarding table. Hence, they do not need to perform address translation and cache/Store Queue access. Bypassing logic will send these loads directly to write-back stage, skipping the address translation and cache/store queue access. Figure 2 shows the complete architecture of the processor with RS-2, additional AGU, Forwarding table and bypassing logic. Structures bordered in red are the additional hardware added to the baseline architecture

Instruction Scheduler: The RS-2 will add additional delay for the memory instructions. As shown earlier in Figure 1, additional delay can have adverse effect on the performance. To mitigate the effect of additional stage inserted in the pipeline, instruction scheduler is changed so that all instructions do not suffer the delay. If RS-2 is empty, i.e., if there is no instruction to be scheduled for translation, the

instruction coming from RS-1 after address computation is directly scheduled for translation. Thus, not all memory instructions suffer delay due to RS-2. Only those instructions which get accumulated in RS-2 due to extra AGUs get delayed which were anyways going to be scheduled later from RS-1 if there were no additional AGUs. However, such additional delays are compensated as early-forwarded loads can skip *address translation* and *cache access* stages of pipeline. In the instruction scheduler only *two* changes have been made over the conventional *Tomasulo Scheduler*

1. Directly scheduling store instruction from RS-1 to AGU when registers required for its address computation are ready.

2. If RS-2 is empty, memory instructions are directly scheduled for address translation.

In summary, the sequence followed by the memory instructions in our proposed architecture is mentioned below:

1. Load and Store instructions wait in RS-1 for their source registers to be ready. Ready loads and stores are issued from RS-1 to AGUs.

2. After their address computation in AGUs, memory instructions wait in RS-2 for further execution.

3. Store instructions make entry in the Forwarding table by indexing into it using LSBs of the address. If an entry is already present at that index, it is overwritten.

4. If there are no memory instructions waiting in RS-2, they are directly scheduled to address translation stage of the pipeline.

5. Loads waiting in the RS-2 also index into Forwarding table using LSBs of their address. If a store in the Forwarding table is older than the load and if address aliases, then the load takes data from the table.

6. If memory-order violations for a particular load increases to certain fixed threshold, early-forwarding to it is dynamically stopped.

7. All early-forwarded loads bypass address translation and cache access stages.

8. When a store gets scheduled from RS-2 to address translation stage, its corresponding entry in the Forwarding table is invalidated.

9. If the latest preceding aliasing store has already committed, the early-forwarded load can take data from wrong preceding store. The load instruction checks Store Queue for this condition and flags such violation, if any.

4. EXPERIMENTAL METHODOLOGY

4.1 Simulation Infrastructure

To simulate the proposed architecture, we used cycle accurate *gem5* [5] simulator. Power and area modeling were done using McPAT [6]. Baseline architecture, as used in [7], is a 4-wide OoO superscalar processor implementing ARM 64-bit ISA working at 2GHz. It is a 7-stage processor consisting of Fetch, Decode, Rename, Issue, Execute, Writeback and Commit stages. Detailed configuration is shown in Table 2. The baseline architecture is an aggressive architecture with 4 load/store execution units while early-forward architecture has only 2 load/store execution units. Also issue width is 6 in baseline while 4 in early-forward architecture. The memory instructions are allowed to be issued OoO based on Store Sets memory predictor [3].

Table 2: System Configuration

Parameter	Baseline	Early-fwd
Fetch width	4	
Issue width	6	4
Commit width	4	
RS	128	
Reorder Buffer	192	
Load Queue/Store Queue	32/32	
L1 inst cache	32KB, 2-way, 2-cycles	
L1 data cache	64KB, 2-way, 2-cycles	
L2 unified cache	2MB, 8-way, 20-cycles	
Branch predictor	Tournament, 4096 BTB,16 RAS	
Physical Register File	256 INT, 256 FP	
Execution units	4 INT, 2 FP units	
Memory units	4	2
Forwarding per cycle	NA	4
Extra AGU	NA	2
LFST/SSID	1024/1024	

The RS entries are freed upon issue of instructions, except memory instructions which hold the entries until completion. In early-forward architecture, entries of RS-2 are held by memory instructions until completion.

4.2 Benchmarks

For evaluation, 18 benchmarks (8 Integer and 10 Floating Point) from SPEC CPU2006 [8] benchmark suite were used. Each benchmark was fast-forwarded for 1 billion instructions and 100 million instructions were executed in detailed mode using reference inputs. All workloads were compiled for ARM 64-bit ISA using gcc 4.8.4 with full optimization (-O3).

5. RESULTS AND ANALYSIS

Figure 3 shows performance improvement compared to the aggressive baseline architecture. The graph shows normalized instructions per cycle (IPC) or speedup. The maximum performance improvement is 6.7% for *milc* and average improvement of all the benchmarks is 1.6%. There is performance enhancement as a result of early-forwarded loads which bypass TLB and cache access stages. Replay loads which now need 2-cycles to execute compared to baseline where it needs 3-cycles to execute also contribute to gain in IPC. Replay loads are those loads which are mis-speculated and needs re-execution. Such loads have their address already computed.

