A Study of Out-of-Order Completion for the MIPS R10K Superscalar Processor

Prabhat Mishra pmishra@cecs.uci.edu
Nikil Dutt dutt@cecs.uci.edu
Alex Nicolau nicolau@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems
University of California, Irvine, CA 92697-3425, USA
http://www.cecs.uci.edu/~aces

Technical Report #01-06
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA

January 2001

Abstract

Instruction level parallelism (ILP) improves performance for VLIW, EPIC, and Superscalar processors. Out-of-order execution improves performance further. The advantage of out-of-order execution is not fully utilized due to in-order completion. In this report we study the performance loss due to in-order completion for MIPS R10000 processor.

Contents

List of Figures

List of Tables

1 Introduction

Many micro-architecture studies focus on issues related to branch prediction, cache size, instruction window size and how design parameters interact [7]. Many studies have focused on available instruction level parallelism (ILP) for superscalar and superpipelined processors [3]. It would be interesting to study how ILP interacts with these design parameters. In this report we study the impact of out-of-order graduation on processor performance.

In this report *completion* of an instruction implies the instruction has completed execution but still in the completion queue. This completed instruction will be removed from the completion queue and results will be committed to register file during *graduation*. *Completion* refers to completion of execution and *graduation* refers to committing the results.

Contemporary superscalar processors use in-order graduation. This is to ensure sequential execution behavior in the presence of out-of-order execution of instructions. This also ensures that all exceptions are reported in program order. Consider a completion queue (active list) of 8 instructions as shown below. The instruction graduates from the queue from the top (index 0). Index column means the index of the queue. Instruction means the instruction stored in that index. The instructions are inserted into the queue in the program order by the decode unit. Due to out-of-order execution different instructions finish (shown as Done in Status column) execution at different point of time. If out-of-order graduation is employed, i.e., if we allow IADD instruction at index 1 to graduate before IVLOAD at index 0 (lower index implies older in sequential program order), sequential execution semantics may not be preserved. Execution will be incorrect if IVLOAD generates exception due to segmentation fault, for example. In sequential execution program will terminate at IVLOAD instruction itself and will never execute IADD, in case of such segmentation fault. However, in this scenario, if we commit the results of IADD before IVLOAD graduates we will generate different memory state than the earlier one if IVLOAD generates exception. Similarly we can not allow DADD to graduate since it is after the branch instruction which is yet to complete. In other words, trying to graduate instructions out-of-order may generate incorrect results in the presence of exceptions, interrupts, branches etc.

```
Index    Instruction         Status

  0    IVLOAD R3, -4, R5   Not Done
  1    IADD R5, R5, 1        Done
  2    ISUB R4, R4, 4        Done
  3    ILE $cc, R4, R5       Done
  4    IF $cc L8           Not Done
  3    IVSTORE R3, -4, R5  Not Done
  6    ILSH   R3, R3, 1    Not Done
  7    DADD   R7, R7, 1      Done
```

The natural question is why we are studying out-of-order graduation if it is incorrect to perform. We want to explore if this out-of-order execution is a performance bottleneck. Moreover, in embedded systems, we do know the application behavior. Hence, it would be appropriate to decide when to perform out-of-order graduation and when not based on expected exceptions and interrupts.

Table 1 shows the fetch, issue, execution and graduation methodology (viz., in-order or out-of-order) for five processors. UltraSparcI and IA64 are more VLIW type. MIPS R10K and Alpha has out-of-order issue and execution. PowerPC has in-order issue but out-of-order execution. All these three processors graduates in-order. In other words, for contemporary processors with out-of-order execution the graduation is always in-order.

MIPS R10K maintains 32 entry **active instruction list**. Instruction issue will be stalled if active instruction list is full. Since graduation is in-order, there may be independent instructions ready to retire/graduate but is not able to do that because of an active instruction in front of it.

Powerpc MPC7400 [2] maintains 8-entry **completion queue**. Issue will be done only if there is empty space in completion queue. Upto 2 instructions can graduate per cycle.

