Reducing Garbage Collector Cache Misses

Hans-J. Boehm
Hewlett-Packard Laboratories
1501 Page Mill Rd., MS 1U-17
Palo Alto, CA 94304-1126
Hans_Boehm@hp.com

ABSTRACT

Cache misses are currently a major factor in the cost of garbage collection, and we expect them to dominate in the future. Traditional garbage collection algorithms exhibit relatively little temporal locality; each live object in the heap is likely to be touched exactly once during each garbage collection. We measure two techniques for dealing with this issue: prefetch-on-grey, and lazy sweeping. The first of these is new in this context. Lazy sweeping has been in common use for a decade. It was introduced as a mechanism for reducing paging and pause times; we argue that it is also crucial for eliminating cache misses during the sweep phase.

Our measurements are obtained in the context of a non-moving garbage collector. Fully copying garbage collection inherently requires more traffic through the cache, and thus probably also stands to benefit substantially from something like the prefetch-on-grey technique. Generational garbage collection may reduce the benefit of these techniques for some applications, but experiments with a non-moving generational collector suggest that they remain quite useful.

1. INTRODUCTION

Tracing garbage collectors [9] conceptually operate in two phases:

1. Identify and mark reachable objects. (The *mark* phase.)
2. Reclaim unmarked objects. (The *sweep* phase.)

The mark phase normally requires a traversal of all reachable objects in the heap, often reading the contents of each object exactly once. Thus much of the heap will have to be read into the cache, the reads are likely to miss, and the new contents of the cache are unlikely to be used more than once.

For many garbage collectors, the sweep phase requires essentially no work, and it is typically omitted from the algorithm description. A pure copying collector may just update a few pointers to cause the allocator to reuse the previous heap space. Baker's treadmill collector [1] only has to update a few pointers to lists of objects. A non-moving garbage collector may simply require the allocator to search for an unmarked object. In all of these cases, the total cost of the "sweep phase" is bounded by a very small constant, which is independent of the heap size.

In contrast, traditional mark-sweep collectors require a full heap traversal, in which unmarked objects are restored to free lists. And indeed, there are advantages to explicitly building free lists from unmarked objects:

- Allocation is fast. The allocator typically needs to locate the appropriate free list for the requested object size, remove the first entry and return a pointer to it. This requires a small number of memory references, and very limited access to global collector data structures.
- All heap scanning for unmarked objects is performed in a tight loop. Pointers to relevant data structures can be moved into registers once, outside the loop. If mark bits for multiple objects are packed into a word, multiple mark bits can be retrieved with a single load instruction. The scanning loop tends to access memory sequentially, an access pattern for which the memory system has been tuned.
- Little additional space overhead is required. Unlike the Baker Treadmill collector, no additional pointers are associated with live objects. We do not need to copy any section of the heap. We need only one bit per object overhead for marking. (This may be partially offset by fragmentation issues.)

As a result, many garbage collectors do use a sweep phase that is somewhat separated from both the mark phase and the allocator. Such a sweep phase naturally has cache behavior similar to the mark phase: Each object is touched exactly once. Unlike the mark phase, accesses tend to be more sequential, since mark bits can often be read many at a time, most memory accesses are likely to be write accesses, and the objects touched are exactly those not examined by the marker. The last point makes it even less likely that any of the cache contents left by the marker will be reused.

We argue that, as a result of their memory access patterns, the performance of both the mark phase and, if there is one, the sweep phase is largely determined by memory-system issues. Thus garbage collection algorithms should be compared primarily on the basis of the generated traffic through the memory hierarchy.

Older work on garbage collection often attempted to provide reasonable performance in the presence of paging [11, 2]. More modern implementations appear to have largely given up on this goal. Java virtual machines generally perform unacceptably in the presence of paging. Economics have made this state of affairs acceptable. Rapid increase of memory sizes combined with much smaller improvements in disk performance have made it nearly unavoidable.

