On Reducing Load/Store Latencies of Cache Accesses

Jia-Jhe Li
Department of Computer Science
National Taiwan Ocean University
Keelung 20224
Taiwan
jjli@pllab.cs.ntou.edu.tw

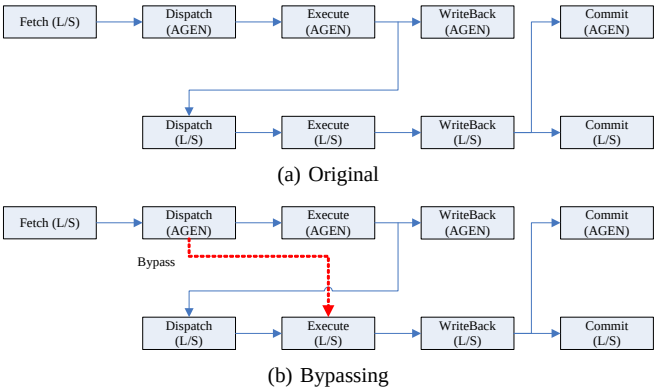
Yuan-Shin Hwang
Department of Computer Science
National Taiwan Ocean University
Keelung 20224
Taiwan
shin@cs.ntou.edu.tw

***Abstract*— Effective address calculation for load and store instructions needs to compete for ALU with other instructions and hence extra latencies might be incurred to data cache accesses. Fast address generation is an approach proposed to reduce cache access latencies. This paper presents a fast address generator that can eliminate most of the effective address computations. Experimental results show that this fast address generator can reduce effective address computations of load and store instructions by about 74% on average for SPECint2000 benchmarks and cut the execution times by 8.5%. In addition, further improvement can be made if data of previous load operations are buffered in the unused data field of LSQ entries as well. Runtime impact will expand to 10.5% on average when the default LSQ is modified to the cached LSQ design.**

I. Introduction

Each data cache access of a load or store instruction can be broken into two phases: effective address computation (AGEN) phase and cache access (L/S) phase. Modern processors often divide a load or store instruction into two sub-instructions to match these two phases. For instance, when a load or store instruction of Alpha 21264 enters the pipeline [1], [7], the *AGEN* and *L/S* phases will be treated as two consecutive and dependent instructions, as shown in Figure 1(a). After the effective address is computed at the *Execute* stage, the result will be sent to the next pipeline stage and meanwhile be forwarded to the *Dispatch* stage of the *L/S* phase as well. Therefore, the *WriteBack* stage of *AGEN* and the *Dispatch* stage of *L/S* can then be executed simultaneously in the same clock cycle. In the *Execute (L/S)* stage, the computed address will be compared with the addresses in the LSQ and the tags of L1 cache to determine if the operand can be obtained from a normal store-to-load forwarding in LSQ or be read from a cache block.

Since the *AGEN* phase of a load or store instruction needs to compete for ALU with other instructions to compute the effective address, extra latencies might be incurred to data cache accesses. In order to minimize the impact of the *AGEN* phase on performance, researchers have proposed various fast address generation techniques to reduce cache access latencies [2], [3], [4], [5], [8], [10]. They have developed polices and implementations to predict or look up the effect address of a load or store instruction at the *Dispatch* stage



(a) Original

(b) Bypassing

Fig. 1. Pipeline Stages of Load and Store Instructions

of the *AGEN* phase. As a result, the *L/S* phase a load or store instruction can enter its *Execute (L/S)* stage to start cache access before the *Execute (Agen)* stage actually computing the effective address. If the effective address is looked up from a buffer, *Execute* stage of the *AGEN* phase can be skipped, as depicted by the *Bypass* path in Figure 1(b). On the other hand, if the effective address is predicted, *Execute* stage of the *AGEN* phase will be executed in parallel with *Execute* stage of the *L/S* phase to determine if the speculated effective address is correct. When the address speculation is wrong, *Execute* stage of the *L/S* phase will be performed again with the computed effective address to access the right data.

These proposed methods look for one of the following three possible sources of fast address computations:

- load or store instructions with zero offsets [8],
- load or store instructions with small non-negative offsets, i.e. instructions whose computations of effective addresses do not generate carries [2], [3], or
- load or store instructions whose effective addresses have been computed recently, i.e. instructions whose effective addresses can be looked up from a buffer [4], [5], [10].