In both cases mentioned above, it may lead to performance penalty which could have been avoided by allowing independent instructions to graduate out-of-order, by adding extra logic which picks up every completed instruction inside the active instruction list (in R10K) and checks the dependency with the instructions above in the queue.

Consider the snapshot of the 8-entry completion queue shown above. The queue is full. Therefore, no more instructions will be decoded till there is space available in the queue. Due to in-order graduation semantics the completed instruction IADD is not allowed to graduate till the IVLOAD instruction graduates. If load is hit in cache then it may held the completed instructions for two cycles. If load is a miss in primary cache then it will stall decode for 4 - 50 cycles depending on where it gets the data, secondary cache or main memory. Though I described the situation for load instruction stopping the graduation process, any instruction with longer latency are potential candidates for this viz., FMUL, FDIV etc.

These independent instructions, which gets completed prior to previous instructions, get generated due to software pipelining, loop index increment, array index computation or loop termination check etc. If we allow IADD to graduate then decode can insert instructions in completion queue.

Table 1. Instruction Fetch, Issue, Execution and Graduation

	Fetch		Issue		Execution		Graduation	
	In-order	OOO	In-order	OOO	In-order	OOO	In-order	OOO
MIPS R10K	4			4		5	x	
Alpha 21264	4			4		6	?	
MPC7400	4		2			8	2	
UltraSparc I	4		4		4			x
IA 64	6		6		9			x

Section 2 describes MIPS R10K processor briefly which we implemented in our cycle-accurate, retargetable SIMPRESS [4] simulator. Section 3 shows the results for out-of-order execution. Section 4 concludes the paper.

2 MIPS R10000 Architecture

The MIPS R10000 ([8], [9]), is a dynamic, superscalar microprocessor that implements the 64-bit Mips-4 instruction set architecture. It fetches and decodes four instructions per cycle and dynamically issues them to five fully-pipelined, low-latency execution units. Instructions can be fetched and executed speculatively beyond branches. Instructions graduate in order upon completion. Although execution is out of order, the processor still provides sequential memory consistency and precise exception handling. With speculative execution, it calculates memory addresses and initiates cache refills early. It's hierarchical, nonblocking memory system helps hide memory latency with two levels of set-associative, write-back caches. To cope with the complexity of out of order superscalar processing, the R10000 uses a modular design that locates much of the control logic with in regular structures, including the active list, register map tables, and instruction queues.

R10000 fetches and decodes four 32-bit instructions per cycle. If one of these is a branch, its target address is calculated, the branch path is predicted, and instructions are speculatively fetched along the predicted path. Decoded instructions are put into a 32-entry *Active List* and three 16-entry instruction queues. The *Active List* keeps track of the original instruction order. The instruction queues dynamically issue each instruction to the appropriate execution unit after all its operands have become available. The *Floating-point Queue* issues instructions to the floating-point multiplier and adder. The *Integer Queue* issues instructions to two ALUs. The *Address Queue* issues instructions to the Load/Store unit (Address Calculation Unit and TLB) and the *Data Cache*. The *Address Calculation Unit* calculates 44 bit virtual memory addresses and TLB translates them to 40-bit physical addresses. Instructions graduate in order upon completion. Although execution is aggressively out-of-order, the processor still provides sequential memory consistency and precise exception handling.

Figure 1 shows the major blocks in the R10000 processor. In the following sections we describe MIPS R10K register files, issue queues and memory hierarchy.

2.1 Register Files

Integer and floating-point register files each contain 64 physical registers. The integer register file has seven read ports and three write ports. These include two dedicated read ports and one dedicated write port for each ALU and two dedicated read ports for the address calculate unit. The integer registers seventh read port handles store, jump-register, and move-to-floating-point instructions. It's third write port handles load, branch-and-link, and move-from-floating-point instructions. The floating-point register file has five read and three write ports. The adder and multiplier each have two dedicated read ports and one dedicated write port. The fifth read port handles store and move instructions; the third write port handles load and move instructions.