The early-forwarding case is also compared with a baseline having 4-wide issue with 4 load-store units (which is more closer to the proposed design) and a baseline having 4-wide issue and only 2 load-store units. Compared to the less aggressive baseline architecture, performance improvement is up to 7.32% maximum for *milc* benchmark (2.2% on average) as shown in Figure 3.

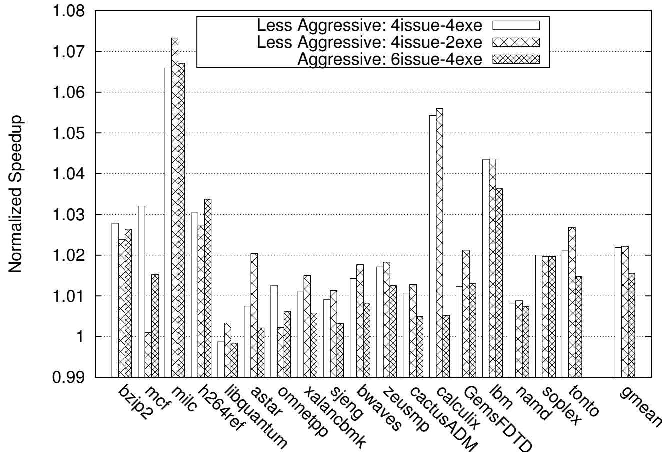

Figure 3: Performance enhancement using 2-level RS

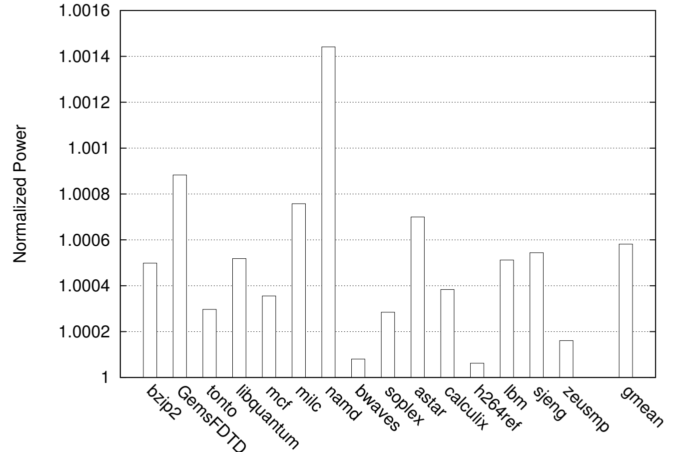

Figure 4: Power overhead with 2-level RS

5.1 Simulation data

Table 3 gives the percentage of early-forwarded loads and percentage of early-forwarded loads that got violated. On average, 2.23% of total issued loads gets early-forwarded and can bypass TLB and cache access. Maximum loads forwarded is 20.31% of total issued loads for *soplex* benchmark.

Table 3: Simulation Data

Benchmarks	% Fwd	% Fwd Violated
astar	0.00	0.00
bwaves	0.31	0.00
bzip2	4.51	0.00
cactusADM	0.00	0.00
calculix	0.11	0.00
GemsFDTD	2.38	0.00
h264ref	0.70	3.49
lbm	3.66	0.00
libquantum	1.66	0.00
mcf	0.01	1.13
milc	0.99	0.00
namd	0.59	0.14
omnetpp	0.32	1.53
sjeng	0.28	8.68
soplex	20.31	0.00
tonto	0.38	0.00
xalancbmk	3.87	0.01
zeusmp	0.01	0.00

5.2 Dynamic forwarding

As shown in Table 3, in case of *h264ref* and *sjeng*, the number of forwarded loads that got violated are 3.49% and 8.68% respectively. Analysis showed that 4 static loads in case of *h264ref* and 9 static loads in case of *sjeng* caused 90% of the violations. Dynamically such loads were identified and forwarding for them was stopped when they crossed certain threshold number of violations, 100 in this analysis. By doing so, it reduced the violations by about 90% in each case.

Figure 5 shows the IPC improvement and Figure 6 reports decrement in violations by using dynamic forwarding for *h264ref* and *sjeng* benchmarks. Performance improved by 0.5% and 0.3% by using dynamic forwarding for *h264ref* and *sjeng* benchmarks respectively.

5.3 Area and Power analysis

The extra hardware added are two AGUs, Forwarding table with 32 entries and bypassing logic. As number of stores present in RS at any time is small compared to total number instructions, Forwarding table with 32 entries is sufficient. Combined size of RS-1 and RS-2 in our proposed architecture is kept equal to the size of RS of baseline architecture for fair comparison. Small extra overhead in RS-1 is due to addition of extra ports. The area and power overhead analysis was done using McPAT [6] for 22nm technology running at 2GHz. Total area overhead is 0.1% and power overhead is 0.05% as shown is Figure 4.

