Faster Garbage Collection Using Prefetching

John van Groningen

University of Nijmegen, Department of Computer Science, Toernooiveld 1, 6525 ED Nijmegen, the Netherlands
johnvg@cs.kun.nl

Abstract. Mark-scan garbage collectors and many compacting garbage collectors use a mark phase. We show that for large heaps the speed of such a mark phase can be improved by adding a FIFO queue with nodes that are prefetched with prefetch instructions. This reduces the cost of the high main memory access time.
We implemented this algorithm and measured a speedup between 24 and 81 percent using AMD Athlon processors, between 10 and 26 percent using an AMD Opteron processor and about 20 percent using a Motorola G4 processor.

1 Introduction

Main memory access time has decreased much slower than the perfomance increase of microprocessors over the last decades [9]. Consequently a memory access now takes several hundred processor clock cycles. To reduce the cost of this latency, processors use non-blocking caches, out of order and speculative execution, write buffers and hardware prefetching [7, 3]. However, this is not sufficient to remove the memory access cost for many algorithms.

Garbage collection is such an algorithm. The performance of garbage collectors has increased slower than the performance of the program (mutator) with increased processor speed. For example, the garbage collection cost to recompile the compiler of the functional langauge Clean with itself using a 20 Mb heap has increased from 33 % of the execution time on an old computer to 47 % on a modern computer (see table 1), while the garbage collection overhead increased from 48 % to 87 %. This cost can easily be reduced to less than 10 percent by using a much larger heap, but so much extra memory is not always available.

In this paper we will show how the cost of the mark phase of a garbage collector can be reduced significantly, by adding a queue to the algorithm with nodes that are being prefetched. We will start by describing the garbage collectors mark algorithm and then examine the memory performance of modern computers, and use this information to improve the garbage collection algorithm. Finally we will measure the performance of the improved algorithm and conclude.

2 The Mark Phase of the Garbage Collector

Two garbage collectors are available in the implementation of the functional programming language Clean: a combination of a copying collector with a com-

Table 1. Garbage collection cost when compiling the Clean 2.1 compiler with itself using a 20 Mb heap for several computers

Computer (processor clock speed (Mhz), processor, chipset, memory)	Execute (s)	Garbage collect (s)	Total (s)	GC overhead (% of execute)	GC cost (% of total)
166, AMD K6, Intel, SDRAM 66	190.88	92.02	282.90	48	33
350, AMD K6, ALI, SDRAM 100	104.08	53.58	157.66	51	34
867, G4, Motorola, DDR266	39.08	21.60	60.69	55	36
1400, Athlon, AMD, DDR266	26.72	20.81	47.53	78	44
1533, AthlonXP, VIA, DDR333	20.68	17.50	38.17	85	46
1800, AthlonXP, Nvidia, DDR266 dual	16.14	12.99	29.13	81	45
2000, Opteron, DDR400 dual	10.38	9.04	19.42	87	47

pacting mark-scan collector, and a combination of a (non moving) mark-scan collector and the compacting mark-scan collector. These collectors are described in [5]. We will use the latter collector, because it performs better for the programs we are insterested in. We will first focus on the mark phase of the (non-moving) mark-scan collector, because this is the most expensive phase of the garbage collector. Later in the paper we will also look at the mark phase of the compacting collector.

The mark-scan collector of Clean is a tracing garbage collector, that traverses all nodes of the heap that are still in use. A bit vector with a bit for each word in the heap is used to store the set of used nodes, by marking the bits during the traversal. Scanning the heap is done lazily. When memory needs to be allocated during execution of the program, the next available (large enough) block of memory is searched in the bit vector, and this block is used. Usually this block is larger than the requested size, and the size and position of the rest of the block is saved in registers or variables, which are used to quickly perform future memory allocation requests.

The most expensive phase is the marking phase. It uses a stack on which adresses of nodes that still need to be marked are pushed. It is implemented in the following way, using C syntax:

```
for (stack_p = begin_stack; stack_p<end_stack; ++stack_p){
    node = *stack_p;
    while (true){
        while (! marked (node)){
            mark (node);
            n_pointers = n_pointers_in_node (node);
            if (n_pointers==0)
                break;
            while (n_pointers>1)
                push (argument_of_node (node,--n_pointers));
            node = argument_of_node (node,0);
        }
```

```
        if (stack_empty())
            break;
        node = pop();
    }
}
```

If the stack becomes too large (more than a few kilobytes), a slower marking algorithm that doesn't require additional memory, by using pointer reversal, is used to mark that part of the heap.

If the amount of memory that has to be traversed is larger than the size of the cache(s), the first access to a node usually causes a cache miss, and the processor spends most of its time waiting because of the high latency of memory access. We used the AMD CodeAnalyst processor simulator to confirm this.