2.2 Instruction Pipeline

The instruction pipeline continues to fetch and decode instructions as long as there is room in the Active List and queues. When resource conflicts or operand dependencies prevent the queues

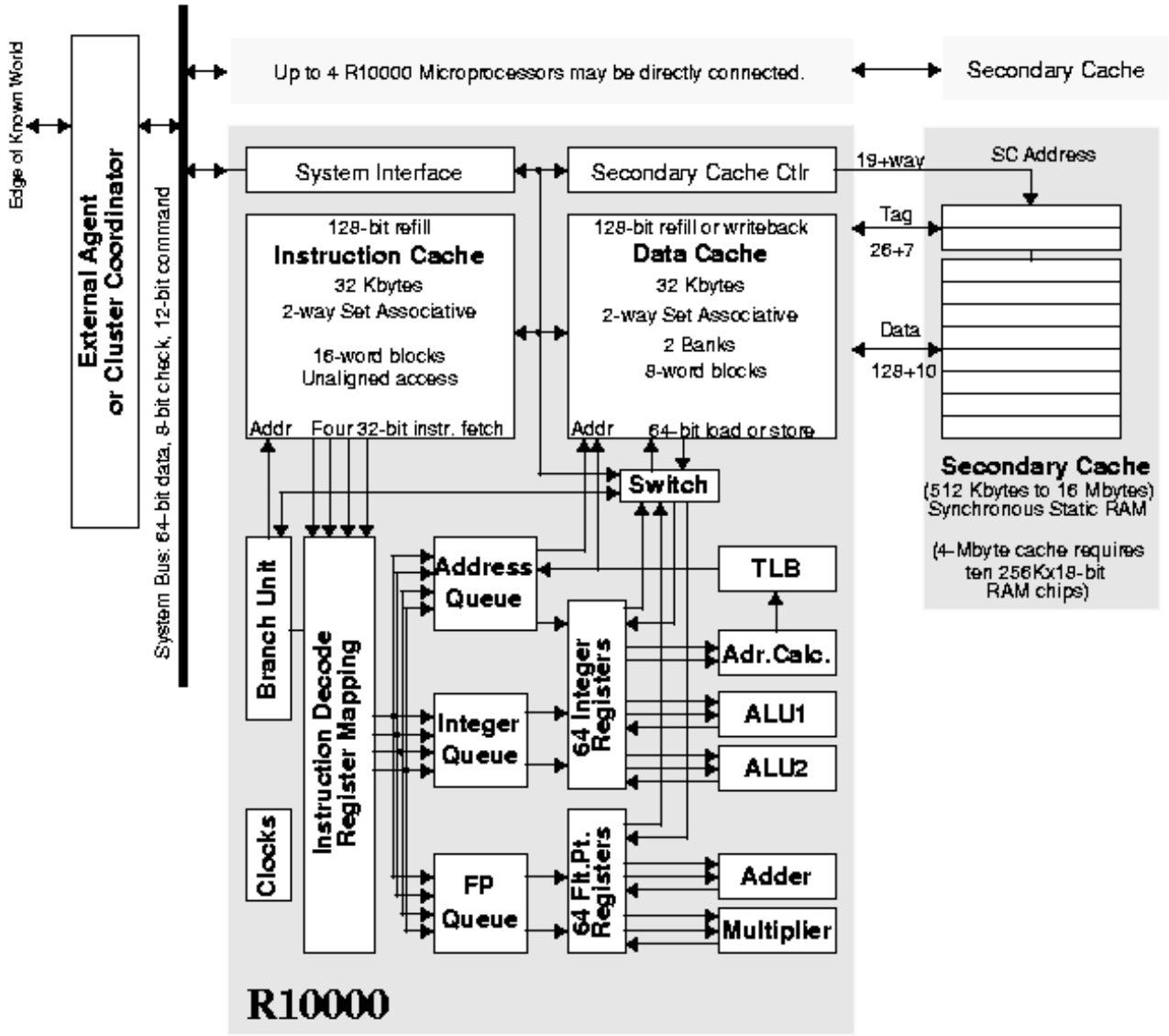

Figure 1. R10000 Microprocessor Block Diagram

from issuing instructions in their program order, the queue's dynamic scheduling hardwire tries to find other instructions that can be issued instead. For frequent operations, each execution unit is fully pipelined with a single-cycle repeat rate. The ALUs execute simple integer operations with single cycle latency, so that dependent instructions can be issued on consecutive cycles. The floating-point units has 3-stage pipelines, but special bypass logic reduces latency to only two cycles. Integer operands are loaded from the Data Cache with two cycle latency. Floating-point loads take an extra cycle of latency, because these units are physically farther from the Data Cache.

