3.3 Memory Traffic Reduction

The important benefit of the proposed technique is shown in Fig. 4.a and Fig. 4.b. They present the percentage reduction in memory traffic after applying the redundant store mechanism. Note that on average we have a 7% of memory traffic reduction for *CB-WA* meanwhile we have a 19% of memory traffic reduction for *WT-NWA* for a 32 KB cache.

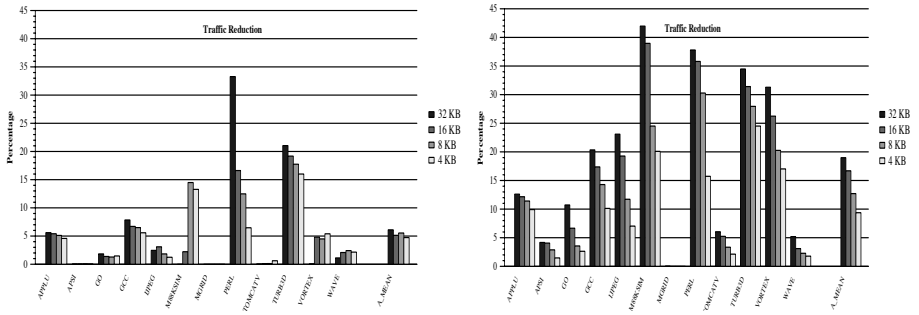

Figure 4: Reduction of Memory Traffic of **a)** *Copy Back-Write A*llocate and **b)** *Write Through-No Write Allocate*

Mainly, this difference is due to the fact that all the stores have to go to the next level of the memory hierarchy for *Write Through*, so every redundant store that can be identified results in a saving on memory traffic. On the other hand, using *Copy Back,* every redundant store reduces the frequency of the dirty bit setting, which not always results in a traffic reduction, because for instance two consecutive redundant stores mapped to the same cache line, just reduce the traffic in one block. Specially significant is the traffic reduction that we achieve using *WT-NWA*. Note that some benchmarks achieve more than 30% in traffic reduction.

4. Conclusions

This work presents an easy mechanism for reducing the memory traffic between cache and the next level of the memory hierarchy. The novel idea requires *minimal hardware support*. It is based on the particular behaviour of some writes to memory that do not change its contents and the idea can be applied to any of the current cache organizations. These particular stores are what we call redundant stores. We have shown that we can achieve a significant memory traffic reduction. On average we can achieve close to 7% for a cache with *CB-WA* and 19% with *WT-NWA*.

Acknowledgements

This work have been supported by projects CYCIT 511/98 and ESPRIT 24942, and by the grant AP96-52460503. The research described in this paper has been developed using the resources of the Center of Parallelism of Barcelona and the ECE Department of the Carnegie-Mellon University.

References

[1] D. Burger, J. R. Goodman, and A. Kägi, "Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors". *In Proc. of the 23rd Int. Symp. on Computer Architecture*, 1996.

[2] T.-F. Chen and J.-L. Baer. "A performance Study of Software and Hardware Data Prefetching Schemes". In *Proc. of the 21st Int. Symp. on Computer Architecture,* 1994

[3] J. González and A. González. "Speculative Execution via Address Prediction and Data Prefetching". *In Proc. of the 11th ACM Int. Conf. on Supercomputing,* 1997.

[4] J. R. Goodman, "Using Cache Memory to Reduce Processor Memory Traffic" *In Proc. of the 10th Int. Symp. on Computer Architecture,* 1983.

[5] N.P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of Small Fully Associative Cache and Prefetch Buffers". *In Proc. of the 17th Int. Symp. on Microarchitecture*, 1990.

[6] D.M. Tullsen, S.J. Eggers and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism". *In Proc. of the 22nd Int. Symp. on Computer Architecture*, 1995

3.1 Amount of Redundant Stores

This section analyses the behaviour of redundant stores in cache. Different cache sizes and policies have been considered. Fig. 2.a and Fig. 2.b show the percentage over all the stores that access the cache that we can qualify as redundant stores because they do not change its state. Particularly, Fig. 2.a shows the percentage of redundant stores for *CB-WA* caches and Fig. 2.b shows the percentage using *WT-NWA*.