3 Memory Performance

The performance problem with the marking algorithm is caused because the address of the next node to be marked is stored in the node currently being marked. Therefore to continue with the next node, the processor has to wait until the memory subsystem has retrieved the contents of the current node.

We measured the cost of chasing pointers in such a way on several different computers. We did this by creating a linked list of random addresses in memory, using only the first word of a cache line for those pointers, and measured how fast such a list could be traversed. Because memory accesses are faster when the same page is used as in the previous access (page hit), we did the same for a random list that fits in a memory page. The results are in table 2.

On the faster computers the memory latency is more than 200 processor clock cycles for a page miss and more than 140 for a page hit. The Opteron processor is faster because of its integrated memory controllor, but a memory access still takes 159 (page miss) or 103 (page hit) clock cycles.

Table 2. Time to load a cache line from memory

Computer (processor clock speed (Mhz), processor, chipset, memory)	Page miss (ns)	Page hit (ns)	Page miss penalty (miss/hit)	Page miss (clock cycles)	Page hit (clock cycles)
166, AMD K6, Intel, SDRAM 66	369	251	1.47	61	42
350, AMD K6, ALI, SDRAM 100	261	145	1.81	92	51
867, G4, Motorola, DDR266	158	69	2.28	137	60
1400, Athlon, AMD, DDR266	180	156	1.15	251	219
1533 Mhz AthlonXP, VIA, DDR333	191	113	1.70	294	173
1800, AthlonXP, Nvidia, DDR266 dual	118	80	1.47	213	145
2000, Opteron, DDR400 dual	79	51	1.54	159	103

Most of the nodes that have to marked are small. Only about 30 instructions are required to mark a small node like a list constructor node, but it often takes more than a hundred clock cycles to load the node from memory.

However, modern computers can perform multiple memory operations in parallel using speculative loads, busses that allow requests to be handled and responses delivered out of order, and (DDR) SDRAM with four banks, that can be accessed in an interleaved fashion [3].

To measure the benefit of accessing memory concurrently we modified our linked list traversal program. Instead of one list, we now traverse two random lists almost in parallel, by alternating accesses to the two lists. The results are presented in table 3.

Table 3. Time to load a cache line from memory, loading 2 cache lines at a time

Computer (processor clock speed (Mhz), processor, chipset, memory)	Page miss (ns)	Page hit (ns)	Page penalty (miss / hit)	Page miss (clock cycles)	Page hit (clock cycles)
166, AMD K6, Intel, SDRAM 66	369	251	1.47	61	42
350, AMD K6, ALI, SDRAM 100	261	144	1.81	91	50
867, G4, Motorola, DDR266	144	46	3.12	125	40
1400, Athlon, AMD, DDR266	146	93	1.57	205	131
1533, AthlonXP, VIA, DDR333	98	65	1.52	151	99
1800, AthlonXP, Nvidia, DDR266 dual	65	50	1.31	117	90
2000, Opteron, DDR400 dual	42	29	1.45	83	57

On the faster computers the time per memory access is about 1.7 times lower, because the accesses to both lists overlap. The cost of a memory access with a page hit is now less than 100 clock cycles for all computers except one.

More concurrency looks possible, because the bandwidth is still below the maximum possible bandwidth, and (DDR) SDRAM has four banks, so we also tried traversing four random lists in parallel. These results are in table 4.

Table 4. Time to load a cache line from memory, loading 4 cache lines at a time

Computer (processor clock speed (Mhz), processor, chipset, memory)	Page miss (ns)	Page hit (ns)	Page miss penalty (miss / hit)	Page miss (clock cycles)	Page hit (clock cycles)
166, AMD K6, Intel, SDRAM 66	369	251	1.47	61	42
350, AMD K6, ALI, SDRAM 100	262	145	1.81	92	51
867, G4, Motorola, DDR266	143	38	3.78	124	33
1400, Athlon, AMD, DDR266	149	47	3.19	209	66
1533 Mhz AthlonXP, VIA, DDR333	59	42	1.39	90	65
1800, AthlonXP, Nvidia, DDR266 dual	42	34	1.26	76	60
2000, Opteron, DDR400 dual	25	16	1.62	51	31

Compared to traversing one list, the time per memory access when traversing 4 lists is almost 3 times lower on many of the faster computers. The worst cost of a memory access with a page hit is now 66 clock cycles, and less than 100 clock cycles for a page miss on the faster computers.

4 Marking Using a Queue with Prefetched Nodes

Therefore to improve the performance of the mark phase, we should try to load several nodes in parallel. To achieve this we add a FIFO queue to the mark algorithm.

Instead of immediately marking a node, we first load the node using a prefetch instruction [2], and store the address of the node in the queue. If the queue is not full, we try to find another node to prefetch and store in the queue. Such a node can be found on the stack that is used by the mark algorithm, or if this stack is empty, we use a node from the machine stack that contains all the roots that need to be marked. This process is repeated until the queue is full, or no more nodes are available.