2.3 Register Renaming

During instruction decode, integer and floating-point registers are renamed using separate mapping tables. This hardware handles almost any sequence of four instructions, including sequences with dependencies and instructions destined to the same functional units. Renaming maps 32 logical register numbers into 64 physical registers. The physical registers contain both committed and speculative values. When each instruction is decoded, its result is assigned to a physical register from a *Free List* of currently unused registers. At graduation, this register contains a new committed value, and the previously assigned physical register is returned to the Free List. Thus, each physical register is uniquely associated with just one value; dependencies can be determined simply by comparing physical register numbers.

2.4 Branch Prediction

The direction taken by a conditional branch is predicted using a 2-bit algorithm, based on a 512 entry Branch History Table. Each prediction is verified as soon as its branch condition is determined. if its prediction was incorrect, all instructions fetched along the mis-predicted path are immediately aborted, and the processor state is restored from a 4-entry Branch Stack. This allows rapid recovery for up to four mis-predicted branches. Fetching along predicted paths may have initiated unneeded cache refills. However, the cache is non-blocking, and the correct path can be fetched while these refills are completed.

2.5 Integer Queue

The integer queue issues instructions to the two integer arithmetic units: ALU1 and ALU2. The integer queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly decoded integer instructions are written into empty entries in no particular order. Instructions remain in this queue only until they have been issued to an ALU. Branch and shift instructions can be issued only to ALU1. Integer multiply and divide instructions can be issued only to ALU2. Other integer instructions can be issued to either ALU. The integer queue controls six dedicated ports to the integer register file: two operand read ports and a destination write port for each ALU.

2.6 Floating-point Queue

The floating-point queue issues instructions to the floating-point multiplier and the floating-point adder. The floating-point queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly decoded integer instructions are written into empty entries in no particular order. Instructions remain in this queue only until they have been issued to a floating point execution unit. The R10000 has four independent 64-bit floating-point execution units. The adders and multiplier are each fully pipelined with single-cycle repeat rate and latency of just two cycles. The adder includes a leading-zero predictor and a dual-carry-chain adder to round its result before it is normalized. Division and square root are performed in separate iterative units which operate concurrently with the pipelined units. To reduce latency, the divider cascades two stages, so it can generate four bits per cycle. These units share the multiplier's issue and register file ports. Separate ports are not justified, because the issue rate is low.

2.7 Address Queue

The address queue issues instructions to the load/store unit. The address queue contains 16 instruction entries. Unlike the other two queues, the address queue is organized as a circular First-In First-Out (FIFO) buffer. A newly decoded load/store instruction is written into the next available sequential empty entry; upto four instruction may be written during each cycle. The FIFO order maintains the program's original instruction sequence so that memory address dependencies may be easily computed. Instructions remain in this queue until they have graduated; they cannot be deleted immediately after being issued, since the load/store unit may not be able to complete the operation immediately. An issued instruction may fail to complete because of a memory. a cache miss, or a resource conflict; in these cases, the queue must continue to reissue the instruction until it is completed. The address queue has three issue ports:

- First, it issues each instruction once to the address calculation unit. The unit uses a three stage pipeline to compute the instruction's memory address and to translate it in the TLB. This port controls two dedicated read ports to the integer register file. If the cache is available, it is accessed at the same time as the TLB. A tag check can be performed even if the data array is busy.

- Second, the address queue can re-issue accesses to the data cache. The queue allocates usage of the four sections of the cache, which consist of the tag and data sections of the two cache banks. Load and store instructions begin with a tag check cycle, which checks to see if the desired address is already in cache. If it is not, a refill operation is initiated, and this instruction waits until it has completed. Load instructions also read and align a doubleword value from the data array. If the data is present and no dependencies exist, the instruction is marked done in the queue.