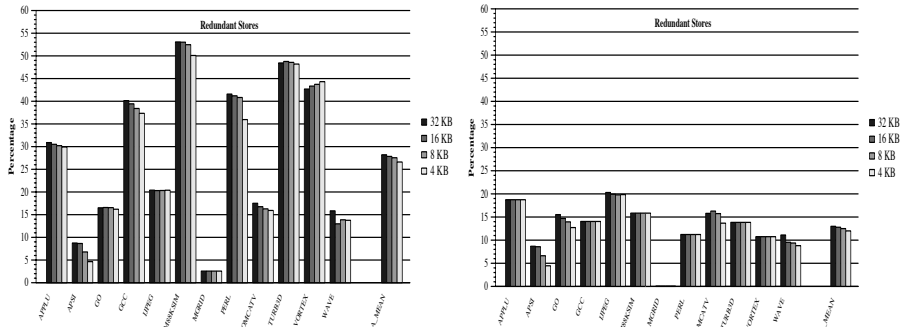

Figure 2: Percentage of Redundant Stores of **a)** *Copy Back-Write Allocate* and **b)** *Write Through-No Write Allocate*

Note that the amount of redundant stores in cache is significant in some benchmarks. On average, around 30% of the stores are redundant for a *CB-WA* cache and 15% for a WT-NWA cache. This difference is due to the fact that *CB-WA* has better *store miss ratio* than *WT-NWA*. The former has half store miss ratio than the latter.

3.2 Amount of Memory Traffic

This section analyses the traffic ratio that exists between the memory cache and its next level of the memory hierarchy. The traffic is computed as the number of bytes transmitted between memories. Fig. 3.a and Fig. 3.b show the memory traffic for each policy and different sizes. Obviously, the memory traffic decreases when we use bigger

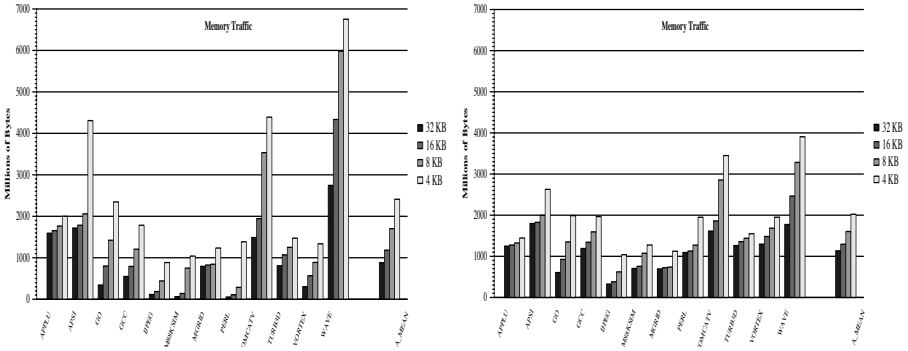

Figure 3: Memory Traffic (Millions of bytes) of **a)** *Copy Back-Write Allocate* and **b)** *Write Through-No Write Allocate*

cache sizes due to the fact that the miss ratio decreases. The memory traffic has two main sources: the hit/miss ratio and the size of the cache block. Note that on average *CB-WA* and *WT-NWA* have similar rate of memory traffic. Although *CB-WA* has better hit ratio than *WT-NWA* and intuitively it should have lower memory traffic, it has to transmit the whole cache block on a miss if the replaced block is dirty, while *WT-NWA* only has to transmit a single value per each write.

whether the buffer can send the value to the next level of the hierarchy. Here we have a reduction in memory traffic because memory write values present in cache do not have to go to the next level of the memory hierarchy if they are identified as redundant stores.

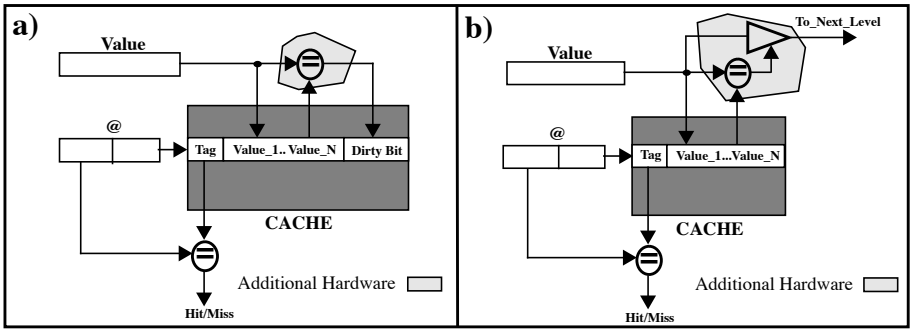

Figure 1: Redundant Store Mechanism with **a)** Copy Back **b)** Write Through

A store in a conventional cache first reads the tag and afterwards writes the new value. Our mechanism, reads simultaneously the old value and the tag and then writes the new value if necessary, so we do not increase the cache latency.

3. Experimental Framework and Performance Evaluation

We have developed a functional parameterized simulator for cache memories. Mainly, this simulator provides information about required memory bandwidth and hit/miss ratios. Different cache configurations and cache policies have been simulated. For the evaluation we have been considered a subset of the Spec95 benchmark suite. The programs have been compiled with the DEC Fortran and C compilers for a DEC AlphaStation 600 5/266 with full optimizations, and instrumented by means of the Atom tool. Each program was run with the reference input sets and statistics were collected for *1 billion* of instructions after skipping the initial part corresponding to initializations.

