Emulating Write-Allocate on a No-Write-Allocate Cache

Andrew W. Appel
Princeton University
appel@princeton.edu

CS-TR-459-94
Princeton University
June 20, 1994

Abstract

A cache write-miss policy of *write-allocate* leads to better performance for many programs than a policy of *no-write-allocate*.

I show a compiling technique to emulate a write-allocate policy even on a no-write-allocate machine, if that machine has non-blocking cache read misses (e.g. DEC Alpha 21064).

The trick is to pre-fetch garbage *before* the write, so that the cache line will be allocated and the write will hit the cache. Measurements show a factor-of-1.2 improvement in execution time.

Jouppi [5] analyzed cache write-miss policies. Some machines allocate a cache line containing newly-written word (*write-allocate*); others bypass the cache on a write miss and send the word directly to the next level of the memory hierarchy (*no-write-allocate, write-around*). Write-allocate machines may then fetch the rest of the cache line (*fetch-on-write*) or may mark the rest of the line invalid until overwritten (*write-validate*). Jouppi found that "Write-validate and write-around always outperform fetch-on-write. In general write-validate outperforms write-around since data just written is more likely to be accessed soon again than data read previously."

Diwan, Tarditi, and Moss [3] show that garbage-collected programs run *much* faster on write-validate caches than on write-around, since *all* "initializing writes" are misses, and almost all such data will be read back soon (incurring a read-miss on a write-around cache).

The technique described in this paper is useful whenever:

1. The cache is write-around;
2. Read misses do not stall (until the target register is subsequently used);
3. A particular write will miss the cache;
4. The data written is likely to be read back soon;
5. The (approximate) address written is predictable tens of instructions in advance.

Conditions 1 and 2 are true of some modern machines; conditions 3,4,5 are true of many programs, in particular those programs that use compacting garbage collection.

Under these conditions, deliberately causing a read miss will improve performance! Several cycles in advance of writing (and missing) address A, the compiler should insert a read from address A. The read will force the allocation of a line in the cache, so that subsequent writes and reads will hit.

t	Old	New
0		read miss (no stall)
20	write miss (no stall)	write hit (no stall)
70	read miss (no stall)	read hit (no stall)
73	use of value **stall!**	use of value (no stall)

The DEC Alpha 21064 has an 8 kbyte primary cache with 32-byte cache lines and a *write-around* miss policy [2]. Just as predicted by Diwan *et al.*, the performance of Standard ML of New Jersey [1] on the 21064 is terrible. Almost every write is sequential (in order of increasing addresses); these writes all miss; and the cache line is almost always read back fairly soon (before 2k subsequent 32-bit writes). Thus, soon after every write-miss there is a read-miss on that line, followed by a stall when the target register of the read is used in a subsequent operation. With write-allocate, this read would have hit the cache.

I implemented the "garbage-prefetch" technique in the SML/NJ compiler. Most writes issued by the compiler are easy to analyze statically: a dedicated register A points to the free region of the garbage-collected heap. To allocate and initialize a record

Program	Normal			Garbage-prefetch			Speedup	Speedup
	user	sys	real	user	sys	real	(user+sys)	(real)
Life	11.52	0.02	11.70	8.20	0.02	8.32	1.40	1.41
Knuth-Bendix	6.05	0.10	7.38	4.87	0.11	6.12	1.23	1.21
Lex	8.42	3.26	12.62	6.53	3.29	10.72	1.19	1.12
Yacc	2.54	0.41	3.75	2.35	0.43	3.83	1.06	0.97
Average							1.22	1.18

Figure 1: Performance of garbage-prefetch technique

Benchmarks are described by Appel [1]. Average of three runs of each benchmark are shown. Variation in run times of each benchmark was approximately 10% in each component.

on the heap, several writes are done at constant offsets from A, then A is incremented by the size of the record (increments within one basic block are coalesced). Approximately 99% of all writes are these sequential writes to initialize new objects.

At every point where A is incremented by n bytes, instructions are inserted to read words at $A+M, A+M+B, A+M+2B, ..., A+M+kB$, where

- $M = 96$ is the size of two or three cache lines (so the read-miss will be handled before M more bytes are written);
- $B = 32$ is the size of one cache line;
- $k = \lfloor (n-1)/B \rfloor$ is the number of cache lines needed to allocate.

The extra read instructions target a real machine register, but should not increase register pressure significantly because the live range of each value is empty.

Because most basic blocks allocate less than one cache line, usually only one extra read will be performed per block; this should not flood the memory system with too many extra cache misses.

This took about 6 lines of code to implement, and improves performance by a factor of 1.2 on a variety of benchmark programs on the DEC Alpha 21064 (DEC 3000/400); see figure 1.

This result is not definitive, since our instruction scheduler for this architecture is still very immature: garbage-prefetching improves CPI from approximately 4 to approximately 3, on a dual-issue machine whose peak performance is 0.5 CPI.

Read misses on the 21064 stall the "A-box"—which performs loads, stores, and most branches—for about 5 cycles. Other instructions are not stalled. Our scheduler does not (yet) handle resource conflicts; better scheduling of the garbage-prefetches (which have almost no data dependencies) might improve performance. In any case, the 21064 only barely satisfies condition 2. On write-around machines that can handle cache misses without blocking the memory interface, garbage-prefetch should be even more attractive.

Note: Some machines (e.g. IBM R/S 6000 [4]) have a cache-line-zero instruction, whose purpose is to allocate a cache line without reading soon-to-be overwritten data. On such a machine, zeroing line $A+M$ (etc.) whenever A is incremented should improve performance. (Peng and Sohi suggest zeroing the line at A for this purpose [6], but this requires that A always be aligned on cache-line boundaries.) The garbage-prefetch technique is just an expensive emulation of the cache-line-allocate instruction.

Acknowledgements: Martin Carlisle, Ken Cline, Amer Diwan, Lal George, John Reppy, Anne Rogers, and David Tarditi constructively criticized early drafts of this paper.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Digital Equipment Corp., Maynard, MA. *DECchip(tm) 21064-AA Microprocessor Hardware Reference Manual*, first edition, October 1992.

[3] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–14. ACM Press, 1994.

[4] William R. Hardell, Dwain A. Hicks, Lawrence C. Howell, Warren E. Maule, Robert Montoye, and David P. Tuttle. Data cache and storage control units. In *IBM RISC System/6000 Technology*, pages 44–50. IBM, 1990.

[5] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201. ACM Press, May 1993.

[6] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.