The oldest node address is then removed from the queue, this node is marked and its arguments are pushed on the stack, except for the first one. This one is used for the next iteration of the mark algorithm, so it is prefetched and stored in the queue, etc. If the node was already marked, or has zero arguments, we pop a node from one of the stacks or the queue, and use this node for the next iteration.

So in C syntax the marking algorithm with prefetching is:

```
size_of_queue=0;
for (stack_p = begin_stack; stack_p<end_stack; ++stack_p){
    node = *stack_p;
    if (marked (node))
        continue;

    while (true){
        prefetch (node);
        push_queue (node);
        while (size_of_queue!=max_queue_size){
            if (! stack_empty())
                node = pop();
            else if (stack_p<end_stack)
                node = *stack_p++;
            else
                break;
            if (! marked (node)){
                prefetch (node);
                push_queue (node);
                ++size_of_queu;
```

```
            }
        }
        node = pop_queue();
        if (! marked (node)){
            mark (node);
            n_pointers = n_pointers_in_node (node);
            if (n_pointers>0){
                while (n_pointers>1)
                    push (argument_of_node (node,--n_pointers));
                node = argument_of_node (node,0);
                continue;
            }
        }

        if (! stack_empty())
            node = pop();
        else if (stack_p<end_stack)
            node = *stack_p++;
        else if (! queue_empty ()){
            node = pop_queue();
            -- size_of_queue;
        } else
            break;
    }
}
```

Note that nodes that have already been marked are not prefetched and not stored in the queue. Instead the algorithm tries to find another (unmarked) node to prefetch.

Arrays are treated specially. The elements of arrays are not pushed on the stack (not included in the algorithm above), but are marked by a recursive call to the algorithm, in which the array is used instead of the stack. In that case the `begin_stack` and `end_stack` variables point to the beginning and end of the array. This also makes it possible to prefetch the nodes in the array, because the queue can now be filled with nodes from the array instead of the stack.

Because of the extra instructions needed to maintain the queue, this algorithm will be slower if most of the data can be loaded from the cache, but it will be faster on most processors when data has to be loaded from memory. Therefore, if the amount of data to be marked is small, we use the normal mark algorithm without prefetching, and for larger amounts we use this new algorithm with prefetching.

5 Marking and Reversing Pointers

After running the mark scan algorithm described in the previous sections memory may become too fragmented. If this happens, a compacting garbage collec-

tion is performed the next time. The first phase of this algorithm [5] marks the reachable nodes like the mark algorithm described above, but also has to reverse the backward pointers in the heap, because this is required for the next phase. To make this possible, we don't push just the arguments of a node on the stack when a node is marked, but for each argument we also push the address of the argument field that contains the pointer to the argument node. And of course, some extra code is added to reverse the backward pointers.

We have optimised this algorithm in the same way as the other (non-compacting) mark algorithm, by adding a queue with nodes that are prefetched. For each prefetched node in the queue, this queue now also contains the address of the argument field that contains the pointer to the node, to make reversing backward pointers possible.

6 Measurements

We have implemented the prefetching algorithms described above in the garbage collector of Clean and measured the performance of prefetch queues with three, four or five prefetched nodes.

When the compiler compiles itself the mark phase of the garbage collector is between 20 and 34 percent faster (see table 5) using a PowerPC G4 or Athlon processor with four prefetched nodes, and 10 percent faster with the Opteron.

Table 5. Compiler mark

Computer (processor clock speed (Mhz), processor, chipset, memory)	No Prefetch	Prefetch					
		3	4	5	3	4	5
	MB / s	MB / s			% speedup		
867, G4, Motorola, DDR266	121.54	144.65	146.09	143.54	19	20	18
1400, Athlon, AMD, DDR266	134.11	178.38	179.20	179.78	33	34	34
1533, AthlonXP, VIA, DDR333	158.25	206.09	207.78	207.06	30	31	31
1800, AthlonXP, Nvidia, DDR266 dual	213.53	263.43	264.38	259.73	23	24	22
2000, Opteron, DDR400 dual	305.11	337.27	336.72	333.25	11	10	9

When linking a large (more than 7 MB) application the mark phase of the garbage collector is between 37 and 81 percent faster (see table 6) with four prefetched nodes using an Athlon processor, and 19 percent with the Opteron. We have not run this program on the G4, because this linker does not run on the operating system used by this computer.

Finally, we measured the performance using a mergesort algorithm that sorts large lazy lists of integers (see table 7). Prefetching improves the speed of the mark phase of the garbage collector between 20 and 53 percent with four prefetched nodes. Prefetching five nodes is faster in this case, except for the G4.

We have also measured the speed of the mark and reverse backward pointers phase of the compacting garbage collector for the same programs and computers.

