The Effect of Load/Store Queue Size on System Performance

Daniela Curcievska[1], Pece Mitrevski[1] and Marjan Gusev[2]

Abstract. **As processors continue to exploit more instruction level parallelism, greater demands are placed on the performance of the memory system. Load latency is identified as a major bottleneck in modern superscalar processors. In this study we propose load-forwarding via LSQ in issue and execution phases of the pipeline, in order to get shorter execution time. We come to conclusion how changing the LSQ size affects system performances. Namely, on average, a 6-9% decrease in the number of execution cycles with the increase of LSQ size up to 32 entries, which is not the case afterwards. With additional increase in size, gain speedup becomes insignificant, while for some of the test-examples the number of execution cycles slightly increases.**

Keywords. **ILP, memory, memory instructions, load forwarding, pipeline, RUU, LSQ**

I. Introduction

The objective of modern superscalar processors is to maximize the instruction level parallelism (ILP) that can be extracted from programs. The most basic method uses for extracting more ILP from program is out-of-order execution. Unfortunately, out-of-order execution by it self does not provide the desired level of ILP. The program's control flow [1] and data flow [2] impose serious limits on level of parallelism that can be extracted. Therefore, most modern processors employ aggressive branch prediction mechanisms to relax the control flow constraints that limit the ILP. To overcome the data-flow limits, researchers have suggested data speculation [2, 3, 4, 5, 6] to be used.

II. Memory Dependence Problem

Modern processors exploit ILP by executing instructions in an order that is defferent from the sequential program order, which is called out-of-order execution. In other words, independent instructions whose operanda are ready can be scheduled and executed before older instruction that are still waiting for their operands.

Hence, to support out-of-order execution, the hardware needs to be able to precisely determine the dependence among instructions so that sequential program semantics will not be violated. In case of register dependences, determining which intructions are dependent is easy due to the explicit encoding of architecture register names in the instruction format. Memory dependences are much harder to determine, because memory addresses are not explicitly encodedin the instruction format and need to be dynamically generated. However, this dynamic generation of memory addresses is not done in sequential program order. Hence, when a load instruction is ready to be scheduled, it is likely that there are older store instructions in an instruction window whose addresses have not yet been determined. This problem is know as the memory anti-aliasing [7]. A related concept, the process of determining if two memory instuctions access the same memory location is called *memory disambiguation*.

There are several ways to attack the memory anti-aliasing problem. One possible solution is to execute all store and load in total program order. Considering that load and store instructions comprise a large fraction of instructions in most programs, imposing a total order on memory reference instructions would seriously limit the ILP that can be extracted from programs. A slightly less conservative approach is to delay the scheduling of load until all previous store addresses become available. This approach limits the amount of ILP extracted from memory disambiguation, because it is unlikely that a load will conflict with many of the previous stores.

The aggressiveness of memory disambiguation using speculative scheduling also depends on how much more parallism we want to exploit. A less aggressive approach is to predict whether a load will conflict with any older store in the instruction window. In this case, the load can not be scheduled until all older stores execute. A more aggressive scheme is to predict that a load conflicts with a particular earlier store, if any, and delay the scheduling of the load until that particular store executed. However, a load instruction will not be unnecessarily delayed when the predict is corect.

One extreme form of speculative memory operation scheduling is to always assume that the load that is to be scheduled will not conflict with any of the unknown store addresses. This kind of extreme speculation is not the best performing technique due to the cost of recovery as a result of misprediction.

It is real to assume that existing of Load and Store Queue (LSQ) that enables, when a store instruction executes, it writes its data into LSQ, and a later load that is dependent on the store to access and read data from LSQ will increase the amount of ILP. This is called *load forwarding* [8]. In order to

[1]Daniela Curcievska and Pece Mitrevski are with the Faculty of Technical Sciences, University "Sv. Kliment Ohridski", I. L. Ribar bb, 7000 Bitola, Macedonia, E-mail: daniela.curcievska@uklo.edu.mk, pece.mitrevski@uklo.edu.mk

[2]Marjan Gusev is with the Institute of Informatics, Faculty of Science, "Ss. Cyril and Methodius" University, Arhimedova 5, P.O Box 162, 1000 Skopje, Macedonia, Email: marjan@on.net.mk

be able to do load forwarding, a load needs to have its address calculated. In continuation, the Baseline Superscalar Processor Model which supports usage of LSQ data structure will be presented.