At the same time, cache considerations have become much more important for garbage collectors. Cache miss penalties are still much less than those associated with page faults, but a complete cache miss often incurs a penalty corresponding to hundreds of instruction executions.[1]

Typically, even the largest cache level is still appreciably smaller than the heap. Thus the above observations make cache misses in the collector essentially unavoidable.

Even generational collectors often encounter similar problems. The youngest generation must be large enough to amortize the cost of scanning the root set (i.e. the non-heap-allocated pointer variables) and the remembered set (i.e. pointers from other generations). Thus it most commonly does not fit into any level of the cache.

The mutator will normally touch the entire youngest generation during a collection cycle. If the cache is too small to hold the entire generation, the contents left in the cache at the beginning of the mark phase are likely to include recently allocated dead objects, but not all live objects allocated near the beginning of the collection cycle. Thus the marker is still likely to encounter cache misses, and it is likely to evict objects subsequently needed for allocation, if they haven't already been evicted.

A generational collector will also still need to mark (and possibly sweep) the entire heap occasionally. For applications that primarily build and drop large data structures, this is likely to happen frequently.

2. RELATED WORK

Much of the prior work on the relationship between cache performance and garbage collection has concentrated on using a copying or other compacting collector to improve the locality of the mutator (client) [5, 10]. We believe this work is orthogonal to ours and could largely be combined with it.

We concentrate exclusively on the performance of the collector itself. Neither of the techniques discussed here should significantly impact the performance of the mutator.

[1] Most current machines have main memory latencies on the order of 200nsecs, which is typically 100-200 cycle times, and may correspond to several times that many instructions. Many such measurements, mostly obtained using Larry McVoy's "lmbench", are available on the web.

There has been some work on automatically inserting prefetch instructions in pointer-based code. The prefetch-on-grey technique described here is very similar to the "Greedy Prefetch" strategy for recursive data structure traversals from [4], though we use some refinements, and our garbage collector mark loop would not have been recognized as a recursive data structure traversal by a compiler.

An alternate approach to reducing sweep time in sparse heaps is discussed in [6]. It ignores the cache benefits of lazy sweeping, and thus helped to inspire some of the measurements presented here.

3. ASSUMPTIONS

Our discussion will take place in the context of a mark-sweep collector.

Our collector[2] segregates objects by size. Each page in the heap (or "heap block") contains objects of a single size. We do not believe that this is a necessary assumption for any of the techniques described here, but it makes their description easier.

We will assume that the collector maintains its data structures outside the user-visible heap. In particular, mark bits are stored separately, not in or near the objects. In our case, the mark bits are part of a descriptor associated with each page in the heap. We reserve a mark bit for each word in the page, though only those corresponding to the beginning of an object are used.

The separation of mark bits from the actual heap objects is essential for our discussion and results on lazy sweeping. Although this can increase the number of instructions executed in the mark phase, it has some major advantages:

1. The mark phase does not need to examine pointer-free objects at all. They do not need to be brought into the cache. If pointer-free objects are segregated (as they are in our environment), pages containing pointer-free objects can remain paged out during a garbage collection. In our experience, this easily justifies the technique by itself.
2. Sequences of small unreachable objects can be recognized and reclaimed efficiently as a group, by examining a sequence of mark bits with a single instruction. In our environment, we take advantage of this if an entire heap block (roughly a page) turns out to be empty. In practice, this is a frequent occurrence.

The collector data structures are a small fraction of the size of the heap. In our informal discussions we will ignore their impact. In fact, and in our measurements, they do compete with the user heap for the cache.

4. PREFETCH ON GREY

Abstractly, we can describe the mark phase of a garbage collector as follows [7]. Each object has an associated color: White objects have not yet been visited, grey objects have

[2] See **http://www.hpl.hp.com/personal/Hans_Boehm/gc**.

been visited but pointers inside the object have not yet been examined, and black objects have both been visited and contained pointers have been followed.

The mark algorithm can then be described as:

Ensure that all objects are white.
Grey all objects pointed to by a root.
while there is a grey object g
    blacken g
    For each pointer p **in** g
        if p **points to a white object**
            grey that object.