This paper presents a fast address generator that identifies load and store instructions of all three categories. Consequently, this fast address generator can eliminate most of the effective address and hence can effectively reduce latencies of data

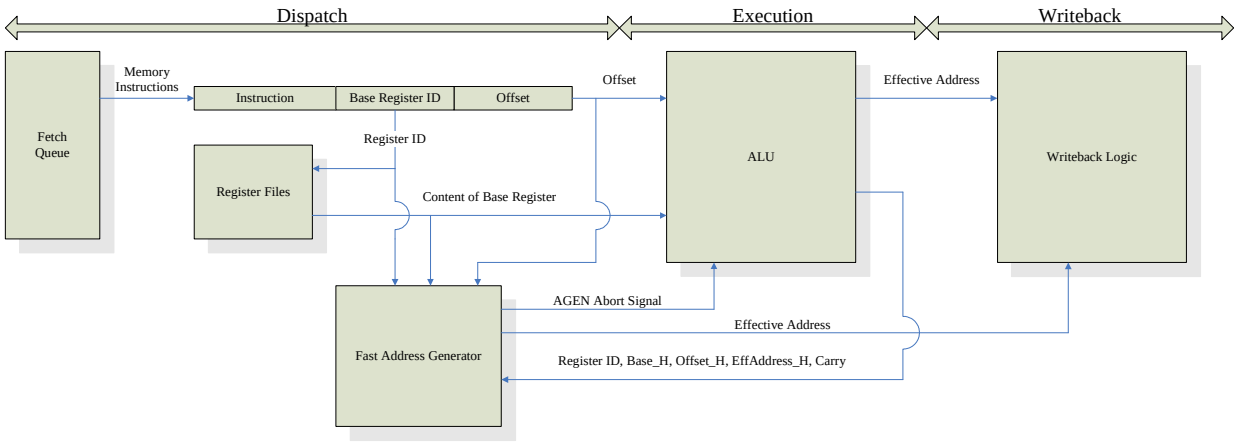


Fig. 2. Datapath

cache accesses.

Experimental results show that this fast address generator can reduce effective address computations of load and store instructions by about 70% on average for SPECint2000 benchmarks [11] and cut the execution times by 8.5%. In addition, further improvement can be made if data of previous load operations are buffered in the unused data field of LSQ (load/store queue) entries [9]. Execution times reduction can achieve 10.5% on average when a cached LSQ is applied since the pipeline stage of accessing the target cache line can be skipped as well.

The main results of this paper are as follows:

- A fast address generator is presented that identifies load and store instructions of all three categories as mentioned above.
- This technique eliminates effective address calculation of 74% load and store instructions for SPECint2000 benchmarks, which results in 8.5% reduction of average executiont times.
- Cached LSQ can further extend the performance improvement by another 2% to reach 10.5%.

The rest of this paper is organized as follows. Section II introduces the fast address generator and Section III briefly describes the cached LSQ. Experimental results will be presented in Section IV. Section V concludes this paper.

II. Fast Address Generation

The goal of fast address generation is to predict the effective addresses of as many load and store instructions as possible at *Dispatch* stage.

A. Datapath

The SimpleScalar pipeline is modified to accommodate a fast address generator that predicts effective addresses of load and store instructions [1]. Figure 2 depicts the datapath of the simulated Alpha 21264 pipeline [7]. As a load or store instruction enter the *Dispatch* stage from the fetch queue, its base and offset will be fed into the fast address generator. If the effective address of this instruction can be predicted, the effective address computation of this instruction by ALU will be aborted, i.e. the *AGEN* phase of this instruction will skip the *Execution* stage and go to the *WriteBack* stage directly. The predicted effective address will be forwarded to the *WriteBack* stage of the *AGEN* phase, where the *WriteBack* logic will update the information of the *L/S* phase of the load or store instruction in the issue queue.

B. Fast Address Generator

In order to predict the effective addresses of most load and store instructions at the *Dispatch* stage of *AGEN* phase, this fast address generator will look for all the three possible sources of fast address computations listed in Section I. Fortunately, the first two types of instructions can be combined, since zero-offset instructions can be categorized into instructions with small non-negative offsets as well. In other words, computations of effective addresses for these instructions will not generate carries. Consequently, the fast address calculation mechanism proposed by by Austin et. al. can be used to perform carry-free addition, as the *NAND-OR Check* component depicted in Figure 3.

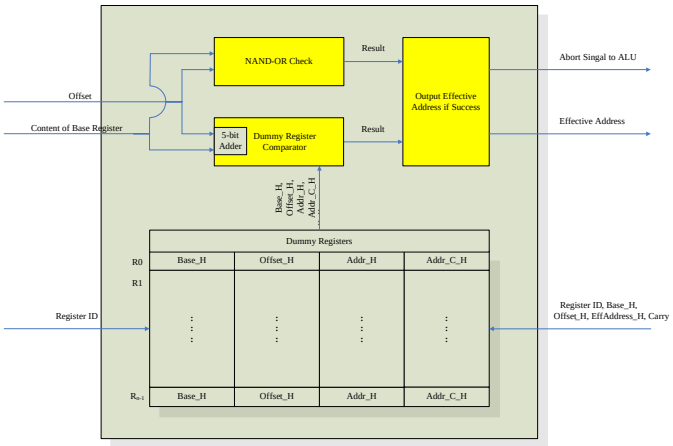


Fig. 3. Fast Address Generator

As for the third category, this address generator looks for load or store instructions whose effective addresses have been computed recently. That is, instructions that repeatedly access those cache blocks in the current working set will fall into this category. This paper deploys a *dummy register file* to store the bases, offsets, and effective addresses of recent load and store instructions, as shown in Figure 3. When a load or store instruction enters the *Dispatch* stage, the *dummy register comparator (DRC)* consults the dummy register file to check if the effective address has already been computed.