- Third, the address queue can issue store instructions to the data cache. A store instruction may not modify the data cache until it graduates. Only one store can graduate per cycle, but

it may be anywhere within the four oldest instructions, if all previous instructions are already completed.

2.8 Memory Hierarchy

R10000 implements a nonblocking memory hierarchy with two levels of set-associative caches. It finds cache misses early, and begins refills in parallel with other useful work. The on-chip caches provide concurrent access for instruction fetch, data load and store, and refill. All caches least-recently-used(LRU) replacement algorithm.

The Data Cache is 2-way interleaved with independent tag and data arrays for each bank. These four arrays operate under shared control of the Address Queue and the External Interface. The queue concurrently processes up to 16 load and store instructions in four separate pipelines. It dynamically issues instructions to the address calculation and translation pipeline. The other three pipelines can concurrently perform tag checks, execute loads, and graduate store instructions. Although address calculation and loads can occur out-of-order, instructions appear to execute with strong memory order. The Data Cache pipelines interact during several common cache sequences. Loads can perform the tag check and data read during the same cycle as the address translation. Store instructions immediately issue a tag check to initiate any required replacement as early as possible, but writing data into the cache is delayed until the store becomes the oldest instruction and graduates. A miss in the primary data cache will initiate a refill sequence from the secondary cache. For loads, refill data can be bypassed directly into the register file. The two banks of the Data Cache are each divided into two logical arrays to support 2-way set-associativity. The processor simultaneously reads the same doubleword from both cache ways, because it checks the cache tags in parallel and later selects data from the correct way. The external interface refills or writes back quadwords by accessing two doublewords in parallel. This is possible because the cache way is known in advance.

The primary cache consists of 2K doublewords. Number of data lines is 8. To store a doubleword 8 locations are needed. Least significant 3 bits are used for this alignment. Address lines 3-13 is used to access the Data Cache and 5-13 is used to access the Tag Cache. Block size is 4. Tag value consists of the bits 14-39 of the physical address. It has read/write latency of 1 cycle. The secondary cache consists of 512K doublewords. Address lines 3-21 is used to access the Data Cache and 6-21 is used to access the Tag Cache. Block size is 16. Tag value consists of the bits 22-39 of the physical address. It has read/write latency of 2 cycles. After a primary cache miss, two quadwords are read in the following sequence from the secondary cache. First, its tag is read along with the first quadword. The tag of the alternate way is read with the second quadword. If the first way hits, the data is available immediately. If the alternate way hits, the secondary cache is read again. If neither way hits, the secondary cache must be refilled from Main Memory. Main Memory has read/write latency of 4 cycles.

3 Experiments

In this section we demonstrate the performance improvement due to out-of-order graduation in MIPS R10K processor.

3.1 Experimental Setup

We performed a set of experiments using R10K processor model in SIMPRESS [4]. Figure 2 shows the R10K processor model we implemented in VSAT-GUI [4].