In reality, we keep a table containing only one mark bit per object. A mark bit value of zero corresponds to a white object; a value of one indicates a grey or black object. In addition, a separate stack contains the addresses of all grey objects.

When an object is greyed its mark bit is set, and it is pushed onto the mark stack. Blackening an object corresponds to removing it from the mark stack.

Empirically a significant fraction of the time devoted to this algorithm is spent retrieving the first pointer p from each grey object. On a 500MHz Pentium III, we typically see the marker dominate the time spent in the garbage collector, and about a third of the execution time of the marker is accounted for by the load instruction(s) performing the initial load of a candidate pointer p from an object.[3] This should not be surprising, since this is the first reference to the object itself by the marker, and typically the first reference to the cache line since at least the start of the mark phase.

We expect the percentage of mark time accounted for by these misses to increase on future machines, as processor cycle times continue to decrease faster than memory latencies.

Most modern processors provide instructions that allow data to be "prefetched" into the cache in anticipation of future use. This can avoid much of the delay associated with retrieving an object not in the cache, provided it is possible to predict the object reference sufficiently far ahead of time. If the object can only be found in main memory, this may require that the prefetch instruction is issued hundreds of instructions ahead of the actual object reference, since the cache miss will probably require on the order of 100 machine cycles.

Most commonly "prefetch" instructions are introduced by a compiler optimizing numerical code (cf. [3]). Array references can often be automatically predicted. In non-numerical "pointer-chasing" code, this is less feasible (but see for example [4]).

[3]This was measured by profiling the program using a PC-sampling technique with single instruction resolution. A tiny utility to generate such profiles is available from **http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html**.

Nonetheless, in this case, we can manually predict object references quite far ahead of time. As soon as we grey an object, i.e. push it onto the mark stack, we immediately prefetch the first cache line in that object. In our implementation, we in fact prefetch the object slightly before that, specifically as soon as we notice that p is likely to be a pointer, before verifying that it is indeed a pointer and was not previously marked.

We use two other techniques, which slightly improve performance:

1. We minimize the number of cases in which an object that has just been greyed is immediately examined. This is done by permuting the code in the marker slightly, so that the last pointer to be pushed on the mark stack is prefetched first. For reasons related to optimizer deficiencies and register pressure in the mark loop, this turns out to improve matters slightly even without the prefetch.
2. We also linearly prefetch a few cache lines ahead as we are scanning an object. This helps with very large objects. It probably also often prefetches other relevant objects in the data structure we are scanning.

The first technique ensures that for every pointer p inside an object g, the prefetch operation on p is separated from any dereference of p by most of the pointer validity and mark bit checking code for the pointers contained in g. Thus, unlike the cases studied in [4], the prefetch instruction should be very helpful even for the first pointer followed.

This is somewhat less true for a nonconservative collector for which pointer validity checking is typically limited to checking a bit in an object descriptor, and checking the pointer value against `null`.

5. LAZY SWEEPING

Traditional mark-sweep collectors first mark all reachable objects, and then perform a sweep over the entire heap, adding each unmarked object to a free list. Since free list pointers are typically maintained inside the objects themselves, this often involves accessing every unmarked object. Thus the sweep phase can significantly add to garbage collector pause times. Furthermore, it needs to both page in any pages containing unreachable objects, and move at least part of each such object through the cache.

If the amount of reclaimed memory exceeds the size of the cache, it is unlikely that many of the objects brought into the cache by the sweep phase will be reused before they are evicted from the cache by the allocator or mutator. Thus the allocator is again likely to miss the cache when the object is finally allocated.

Hughes [8] first observed that collector pause times can be reduced by performing the sweep incrementally, that is, deferring it until allocation time. (Hughes suggests having the allocator examine mark bits directly, something that would probably make the allocator too expensive in our context.)

Others have suggested different variations on this idea (cf.

[9, 11, 2]), usually in part to improve paging performance. If a section of memory is swept just before the reclaimed memory is reallocated, we may still encounter a page fault while sweeping. But when the allocator needs that same region of memory, it is nearly guaranteed to be resident.