The *NAND-OR Check* component and *DRC* work concurrently to predict effective addresses of load and store instructions before the *Execute* stage. Effective address calculation will be skipped if one or both components hit.

C. Dummy Register Comparator (DRC)

Spatial locality of programs indicates that other words of a cache block will likely be accessed in the near future when a word of the cache block is referenced. As a result, data accesses with the same cache tag but different byte offsets might be clustered in a short time period. Therefore, it is not effective for *DRC* to look for an entry in the dummy registers that matches exactly with the base, offset, and effective address of the current load or store instruction. Instead, *DRC* only compares the tag part (i.e. higher bits excluding bits for byte offset in a cache block) of the base and offset of every load or store instruction and then reports the tag part of the effective address if a match is found. For instance, for a data cache with a block size of 32 bytes, the lowest 5 bits (i.e. the byte offset part) will be ignored during buffer lookup. The byte offset part of effective addresses will be computed by a 5-bit parallel adder, which can be implemented by just two level of AND-OR logic.

1) Dummy Register File: Figure 3 depicts the structure of the dummy register file that buffers the tag part of bases, offsets, and effective addresses of recently executed load and store instructions. Each entry is associated with a processor register. When a load or store instruction enters the *WriteBack (AGEN)* stage after executing effective address calculation, the tag part of its base register will be stored in its corresponding entry in the *Base_H* field and its offset in the *Offset_H* field. In addition, the effective address of this load or store instruction will be buffered in the corresponding entry as well. However, a single *Addr_H* field is not enough, since the byte offset part of effective address calculation by the 5-bit parallel adder might generate a carry. Therefore, two fields are used to store the two possible effective addresses that can be obtained from the same base and offset, depending on the carry of the parallel adder. That is, the *Addr_H* field is used when no carry is generated by the parallel adder; Otherwise, calculated effective address will be stored in the *Addr_C_H* field.

2) Update Policies: This paper has implemented the following two policies for updating the dummy register file.

DRC.always: The corresponding dummy register of the base register used in a load or store instruction is updated when the instruction enters the *WriteBack (AGEN)* stage.

DRC.agen: The corresponding entry of the dummy register file is updated when the *AGEN* phase of a load or store instruction leaves the *Execute* and enters the *WriteBack* stage. In other words, only those load and store instructions that actually perform effective address calculation will update the dummy register file.

III. Cached LSQ

Further runtime saving can be observed if the data of load instructions are buffered in the unused *data* field of LSQ entries as well, i.e., the same technique as the *cached LSQ* proposed by Nicolaescu et al. [9]. When the effective address of a load instruction matches any LSQ entry, the value can be looked up directly from the *data* field. As a result, the instruction can jump directly to the *WriteBack* stage of its *L/S* phase and broadcast the loaded value. Figure 4 depicts this bypassing path from the the *Dispatch* stage of the *AGEN* phase to the *WriteBack* stage of its *L/S* phase, which would translate to an extra saving of one pipeline stage.

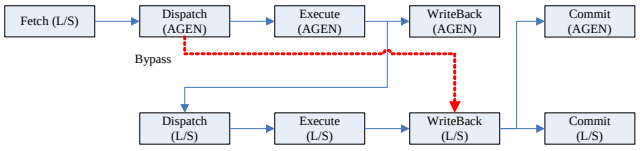


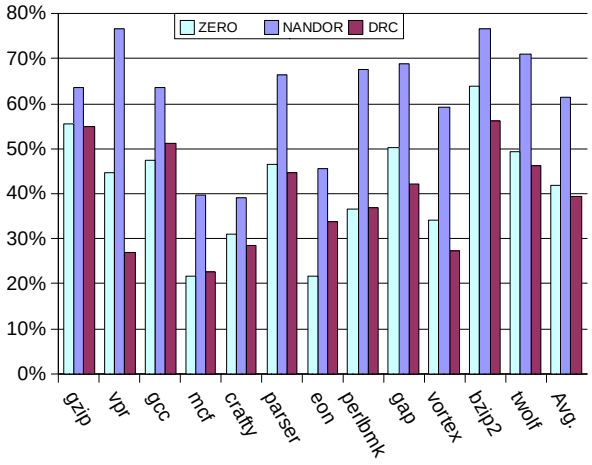
Fig. 4. Bypassing of Cached BO-LSQ

Processor Core	
Instruction Window	64 RUU, 32 LSQ entries
Issue Width	4 instructions/cycle
Functional Units	4 IntALU, 1 IntMult, 4 FpALU, 1 FpMult, 2 Memory Ports
Pipeline	5 Stages: Fetch/Dispatch/Execute/WriteBack/Commit
Memory Hierarchy	
L1 Dcache	64KB, 32B blocks, 4-way, 1-cycle latency
L1 Icache	64KB, 32B blocks, 4-way, 1-cycle latency
L2 cache	1MB, 64B blocks, 8-way LRU, 6-cycle latency
Memory	100-cycle latency