III. Baseline Superscalar Processor Model

In order to describe the usage and the management of the LSQ data structure, an overview of Superscalar Processor Model will be given. Baseline Superscalar Processor Model with its five phases of pipelining is given in Figure 1.

Register Update Unit (RUU) is a hardware data structure that is used to resolve data dependencies by keeping track of an instruction's data and execution needs and that commits completed instructions in program order.

Each register in the general purpose RegFile has two associated n-bit counters (n of 3 is typical)

1. NI (number of instances) – the number of instances of a register as a destination register in the RUU
2. LI (latest instance) – the number of the latest instance

When an instruction with destination register address Ri is dispatched to the RUU, both its NI and LI are incremented. Dispatch is blocked if a destination register's NI is 2n -1, so only up to 2n – 1 instances of a register can be present in the RUU at the same time. When an instruction is committed (updates the Ri value) the associated NI is decremented. And finally when NI = 0 the register is "free" (there are no instruction in the RUU that are going to write to that register) and LI is cleared.

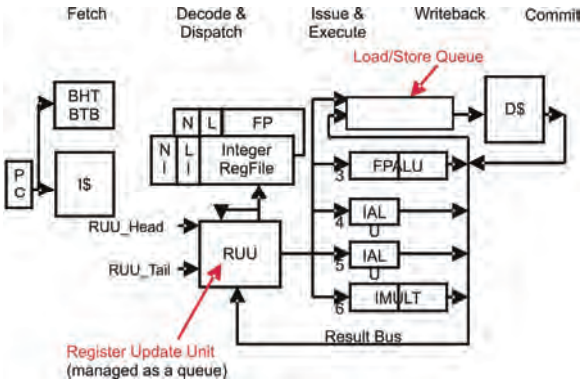

Fig. 1. Baseline Superscalar Processor Model

This processor model support pipeline instruction flow in five phases:

- Fetch (in program order): Fetch multiple instructions in parallel from the I$
- Decode & Dispatch (in program order):
  In parallel, decode the instr's just fetched and schedule them for execution by dispatching them to the RUU; Loads and stores are dispatched as two (micro)instructions – one to compute the effective addr and one to do the memory operation
- Issue & Execute (out of program order): As soon as the RUU has the instr's source data and the FU is free, the instr's are issued to the FU for execution
- Writeback (out of program order): When done the FU puts its results on the Result Bus which allows the RUU and the LSQ to be updated – the instr completes
- Commit (in program order): When appropriate, commit the instr's result data to the state locations (i.e., update D$ and RegFile)

By managing the RUU as a queue, and committing instruction from RUU_Head, instructions are committed (aka retired) in the order they were received from the Decode & Dispatch logic (in program order). Committing the instr's result data to the state locations (RegFile and D$) are buffered in the RUU and LSQ until commit time.

The LI counter allows multiple instances of a specific destination register to exist in the RUU at the *same* time via register renaming, and with that it prevents output dependencies.

The RUU perform the following four tasks in parallel every cycle:

1. Accepts new instructions from the Decode & Dispatch logic
2. Monitors the Result Bus to resolve true dependencies and to do write back of result data to the RUU
3. Determines which instructions are ready for execution, reserves the Result Bus, and issues the instruction to the appropriate FU
4. Determines if an instruction can commit (i.e., change the machine state) and commits the instruction if appropriate

Loads and stores are dispatched to the RUU as two (micro)instructions – one to compute the effective addr and one to do the memory operation:

lw R1,2(R2)	becomes	addi	R0,R2,2
		lw	R1,R0
sw R1,6(R2)	becomes	addi	R0,R2,6
		sw	R1,R0

At the same time with dispatching memory instruction to RUU, LSQ entry for that instr is allocated. Each LSQ entry consists of a Tag field (RegFile addr || LI) and a Content field. The LI counter allows for multiple instances of stores (writes) to a memory address.

When a load completes (the D$ returns the data on the Result Bus) or a store commits (in program order) the LSQ entry is released.

Instruction dispatch is blocked if there is not a free LSQ entry and two free RUU entries.

For load instruction, when the address is computed then the load instruction is ready to issue. The address is copied to the LSQ entry with the (associatively) matching RegFile||LI address and is sent to the D$ for execution. When the load value is returned it is copied into the destination content field of some instruction on the RUU. Advance the RUU_Head and LSQ_Head appropriately.