The same observation applies to the cache. By ensuring that a cache line is reallocated very shortly after it is swept, we replace two cache misses, in the sweep phase and allocator, by a single miss.

We measure the variant of lazy sweeping that has been implemented in our collector for a number of years. After completing the mark phase, the collector scans the mark bit table for each page in the heap. If no mark bits are set in the page, the entire page is added to a pool of free pages *without touching it*. If some mark bits are set, the page is added to a queue of pages waiting to be swept. For present purposes, it is safe to assume that there is one such queue for each object size. (There are actually several, depending on other object attributes.)

The allocator normally maps the requested object size to an appropriate object free list. (This involves a table lookup, so that similar sizes an be rounded to a single size class in the same step.) It then removes the first object from the free list and returns it. If the free list is empty, it sweeps additional pages dedicated to objects of the right size, until it finds one containing some available objects. (A heuristic is used to avoid scanning or touching most pages that are "nearly full".) In the absence of explicit deallocation and incremental collection, no free list ever contains objects from more than one page.

In order to disable lazy sweeping, we modified the collector to optionally sweep all pages immediately after enqueueing them.

6. MEASUREMENTS

We report the impact of the above two techniques on Pentium III/500 execution times of a number of benchmarks. To get some calibration on variation with architecture, we also remeasured a few of the benchmarks on an HP PA-8000-based machine. Currently all benchmarks except one are C programs, and are executed with our collector scanning the heap conservatively.

For reasons discussed in the appendix, a Pentium III, although it does provide prefetch instructions, does not appear to gain maximum benefit from our prefetch-on-grey technique. Preliminary results on an Intel Itanium machine are even more encouraging.

We use the following benchmarks:

gc_bench_java A fairly widely used artificial Java garbage collection benchmark.[4] It repeatedly allocates and drops complete binary trees of various heights. It is probably more dependent on garbage collector performance than any real application. Consecutively allocated objects tend to be dropped at exactly the same time, which is representative of some real applications, and unrepresentative of others. Due to this behavior, our collector tends to reclaim entire pages, and spend almost no time in the sweep phase.

The fact that objects have exactly zero or two outgoing pointers, and most objects are reachable by exactly one path, tends to help our prefetch-on-grey implementation. Objects with a single outgoing pointer provide only a smaller amount of latency between the prefetch and dereference of the outgoing pointer. Multiple paths to the same object are likely to result in redundant prefetches.

The benchmark was compiled with the gcj static Java compiler[5], and garbage collected with our collector. Unlike the C benchmarks, the collector had object layout information for heap objects. This information is encoded in GC descriptors, which can be retrieved from the object's method table. The runtime is currently about 45% higher than the corresponding C program, mostly because the allocation sequence acquires a spin lock, and has not been as well-tuned as its C counterpart.

gc_bench A C translation of the above benchmark, also available from the above location.

holes_gc_bench The above C benchmark, but we allocate and drop an extra tree node between each pair of retained nodes. This causes the collector to spend large amounts of time in the sweep phase.

ptc This is one of the programs in the Zorn memory allocation benchmark suite. It and Ghostscript appear to be the only two benchmarks in the suite with a heap size that exceeds the size of the cache on the HP machine. It translates Pascal to C. It was run on the largest input. The program was written for `malloc/free` memory management, but it builds a single large syntax tree data structure, which is retained until program exit. A number of garbage collections are run during execution, though they could all be avoided by immediately setting the heap size to its final value.

ghostscript This is again the version from the Zorn benchmark suite, run on the largest provided input. As with ptc, we let the garbage collector grow the heap using its normal heuristics. This version bypasses the original Ghostscript custom memory management, but is not well-tuned for a garbage collector.

incremental_ghostscript As above, but we let the collector run in its incremental and generational mode. This is similar to the scheme described in [BoeDemSch91], but the collector is run incrementally during allocation calls instead of in a separate thread. The primary difference is that most collections do not reset the mark bit state, and hence do not attempt to

[4]The benchmark was written by John Ellis, Pete Kovac, and the author. It is available from **http://www.hpl.hp.com /personal/Hans_Boehm/gc/gc_bench/**.