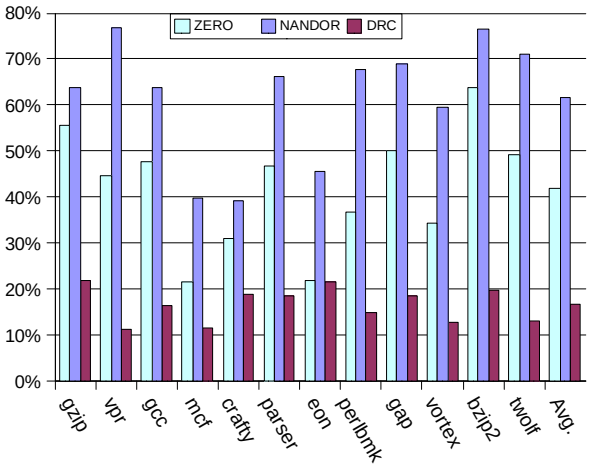
TABLE I
CONFIGURATION OF SIMULATED PROCESSOR

	Issued Instructions				Committed Instructions			
	Total	Loads	Stores	L+S(%)	Total	Loads	Stores	L+S(%)
gzip	74942.6	18898.7	4237.7	30.9%	57970.2	14261.9	3519.4	30.7%
vpr	13527.6	4488.1	1470.9	44.1%	10709.2	3524.1	1192.6	44.0%
gcc	6764.9	2374.4	734.1	45.9%	5117.1	1774.2	588.1	46.2%
mcf	14149.0	4758.8	859.4	39.7%	9168.1	2901.8	532.0	37.5%
crafty	35474.7	11017.3	1651.7	35.7%	27216.4	8271.0	1398.6	35.5%
parser	17934.9	5094.5	1517.6	36.9%	13434.1	3610.7	1219.6	36.0%
eon	3600.1	1074.7	700.2	49.3%	2642.1	771.2	533.9	49.4%
perlbmk	37509.4	12161.9	5295.6	46.5%	27622.3	8842.9	3880.1	46.1%
gap	12726.6	3906.2	1127.9	39.6%	9517.7	2899.1	850.5	39.4%
vortex	19231.8	5325.9	3002.1	43.3%	17813.6	4882.3	2819.8	43.2%
bzip2	65239.5	17217.1	5210.2	34.4%	61127.5	16075.1	4825.2	34.2%
twolf	17274.0	4418.7	1160.0	32.3%	13200.0	3284.5	895.4	31.7%

TABLE II
INSTRUCTION COUNTS (IN MILLIONS)