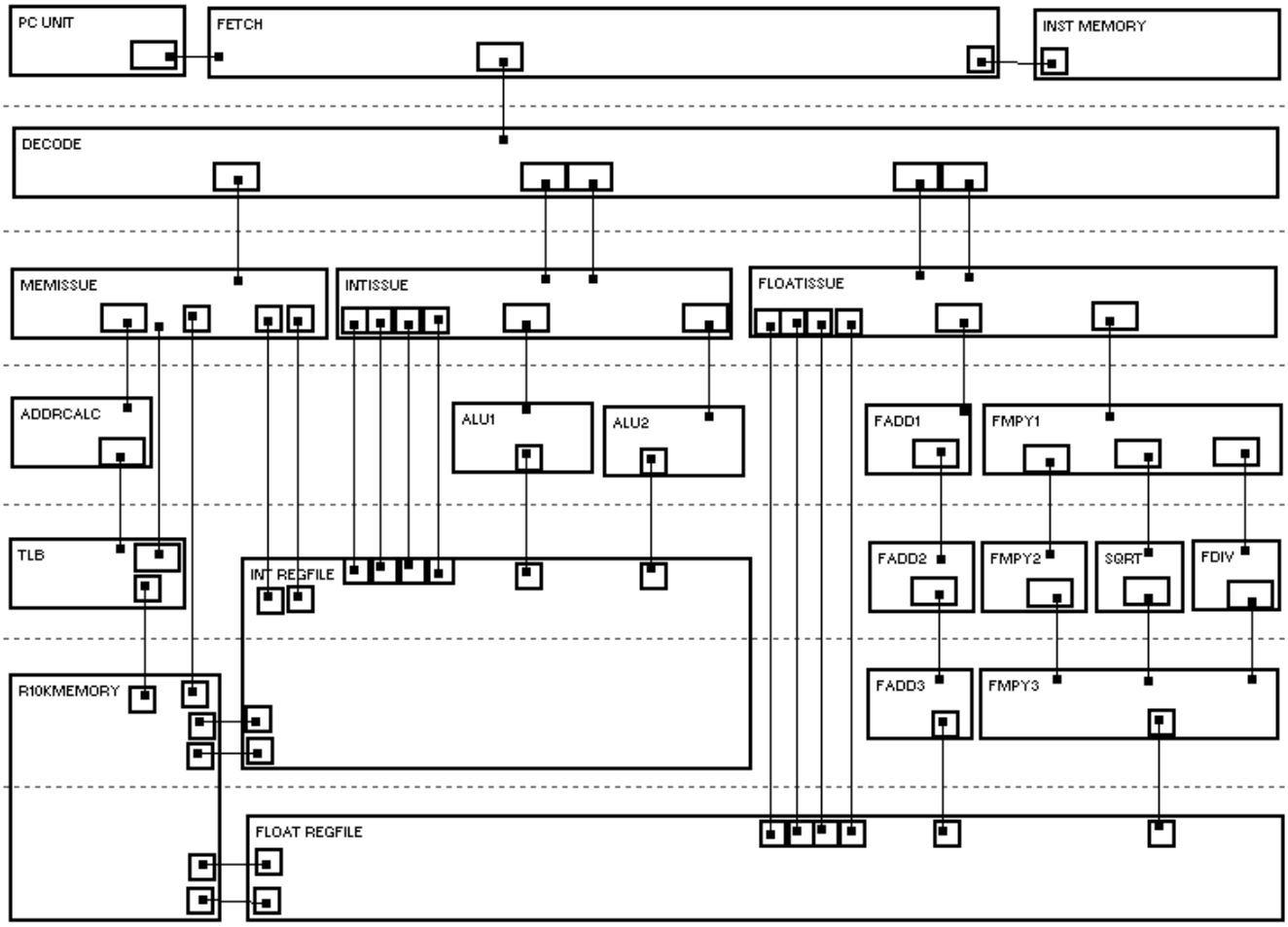

Figure 2. R10000 Microprocessor Model in VSAT GUI

We have used 8-entry *Active List* and allowed out-of-order graduation. In other words, an completed operation is allowed to graduate and commit results while there are operations ahead of it and not graduating in the same cycle. We allow the operations to graduate out-of-order when the prior instructions are not potential candidate for exceptions and do not impose control dependency. The problem of data dependency does not arise because that is already taken care of before exe-

cution. In other words, if this instruction had data dependency with the prior instruction then this instruction would have never completed execution successfully.

We have used a set of benchmarks from DSP and multimedia domains, and compiled them using the EXPRESS [1] compiler. We collected the statistics information using the SIMPRESS [4] cycle-accurate structural simulator, which models MIPS R10000 processor and memory subsystem.

3.2 Results

Table 2 presents a set of experiments we ran to study the performance improvement due to out-of-order graduation. First column shows the benchmarks used. Second column presents the total cycle counts needed for the benchmarks using in-order graduation. Column 3 presents the cycle counts needed to execute the benchmarks using out-of-order graduation. Column 4 shows the performance improvement. We observe an average performance improvement of 6.56%.