[5]See **http://sourceware.cygnus.com/java**. The version we used differed from the (7/24/00) released versions in that it used code supplied by Tom Tromey, Bryce McKinley, and the author to avoid interpreting Java class objects in the garbage collector. We expect our version to become the standard one.

collect objects that survived the preceding collection. True incremental mode is triggered only when the collection time exceeds a 50msec threshold.

large_ghostscript This is ghostscript with a non-incremental collector, but we added code to immediately expand the heap to 20MB, nearly 10 times larger than the maximal live data size.

Table 1 presents the Pentium performance results relative to the "normal" version of the collector, which sweeps lazily, but does not prefetch. For each benchmark, we give the approximate fraction of time spent marking when using the "normal" collector, the approximate fraction of the time spent sweeping, the slowdown from eager sweeping, and the speedup from the addition of prefetching.

All percentages are fractions of the entire "normal" execution time of that benchmark, not fractions of garbage collection time.

Table 2 contains some of the analogous measurements on an HP PA-8000 machine running at 180MHz.

We draw several conclusions from these measurements:

- Sweeping eagerly instead of lazily usually slows down the sweep phase by at least 50%. This is entirely a cache effect.
- For both gc_bench versions, prefetch-on-grey appears to eliminate most of the cache miss overhead in the marker. For data structures that look less like complete binary trees, it is less effective, but still seems to eliminate at least a third of the miss overhead (which is about a third of the mark cost.)
- Sweep times are rarely an issue for our collector, though this is surprisingly architecture dependent. Generally, collector performance is determined by the marker.
- The incremental collector essentially requires lazy sweeping, since otherwise the entire heap is swept, even for minor collections. (This is not a new insight.)

Both prefetch-on-grey and lazy sweeping are very cheap to implement in many contexts, and should clearly be incorporated into future garbage collectors.

It is worth pointing out that in our experiments, the execution time always varies inversely with the number of instructions executed. Measurements with the PCL performance counter library[6] show that lazy sweeping executes slightly more instructions in our implementation, but is almost uniformly no slower. This is more than made up for by a reduction in cache misses. [7] Similarly prefetching adds a small number of instructions.

[6]See **http://www.fz-juelich.de/zam/PCL/**.

[7]In the case of holes_gc_bench, we measured about 8% reduction in the both L1 and L2 cache miss rates on a 300MHz Pentium II. The corresponding cache miss measurements for ghostscript also paralleled the reported execution time measurements.

7. ACKNOWLEDGEMENTS
I would like to thank the anonymous ISMM referees and Brian Lynn for their many detailed comments.

8. REFERENCES
[1] H. G. Baker. The treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3), March 1992.

[2] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 26, 6*, pages 157–164. ACM, June 1991.

[3] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52. ACM, April 1991.

[4] L. Chi-keung and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference for Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices 31, 9*, pages 222–233, September 1996.

[5] T. M. Chilimbi and J. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management*, pages 37–48. ACM, 1998.

[6] Y. C. Chung, S.-M. Moon, K. Ebcioglu, and D. Sahlin. Reducing sweep time for a nearly empty heap. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–387. ACM, January 2000.

[7] E. W. Dikstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[8] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1084, Novemeber 1982.

[9] R. Jones and R. Lins. *Garbage Colection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, New York, 1996.

[10] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective 'static graph' reorganization to improve locality in garbage collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 26, 6*, pages 177–191, JUNE 1991.

[11] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, June 1990.