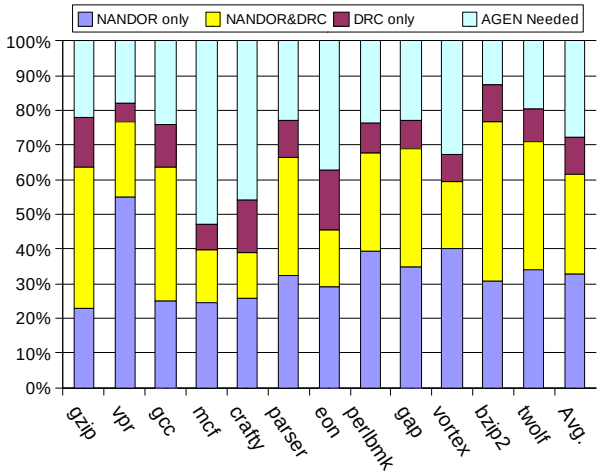


(a) DRC.always

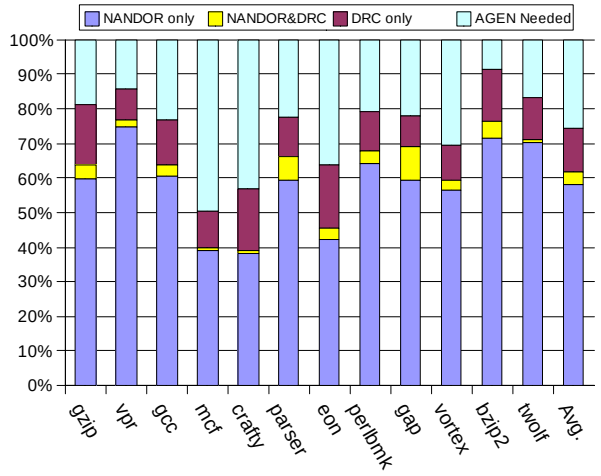


(b) DRC.agen

Fig. 5. Efficiencies of Three Different Approaches



(a) DRC.always



(b) DRC.agen

Fig. 6. Ratio Breakdown of Fast Address Computations

IV. Experimental Results

A. Setup

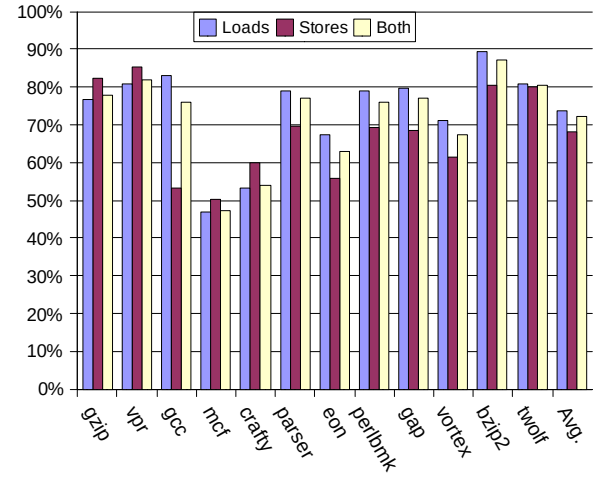
Experiments are conducted by executing the SPECint2000 benchmarks [11] with the TRAIN data set on SimpleScalar v3.0d [1]. The simulated processor is a 21264-like processor [7], and the parameters of the processor and memory hierarchy set by SimpleScalar are listed in Table I. The SPECint2000 binaries with SPEC peak settings were downloaded from the University of Michigan through a link at the SimpleScalar home page [12]. Instead of just collecting a small trace of each program as done by other researchers, this paper chose to simulate the entire program of every SPECint2000 benchmark to avoid the pitfall that a program's locality behavior is not constant over the run of the entire program [6]. Table II lists the numbers of instructions that are issued and committed when no fast address computation is applied. The number of committed instructions are smaller than the numbers of issued instructions because of speculation.

B. Performance Evaluation

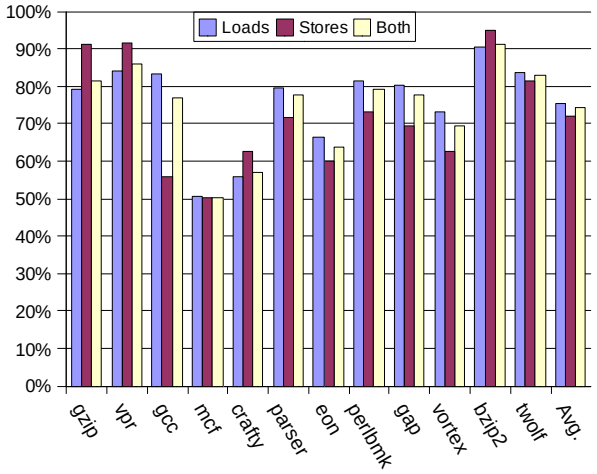
Efficacy

Figure 5 depicts the effectiveness of the three approaches of fast address computations listed in Section I under the two different update policies. This figure shows that 42% of load and store instructions are with zero offsets (i.e. *ZERO*), while 62% of loads and stores uses small offsets (i.e. *NANDOR*). Update polices will not affect these two approaches, since no lookup operations on the dummy register file are performed. However, update policies can significantly affect the efficiencies of *DRC*. Figure 5a demonstrates that *DRC.always* can predict the effective address of 39% load and store instructions, while Figure 5b shows that *DRC.agen* only eliminates 17% of effective address computations.

The fast address generator presented in Section II combines these techniques and hence looks for load and store instructions of all these three categories. As a result, it can identify more opportunities to skip the address computation phase of cache accesses. Figure 6(a) shows that under the

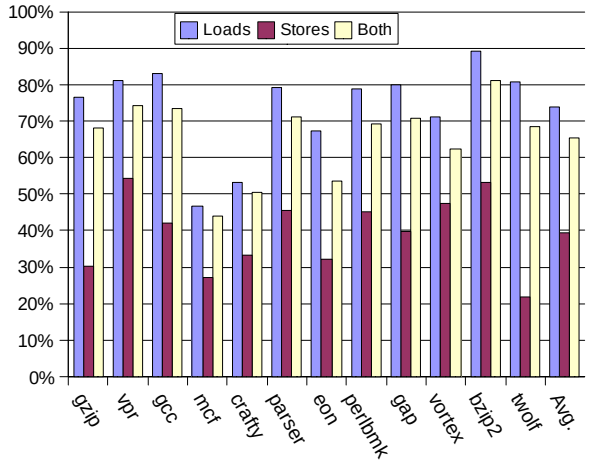


(a) DRC.always

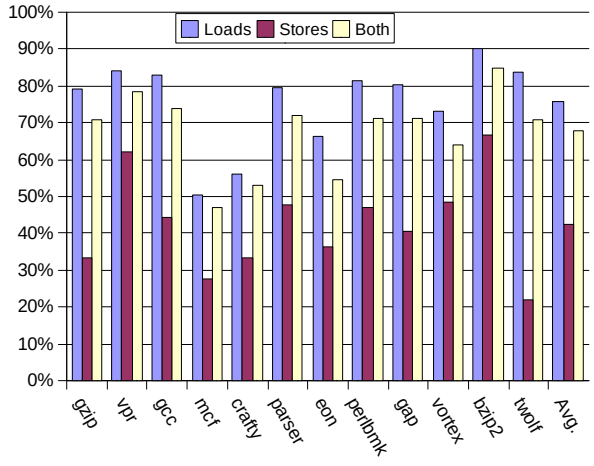


(b) DRC.agen

Fig. 7. Bypass Ratios of Loads and Stores



(a) DRC.always



(b) DRC.agen

Fig. 8. Early Ready Ratios

DRC.always policy this fast address generator can eliminate 72.1% of effective address calculation of load and store instructions on average, which is 10% more than the *NANDOR* approach alone. Out of the 72.1% reductions, 32.8% comes from *NANDOR*, 28.7% is recognized by both *NANDOR* and *DRC*, and the rest 10.6% is contributed by *DRC* alone. This result suggests that the effect addresses of a fairly large number of load and store instructions will be predicted by the both *NANDOR* and *DRC* components.