For store instruction, when the address is computed then the store instruction is read to issue. The computed address is copied to the LSQ entry with the (associatively) matching

RegFile||LI address along with the store data value. When the store instruction reaches the RUU_head, its companion LSQ entry (which by now is at the head of the LSQ) is sent to the D$ (to complete the store operation) and both the RUU_Head pointer and the LSQ_Head pointer are advanced.

Performance can be improved if loads are allowed to bypass previous stores, this is called *load bypassing*. That can be done only if the memory addresses of *all* previous stores dispatched to the RUU are known since must first check for load data dependency on previous uncommitted store instructions. That is facilitated by having a common load and store queue.

Store instructions are not allowed to bypass previous loads, so there are no antidependencies between loads and stores. Also store instructions are committed to the D$ in program order, so there can not be output dependencies between stores.

When a load's address becomes known, the address is compared (associatively) to see if it matches an entry already in the LSQ (i.e., if there is a pending operation to the same memory address).

- If the match in the LSQ is for a load, the current load does not need to be issued (or executed) since the matching pending load will load the data
- If the match in the LSQ is for a store, the current load does not need to be issued (or executed) since the matching pending store can directly supply the destination Content for the current load
- If there is no match, the load is issued to the LSQ and executed when the D$ is next available

When the RUU# of the load instruction appears on the Result Bus along with the memory data, the load completes by updating the RUU and releasing the RUU and LSQ entries for committed load.

When a store's address (and the store data) becomes known, the address is compared (associatively) to see if it matches an entry already in the LSQ (i.e., if there is a pending operation to the same memory address)

- If the match in the LSQ is for a load, the current store is issued to the LSQ
- If the match in the LSQ is for a store, the current store is issued to the LSQ with an incremented LI
- If there is no match, the store is dispatched to the LSQ

Store instructions are held in the LSQ until the store is ready to commit (i.e., until its partner instruction reaches the RUU_Head) at which time the store is executed (i.e., the data and address are sent to the D$) and the RUU and LSQ entries are released.

IV. Out-Of-Order Processor Timing Simulation

In order to show how LSQ size affects system performance, simulations of execution on different test-examples with sim-outorder simulator were made, which is part of The Simple Scalar Tool Set [11].

This simulator support out-of-order issue and execution, based on the RUU. The RUU scheme uses a reorder buffer [10] to automatically rename registers and hold the results of pending instructions. Each cycle the reorder buffer retires completed instructions in program order to the architected register file.

The processor's memory system employs a load/store queue. Store values are placed in the queue if the store is speculative. Loads are dispatched to the memory system when the memory addresses of all previous stores are known. Loads may be satisfied either by the memory system or by an earlier store value residing in the queue, if their addresses match.

The main loop of the simulator, located in sim_main(), is structured as follows:

```
ruu_init();
for ( ; ; ) {
        ruu_commit();
        ruu_writeback();
        lsq_refresh();
        ruu_issue();
        ruu_dispatch();
        ruu_fetch();
}
```

This loop is executed once for each target (simulated) machine cycle. By walking the pipeline in reverse, inter-stage latch synchronization can be handled correctly with only one pass through each stage. When the target program terminates, the simulator has generated the statistics.

The fetch stage of the pipeline is implemented in ruu_fetch(). The fetch unit models the machine instruction bandwidth, and takes the following inputs: program counter, the predictor state and mispericition detection from the branch execution unit(s). Each cycle, it fetches instructions from only one I-cache line. After fetching the instructions, it places them in the dispatch queue, and probes the line predictor to obtain the correct cache line to access in the next cycle.

The code for dispatch stage of pipeline resides in ruu_dispatch(). The routine is where instruction decoding and register renaming is performed. The function uses the instructions in the input queue filled by the fetch stage, a pointer to the active RUU and the rename table. Once per cycle, the dispatch takes as many instructions as possible from the fetch queue and places them in the scheduler queue. The dispatch routine enters and links instructions into the RUU and LSQ.

The issue stage of the pipeline is contained in ruu_issue() and lsq_refresh(). These routines model instruction wakeup and issue to the functional units, tracking register and memory dependences. Each cycle, the scheduling routines locate the instructions for which the register inputs are all ready. The issue of ready loads is stalled if there is an earlier store with unresolved effective address in the LSQ. If the address of the earlier store matches that of the waiting load, the store value is forwarded to the load. Otherwise, the load is sent to the memory system.

The execute stage is also handled in ruu_issue(). Each cycle, the routine gets as many ready instructions as possible from the scheduler queue. The functional units availability is also checked and if they have available access ports, the

instruction are issued. Finally, the routine schedules writeback events using the latency of the function units.