Table 1: Pentium II/500 Relative Performance

Benchmark	Mark Time	Sweep Time	Eager Sweep Slowdown	Prefetch Speedup
gc_bench_java	39%	3%	0%	11%
gc_bench	49%	3%	0%	13%
holes_gc_bench	57%	12%	7%	17%
ptc	27%	0%	0%	4%
ghostscript	44%	5%	5%	5%
incremental_ghostscript	39%	9%	17%	4%
large_ghostscript	8%	6%	3%	1%

Table 2: HP PA-RISC Relative Performance

Benchmark	Mark Time	Sweep Time	Eager Sweep Slowdown	Prefetch Speedup
gc_bench	45%	3%	-1%	11%
holes_gc_bench	40%	36%	34%	9%
ghostscript	26%	4%	3%	8%

APPENDIX

A. MEASUREMENT DETAILS

The Pentium measurements here were obtained on a dual processor 500MHz Pentium III machine with a 100MHz bus and a 512KB L2 cache running RedHat 6.1 Linux. None of the measurements here involved more than one processor. The machine had more than enough physical memory (> 300MB) to preclude paging, and to allow the executable itself to be cached in memory.

Pentium benchmarks were compiled with gcc (egcs 2.91) -O2. We used -O on the PA-RISC machine.

HP PA-RISC measurements were obtained on a single processor 180 MHz HP PA-8000 system running HP/UX 11. The machine has single-level I and D caches of 1 MB each.

The Zorn memory allocation benchmarks are available from **ftp://ftp.cs.colorado.edu/pub/misc/malloc-benchmarks**. The benchmarks we used needed minor modification to allow compilation with modern C compilers. No substantive changes were made, except as described above. The only change required to run Ghostscript with incremental garbage collection was a call to `GC_enable_incremental()`.

The measurements were obtained with collector versions equivalent in performance to the released version 5.1 of the collector.

Most program execution times were in the 3 to 25 second range and repeatable with significantly less than 1% variation on the Pentium III, and up to about 2% variation on the PA-RISC machine. Ptc unfortunately ran for only about 3/4 of a second, even with the largest input.

Execution time measurements are averages across 5 runs. They include user and system mode time.

The percentage of time spent in the mark and sweep phases was obtained with gprof and, in a few cases, verified with other profiling tools. It is probably less accurate than the speedup and slowdown numbers.

The division between sweep and allocation time is unavoidably rather arbitrary. The sweep time we report here includes the time to detect completely empty pages, and to scan partially full pages. It excludes the time used to initialize known empty pages in preparation for allocation, even though that is often larger than sweep times. (Roughly 24% of gc_bench time on the Pentium III is spent clearing empty blocks of memory and writing an initial set of free list pointers into it.) Such initialization must be performed on demand, even in the absence of lazy sweeping, since it depends on the object size assigned to that page, and thus on requested object sizes. A collector that did not detect empty regions in this way would benefit much more from lazy sweeping than we show here.

We did not directly attempt to control the heap size, which can drastically affect garbage collection times. But we have been careful to compare only executions that resulted in very similar heap sizes and GC frequencies.

Prefetch instructions were inserted using gcc's inline assembly code facilities and a roughly comparable facility in the HP compiler. Although this affected the surrounding code somewhat, it did not seem to worsen it appreciably.

Unfortunately, there is no uniform prefetch instruction for all X86 compatible processors. Intel introduced a family of prefetch instructions with the Pentium III. AMD introduced a different and incompatible set of prefetch instructions with the AMD K6-2 at roughly the same time. The Pentium III instructions appear to be ignored on earlier Intel processors. The AMD instructions appear to trap on those processors. We did not have the chance to experiment on AMD-based machines.

Our experiments suggest that on a Pentium III, it is not beneficial to prefetch locations that are about to be written sequentially. Thus we did not attempt to prefetch in the sweep phase.

We initially used the `prefetcht0` instruction for the Pentium III to prefetch during the mark phase. This prefetches directly into all cache levels, and was extremely effective at eliminating the cache misses we has observed in instruction level profiles of the marker. However, it appeared to slow down the remainder of the mark loop, presumably because it interfered with the remainder of the memory traffic. Unlike other processors, the X86 version of the mark loop contains many register spills and the like, at least when compiled with gcc.

The measurements reported here use the `prefetchnta` instruction, which suggests to the hardware that subsequent accesses will not have temporal locality. We found this to be significantly less effective at reducing the original cache misses, but it appeared to introduce no new overhead. Overall results were very slightly better than with `prefetcht0`.