Although the *DRC.agen* policy looks quite inefficient when compared with the *DRC.always* policy, more effective address computations can be skipped when combining *DRC* under the *DRC.agen* policy with *NANDOR*, as shown in Figure 6(b). The reduction rate now reaches 74.4% and the common part is lowered to only 3.7%. This outcome demonstrates that *DRC* under the *DRC.agen* policy complements *NANDOR* very well. Now only one fourth of load and store instructions need to actually perform effective address computations.

Figure 7 displays the efficiencies of this fast address generator from a different point of view, i.e. by comparing the ratios of load and stores instructions that can skip effective address calculation stage. Figure 7(a) shows the bypass ratio of load instructions is 5.9% higher than that of store instruction under the *DRC.always* policy, while the difference is reduced to 3.6% under *DRC.agen*.

When the effective address of a load or store instruction is predicted by the fast address generator and its entry in the issue queue is updated with the predicted address, it is not necessarily ready to be issued to the *Execute* stage because of dependencies. Figure 8 depicts that about half of predicted store instructions will not be ready for execution until dependencies are resolved.

DRC.always policy makes *DRC* update the dummy registers more times than *DRC.agen*. Figure 9 shows there are 0.55 updates per cycle under *DRC.always*, while *DRC.agen* performs only 0.14 modifications per cycle on average. Therefore, *DRC.agen* is more efficient than *DRC.always*. Figure 10 also demonstrates that *DRC.agen* is a better policy, since it gets more hits per update than *DRC.always*.