6. RELATED WORK

Memory Dependence Prediction (MDP) was first proposed by Moshovos [9]. Conventionally an OoO core has a MDP to predicts if a particular load/store pair will alias. Although our approach is different than conventional memory predictors, still the focus of our work is in similar lines to that of memory dependence predictors. In [3], MDP has been proposed to issue load as early as possible, while avoiding memory-order violations. Instruction scheduler uses information about aliased loads/stores while scheduling the load instructions at the earliest. The level of aggressiveness of predictor was increased or decreased as per the characteristic of aliasing load/store pairs. Different colors have been used to represent different speculation levels as per the level of memory-order violations.

Memory-order violations can also be reduced by analyzing number of stores between a load instruction and the previous store instruction that access the same memory location, referred as Store Distance in [10]. They have used compiler to annotate instructions and pass store distance information to the micro-architecture. Their design uses store distance information to identify store upon which load depends and make safe speculations. However their technique

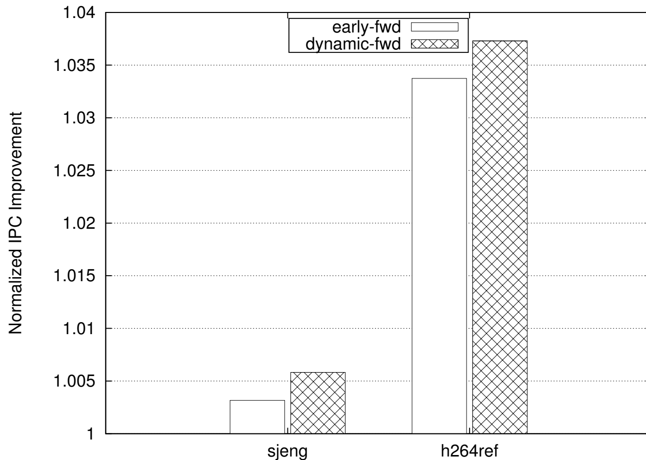

Figure 5: Performance enhancement with Dynamic Forwarding

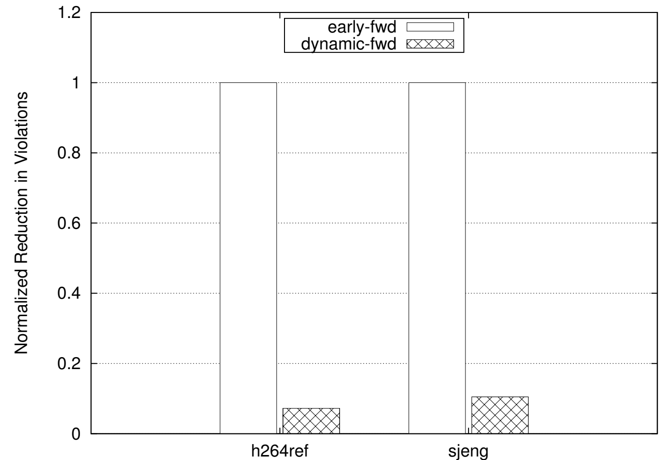

Figure 6: Violation decrement with Dynamic Forwarding

requires ISA modification as stores distances have to be encoded in load instruction. The complex design of load-store queue (LSQ) has been shown to be a bottleneck to processor's performance in [11]. A simple segmented design has been proposed in [11] to redesign the LSQ with a focus to reduce the search bandwidth demand. In [12] a storage-free MDP has been proposed. Sophisticated branch predictor has been used to predict memory dependencies, thus no separate mapped-tables have been used as in [3] to find aliased load/store pairs. However, their technique under performs when compared to [3] for some SPEC benchmarks. Our proposed technique can be used along with all the above mentioned improvisations as our technique aids MDPs by finding aliased load/store pairs using their access addresses.

7. CONCLUSION

In this work, we proposed a novel idea of using 2-level Reservation Station (RS) targeted towards improving single thread performance by early execution of aliased loads. By using 2-level RS, data can be forwarded early from store to the aliased load. We successfully decreased execution latency by early forwarding of loads and also decreased the latency for replay loads. Our analysis on SPEC CPU2006 benchmarks shows that our proposed architecture improves the performance up to 7.32% (2.2% on average) compared to less aggressive similar size baseline architecture. The power overhead is 0.05% and area overhead is 0.1%. Compared to an aggressive baseline architecture, performance improvement is up to 6.7% for *milc* benchmark (1.6% on average).

8. REFERENCES

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[2] S. Onder and R. Gupta, "Dynamic memory disambiguation in the presence of out-of-order store issuing," in *Proceedings of 32nd Annual International Symposium on Microarchitecture*. IEEE, 1999, pp. 170–176.

[3] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3. IEEE Computer Society, 1998, pp. 142–153.

[4] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[6] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2009, pp. 469–480.

[7] A. Perais and A. Seznec, "EOLE: Paving the way for an effective implementation of value prediction," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 481–492.

[8] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[9] A. I. Moshovos, "Memory dependence prediction," PhD dissertation, University of Wisconsin-Madison, 1998.

[10] C. Fang, S. Carr, S. Önder, and Z. Wang, "Feedback-directed memory disambiguation through store distance analysis," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 278–287.

[11] I. Park, C. L. Ooi, and T. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 411.

[12] A. Perais and A. Seznec, "Storage-free memory dependency prediction," *IEEE Computer Architecture Letters*, 2016.