The writeback stage resides in ruu_writeback(). Each cycle it scans the event queue for instruction completions. When it finds a completed instruction, it walks the dependence chain of instruction outputs to mark instructions that are dependent on the completed instruction. If a dependent instruction is waiting only for that completion, the routine marks it as ready to be issued.

Ruu_commit() handles the instructions from the writeback stage that are ready to commit. This routine does in-order committing of instructions, updating of the data caches with store values and data. The routine keeps retiring instructions at the head of the RUU that are ready to commit until the head instruction is one that is not ready. When an instruction is committed, its result is placed into the architected register file and the RUU/LSQ resources devoted to those instructions are reclaimed.

We have simulated the execution of five test-examples that are part of the SimpleScalar Tool Set (math, fmath, llong, lswlr, printf) by changing LSQ size between 4 and 512 entries (Fig. 2).

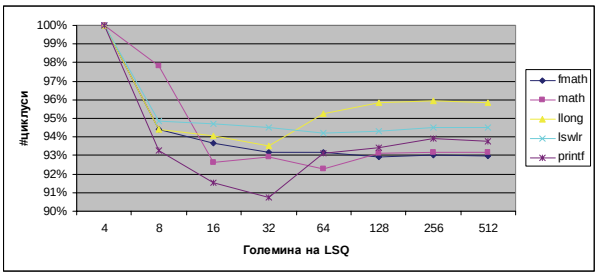

Fig 2. Simulation Results

It is observed that all test-examples follow the same performance trends in the eight different simulations. There is, on average, a 6-9% decrease in the number of execution cycles with the increase of LSQ size up to 32 entries, which is not the case afterwards. The additional speedup becomes insignificant, while for some of the test-examples (llong, printf) the number of execution cycles slightly increased.

V. Conclusion

The LSQ structure enables memory instructions that address the same memory locations to bypass data values among them during Issue and/or Execute phases in the pipeline. However, it is only possible if all memory dependences have already been resolved. As memory instructions get their values from the LSQ (not further down from the memory hierarchy), the execution time is presumably shorter. The main concern is: how the LSQ size affects system performance?

From the previous, we come to a conclusion that even relatively small queue (32 entries in our case) is large enough to perceive performance gain. Further increase in queue size does not lead to additional speedup and, in some cases, it negatively affects system performance.

References

[1] Lam M.S. and Wilson R. P., “Limits of Control Flow on Parallelism”, Proceedings of the 19th Annual Symposium on Computer Architecture; May 1992.

[2] Lipasti M. H and Shen J.P., “Exceeding the Dataflow limit via Value Speculation”, Proceedings of the 29th Annual Syposium on Microarchitecture; December 1996.

[3] Lipasti M. H, Wilkerson C. B. and Shen J. P.,”Value Locality and Load Value Prediction,” Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating System; October 1996

[4] Moshovos A., Breach S.E.,Vijaykumar T. N. and Sohi G. S., ”Dynamic Speculation Synchronization of Data Dependence,” Proceedings of the 24th Annual Symposium on Computer Architecture; June 1997

[5] Gusev M., Mitrevski P., “Modeling and Performance Evaluation of Branch and Value Prediction in ILP Processors”, International Journal of Computer Mathematics, Vol. 80, No. 1, pp. 19-46, 2003

[6] Mitrevski P., Gusev M., “On the Performance Potential of Speculative Execution Based on Branch and Value Prediction”, International Scientific Journal Facta Universitatis, Series: Electronics and Energetics, Vol. 16, No. 1, pp. 83-91, 2003

[7] Patt Y. N., Melvin S.W., Hwu W.W. and Shebanow, “Critical Issue Regarding HPS, A High Performance Microarchitecture,” Proceedings of the 18th Annual ACM/IEEE Workshop on Microprogramming; December 1985

[8] Johnson M., Superscalar Microprocessor Design. Englewood Cliffs N. J.; Printice Hall, 1991

[9] Patterson D.A. and Hennessy J.L. Computer Organization and Design. Morgan Kauffman Publishers, 2005

[10] Sohi G.S., Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers, IEEE Transactions on Computers, Volume 39, Issue 3; March 1990

[11] Austin T.M. and Burger D., The SimpleScalar Tool Set, Version 2.0. Technical Raport 1342, Computer Sciences Department, University of Wisconsin, Medison, June 1997