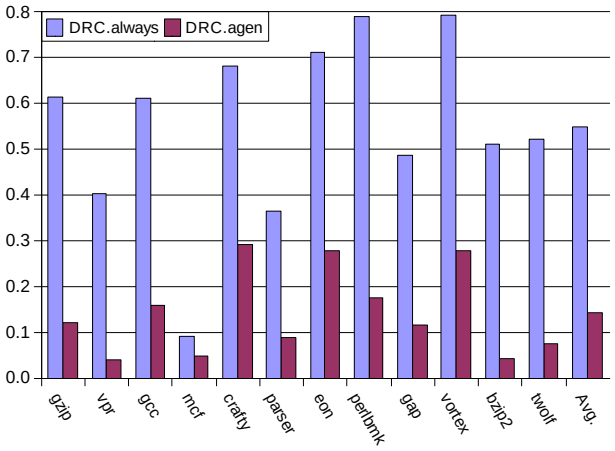


Fig. 9. Updates per Cycle

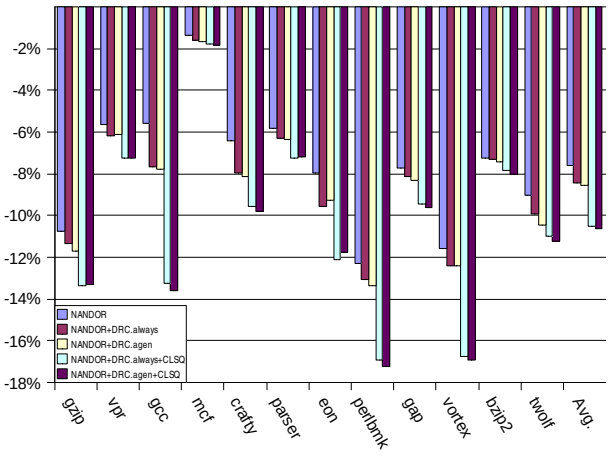


Fig. 12. Run Time Impact

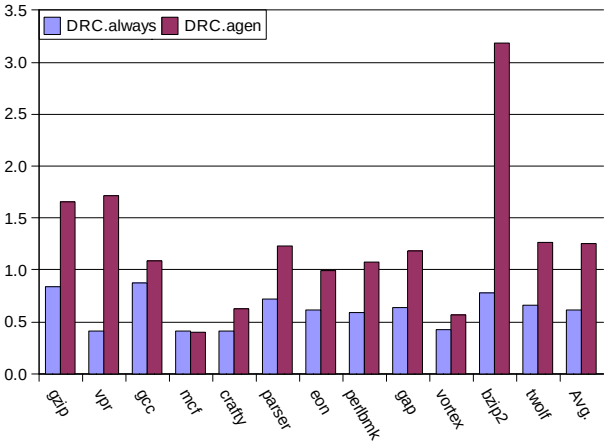


Fig. 10. Hits Per Update

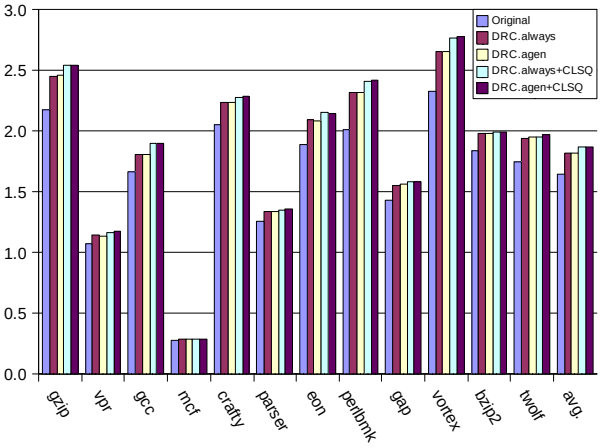


Fig. 13. IPC

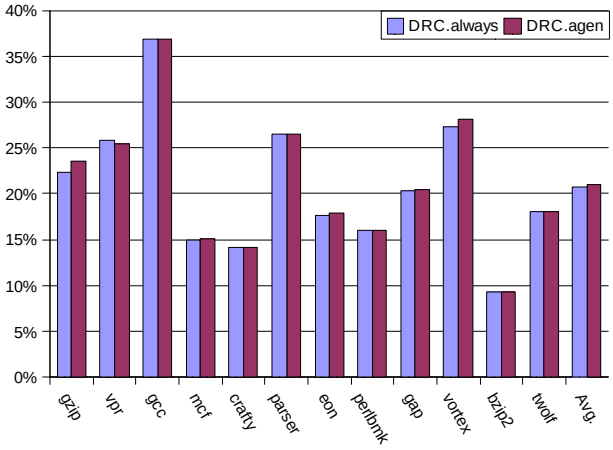


Fig. 11. Loads Forwarded by Cached LSQ

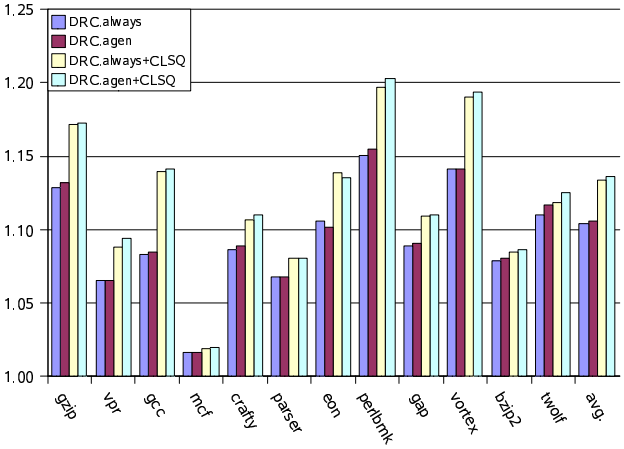
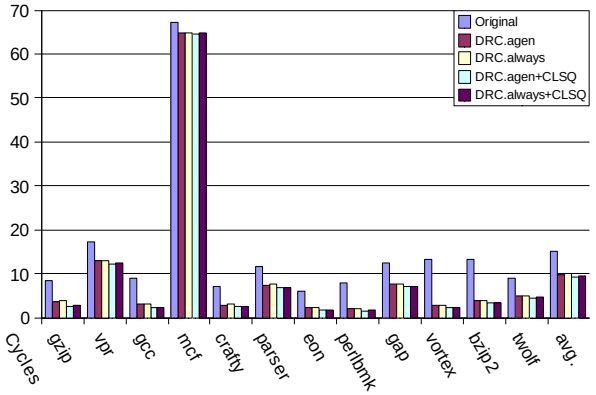
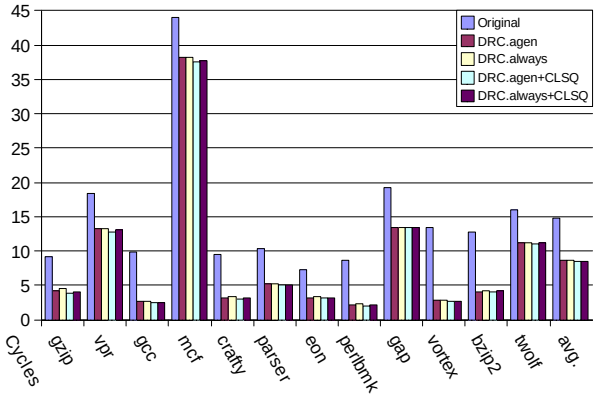