As expected the performance decreases if the load instruction misses in the cache and stalls the decode stage when active list is full. Table 3 presents the performance improvement with varying cache hit ratio for the benchmark StateExcerpt. This result shows the impact only due to load instructions. The first column shows the cache hit ratio. Second column presents the cycle counts for different hit ratios during in-order graduation. Column 3 presents cycle counts for out-of-order graduation. The last column shows the performance improvement. When hit ratio is 1.00 (row 1), i.e., there is no cache miss, the load operation completes in 2 - 4 cycles and thereby allows the completed instructions to graduate soon. However when hit ratio is less than 1.00, implies there are load operations which stalls the decode stage for long time and holds the completed instructions in the active list during in-order graduation. As a result we observe increased performance improvement when hit ratio decreases. We get 27% performance improvement for the benchmark StateExcerpt when all the load operation misses. This is the upper bound for this benchmark considering the active list size and load misses.

4 Summary

We present here a study of performance impact of out-of-order graduation for MIPS R10K processor. The basic observation is that the longer latency operations e.g., load, multiply, division etc. are holding the smaller latency instructions (completed execution) in active list. As a result the active list is getting full and hence decode is getting stalled. The performance impact becomes more observable when the load operation misses.

A careful out-of-order graduation has shown 6.56% performance improvement on an average for DSM and multimedia kernels. The independent instructions, the potential candidate for out-of-order graduation, are generated due to software pipelining, loop index improvement, array index computation or loop termination check etc. We show that long latency operations contribute a lot to the performance loss. The load miss situation is noticeable. However we have not studied the impact of different memory subsystem [5] and thereby different load miss situation on out-of-order graduation.

Benchmarks	In-order graduation	Out-of-order graduation	% improvement
Hydro	518	478	7.72
ICCG	1451	1353	6.75
InnerProd	339	313	7.67
LinearEq	1641	1533	6.58
TriDiag	492	456	7.32
RecurEq	11701	11541	1.37
StateExcerpt	543	458	15.65
Integrate	1820	1684	7.47
DiffPred	3058	2901	5.13
FirstSum	312	296	5.13
FirstDiff	221	205	7.24
2DPartPush	12741	12700	0.32
1DPartPush	2830	2586	8.62
CondComp	9408	9342	0.70
2DHydro	6460	5972	7.55
CondRecur	830	704	15.18
LL20	2109	1899	9.96
LL21	3122	2932	6.09
LL22	1086	975	10.22
LL23	2802	2742	2.14
FirstMin	460	460	0.00
Compres	1724	1724	0.00
Laplace	2116	2080	1.70
Linear	830	704	15.18
Lowpass	2821	2561	9.22
Wavelet	433	409	5.54
		Average	6.56

Table 2. Performance improvement due to out-of-order graduation

Cache Hit Ratio	In-order graduation	Out-of-order graduation	% improvement
1.00	543	458	15.65
0.99	8361	6586	21.23
0.92	8319	6088	26.82
0.00	8421	6147	27.00

Table 3. Impact of cache misses on out-of-order graduation for benchmark StateExcerpt

When we try to schedule the operations differently in the Trailblaze scheduler [6] (in EXPRESS compiler) using the property of shorter independent instructions first, we do not get much performance improvement. In fact, at times it generates worse results. This is an interesting observation. This shows that the static scheduling will of no use for these benchmarks.

Our future work involves studying the effect of the scheduling on bigger benchmarks and also how different scheduling techniques (e.g., smaller latency operations first etc.) interfere with different active list size.

5 Acknowledgments

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and Motorola Corporation. We would like to acknowledge and thank all the EXPRESSION team members for their contributions to this work.

References

[1] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.

[2] http://www.motorola.com/SPS/PowerPC. *MPC7400 PowerPC Microprocessor*.

[3] N. Jouppi and D. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *The 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*.

[4] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proc. EUROMICRO*, 1999.

[5] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-exploration driven by an architectural description language. In *Intl. Conf. on VLSI Design 2001*, Bangalore, India, 2001.

[6] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *ICPP*, St. Charles, IL, 1993.

[7] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction window size, and cache size: Performance tradeoffs and simulation techniques. In *IEEE Transactions on Computers*, Nov. 1999.

[8] N. Vasseghi, K. Yeager, E. Sarto, and M. Seddighnezhad. 200-mhz superscalar risc microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1675–1686, November 1996.

[9] K. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.