	COPY BACK- WRITE ALLOCATE					WRITE THROUGH- NO WRITE ALLOCATE			
	32 KB	*16 KB*	*8 KB*	*4 KB*		*32 KB*	*16 KB*	*8 KB*	*4 KB*
Applu	10.81	11.29	12.15	14.01		16.40	16.84	17.65	19.48
Apsi	12.61	13.12	15.03	29.59		14.99	15.71	18.95	35.26
Gcc	2.40	5.13	7.81	11.61		2.85	6.57	11.58	18.90
Go	3.52	5.73	10.26	17.13		7.71	9.67	12.90	17.87
Ijpeg	1.58	2.69	7.32	15.34		1.63	3.16	7.87	15.98
M88ksim	0.67	1.30	5.52	7.68		1.22	1.82	7.11	11.49
Mgrid	4.97	5.21	5.35	8.46		6.89	7.12	7.25	10.35
Perl	0.27	0.55	1.64	7.35		1.80	3.37	4.91	12.86
Tomcatv	12.06	15.98	29.13	36.26		16.07	18.76	30.87	38.87
Turb3d	5.82	7.59	8.92	10.60		5.79	7.93	10.62	13.52
Vortex	1.88	3.45	5.40	8.04		2.96	5.83	9.09	14.22
Wave	15.33	25.85	36.02	41.97		21.36	31.93	40.90	47.93
A_MEAN	**5.99**	**8.16**	**12.05**	**17.34**		**8.31**	**10.73**	**14.98**	**21.39**

Table 1: Miss Ratios of *Copy Back* with *Write Allocate* and *Write Through* with *No Write Allocate*

This section evaluates the impact on memory traffic reduction as well as the amount of redundant stores for different cache configurations.We have considered four cache sizes: 32 KB, 16 KB, 8 KB and 4 KB, and 32 byte line size. We have been simulated the following cache policies: *Copy Back with Write Allocate (*CB-WA for short*)* and *Write Through with No Write Allocate* (WT-NWA for short). *Direct mapped* caches have been assumed for all the simulations. Table 1 shows the miss ratio of every combination of the above described sizes and policies.

Reducing Memory Traffic Via Redundant Store Instructions

Carlos Molina, Antonio González, Jordi Tubella
Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya,
Jordi Girona 1-3, Edifici D6, 08034 Barcelona, Spain
e-mail:{cmolina,antonio,jordit}@ac.upc.es

Abstract. Some memory writes have the particular behaviour of not modifying memory since the value they write is equal to the value before the write.These kind of stores are what we call *Redundant Stores*. In this paper we study the behaviour of these particular stores and show that a significant saving on memory traffic between the first and second level caches can be avoided by exploiting this feature. We show that with *no* additional hardware (just a simple comparator) and without increasing the cache latency, we can achieve on average a 10% of memory traffic reduction.

1. Introduction

During the last decade, innovation and technological improvements in processor design have outpaced advances on memory design. That is the reason why current high performance processors focus on cache memory organizations, to ease the gap between processor and memory speed [4]. The problem is that many of these techniques used to tolerate growing memory latencies do so at the expense of increased memory bandwidth, and this has been shown to be progressively a greater limit to high performance [1]. The different techniques proposed such as software and hardware prefetching [2], stream buffers [5], speculative load execution [3], and multithreading [6] reduce latency related stalls, but also increase the total traffic between main memory and the processor. In this paper we propose a technique for reducing memory traffic without any special hardware requirement and without affecting the hit/miss ratio of the cache. The only hardware requirement is just a simple comparator.

2. Redundant Store Mechanism

This section describes the changes required on different cache configurations in order to take advantage of the redundant stores for memory traffic reduction. Note that the additional hardware is minimal and it is highlighted in each figure.

Copy Back: In the regular version, a store accesses cache to write its value and sets the dirty bit. We propose to first check if the current value in cache matches the value that this store is going to write. If so, there is a *Redundant Store* and the dirty bit is not set. Fig. 1.a shows how this mechanism works. Here we have a reduction in the frequency the dirty bit is set, and this consequently reduces cache miss bandwidth because fewer blocks of the cache will have to go to the next level of the memory hierarchy when there is a cache miss. Note that to exploit the benefit of redundant stores we only have to add a simple comparator that tries to detect matching values, and if so, the dirty bit is not set.

Write Through: When a store accesses the cache and there is a hit, if the value to be written is the same as the present, we have a *Redundant Store*. Once this store is identified we propose, to do not send to the next level of the memory hierarchy. Fig. 1.b shows how this mechanism works. Note that the output of the comparator decides