Fig. 14. Speedups

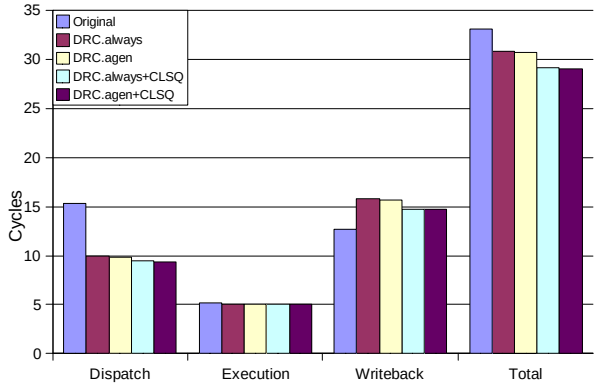


(a) Loads

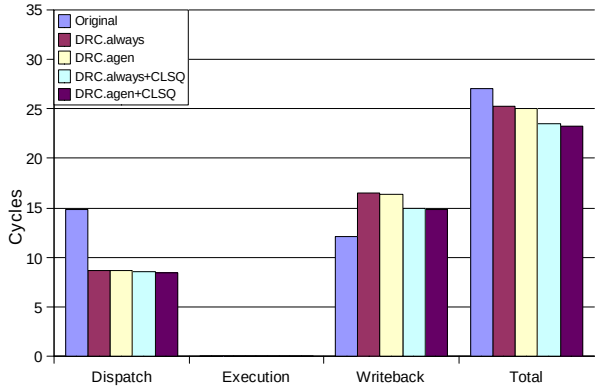


(b) Stores

Fig. 15. Intervals between *Dispatch* to *Execute*



(a) Loads



(b) Stores

Fig. 16. Breakdown of Average Load/Store Latencies

Cached LSQ

The cached LSQ stores the data of load instructions in the unused *data* field of LSQ entries. Consequently, the buffered data can be forwarded to load instructions without actually accessing cache blocks when the effective addresses matches *address* field of LSQ entries. Figure 11 reveals that data of about 20% of load instructions can be retrieved directly from LSQ entries. Skipping cache block accesses can further reduce load/store latencies.

Run Time Impact

Fast address calculation can effectively reduce load/store latencies by eliminating most of effective address calculation of load and store instructions. As a result, performance can be improved, as shown in Figure 12. When only the effective address computations of load and store instructions with zero or small offsets are bypassed by *NANDOR* approach, average execution time for SPECint2000 benchmarks is reduced by 7.6%. Combining *NANDOR* and *DRC* can extend the runtime reduction to 8.5%. The best performance improvement can be achieved when cached LSQ design is incorporated as well, and the impact on execution times reaches 10.5%.

Figure 13 shows that the average IPC for for SPECint2000 benchmarks is improved from 1.65 to 1.82 by the fast address generator alone, while combining the cached LSQ and the fast address generator can push the average IPC to 1.87. Figure 14 translates the IPC improvement to speedups. The speedup of deploying fast address generation is 1.11, whereas incorporating the cached LSQ as well improves the speedup to 1.14.

Prediction efficiency does not translate directly into performance improvement. Eliminating over 70% of effective address calculation of load and store instructions only produces 8.5% of runtime reductions. The reason is because a load or store instruction might not be executed immediately right after its effective address has been predicted by the fast address generator. Figure 15 depicts the average clock cycles that load and store instructions stay at the *Dispatch* stage before performing data accesses at *Execute* stages. *DRC* can effectively reduce the latencies between these two stages for all the SPECint2000 benchmarks. However, the latency reduction at the *Dispatch* stage can not fully converted into performance improvement due to the extra delays incurred at the *WriteBack* stage, as shown in Figure 16. As a result, latency reductions of load and store instructions are smaller than expected.

V. Conclusions

This paper has presented a fast address generator that can reduce load/store latencies of data cache accesses by eliminating effective address calculation of most load and store

instructions. Experimental results have shown that this fast address generator eliminates 74% effective address computations of load and store instructions on average for SPECint2000 benchmarks and reduces the execution times by 8.5%.

ACKNOWLEDGMENT

This research was supported in part by NSC grant NSC94-2213-E-019-008 and MOEA project 94-EC-17-A-01-S1-034.

REFERENCES

[1] T. Austin, E. Larson, and D. Emst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] Todd M. Austin, Dionisios N. Pnevmatikatos, and Gurindar S. Sohi. Streamlining data cache access with fast address calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 369–380, 1995.

[3] Todd M. Austin and Gurindar S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 82–92, 1995.

[4] Michael Bekerman, Adi Yoaz, Freddy Gabbay, Stephan Jourdan, Maxim Kalaev, and Ronny Ronen. Early load address resolution via register tracking. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 306–315, 2000.

[5] Chung-Ho Chen and Akida Wu. Microarchitecture support for improving the performance of load target prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 228–234, 1997.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[7] R.E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[8] Enric Morancho, José María Llabería, Àngel Olivé, and Marta Jiménez. One-cycle zero-offset loads. In *Proceedings of Second International Conference on Parallel and Distributed Systems (EuroPDS 1998)*, 1998.

[9] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. Reducing data cache energy consumption via cached load/store queue. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 252–257, 2003.

[10] Lu Peng, Jih-Kwon Peir, and Konrad Lai. Signature buffer: Bridging performance gap between registers and caches. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 164–175, 2004.

[11] Standard Performance Evaluation Corporation. SPEC CPU2000 v1.1, 2000.

[12] Chris Weaver. SPEC 2000 binaries. http://www.eecs.umich.edu/∼chriswea/benchmarks/spec2000.html.