Table 6. Linker mark

Computer (processor clock speed (Mhz), processor, chipset, memory)	No Prefetch	Prefetch					
		3	4	5	3	4	5
	MB / s	MB / s			% speedup		
1400, Athlon, AMD, DDR266	230.34	404.07	417.82	436.68	75	81	90
1533, AthlonXP, VIA, DDR333	311.24	496.45	453.72	505.37	60	46	62
1800, AthlonXP, Nvidia, DDR266 dual	443.82	606.18	607.96	596.74	37	37	34
2000, Opteron, DDR400 dual	747.73	878.28	892.97	876.67	17	19	17

Table 7. Mergesort mark

Computer (processor clock speed (Mhz), processor, chipset, memory)	No Prefetch	Prefetch					
		3	4	5	3	4	5
	MB / s	MB / s			% speedup		
867, G4, Motorola, DDR266	88.84	105.25	106.26	102.35	18	20	15
1400, Athlon, AMD, DDR266	105.90	157.23	162.25	170.17	48	53	61
1533, AthlonXP, VIA, DDR333	132.26	175.70	185.59	184.84	33	40	40
1800, AthlonXP, Nvidia, DDR266 dual	181.76	233.82	235.48	245.77	29	30	35
2000, Opteron, DDR400 dual	278.48	351.32	351.03	356.99	26	26	28

For the compiler this phase with prefetching (see table 8) is between 14 and 21 percent faster using an Athlon processor and 3 percent with an Opteron with four prefetched nodes.

Table 8. Compiler mark and reverse

Computer (processor clock speed (Mhz), processor, chipset, memory)	No Prefetch	Prefetch					
		3	4	5	3	4	5
	MB / s	MB / s			% speedup		
867, G4, Motorola, DDR266	94.80	110.30	111.66	108.32	16	18	14
1400, Athlon, AMD, DDR266	105.57	123.52	127.25	125.52	17	21	19
1533, AthlonXP, VIA, DDR333	128.04	146.52	145.63	147.97	14	14	16
1800, AthlonXP, Nvidia, DDR266 dual	165.91	184.04	189.72	192.93	11	14	16
2000, Opteron, DDR400 dual	266.61	275.40	273.71	276.13	3	3	4

For the linker, the mark and reverse backward pointers phase with prefetching is in most cases about as fast as without prefetching (see table 9).

Finally for mergesort this mark algorithm is between 24 and 51 percent faster with 4 prefetched nodes (see table 10).

7 Related Work

Boehm [1] prefetches nodes in the mark phase of the garbage collector when a node is pushed on the mark stack. Disadvantages of this approach are that

Table 9. Linker mark and reverse

Computer (processor clock speed (Mhz), processor, chipset, memory)	No Prefetch MB / s	Prefetch 3	4	5	3	4	5
		MB / s			% speedup		
1400, Athlon, AMD, DDR266	178.81	177.88	180.42	178.13	-1	1	0
1533, AthlonXP, VIA, DDR333	233.81	247.71	231.17	247.71	6	-1	6
1800, AthlonXP, Nvidia, DDR266 dual	323.69	331.77	320.83	320.17	2	-1	-1
2000, Opteron, DDR400 dual	535.51	556.00	554.02	662.46	4	3	24

Table 10. Mergesort mark and reverse

Computer (processor clock speed (Mhz), processor, chipset, memory)	No Prefetch MB / s	Prefetch 3	4	5	3	4	5
		MB / s			% speedup		
867, G4, Motorola, DDR266	87.14	104.23	105.85	103.80	20	21	19
1400, Athlon, AMD, DDR266	90.98	131.46	137.06	143.18	44	51	57
1533, AthlonXP, VIA, DDR333	116.48	146.52	153.68	151.41	26	32	30
1800, AthlonXP, Nvidia, DDR266 dual	159.25	197.21	201.11	203.29	24	26	28
2000, Opteron, DDR400 dual	260.87	315.66	322.76	321.31	21	24	23

the next node to be marked can not be prefetched, and other nodes may be prefetched too soon. However an advantage compared to our algorithm is that the only extra instructions that have to be executed are prefetch instructions.

To compare both methods, we implemented Boehms prefetching method in our garbage collector as well, and measured the performance on a 1533 Mhz Athlon XP 1800+ processor. The results (see table 11) for our prefetching method are clearly better.

Table 11. Comparing our prefetch algorithm with the Boehm prefetch algorithm

Program	Boehm prefetch (% speedup)	Prefetch 3 (% speedup)	Prefetch 4 (% speedup)	Prefetch 5 (% speedup)
Compiler	9	30	31	31
Linker	5	37	37	34
Mergesort	17	33	40	40

Nethercote and Mycroft [8] used prefetch instructions to speed up writing in a copying garbage collector and allocating memory from the heap for the functional programming language Haskell. This improved the performance on an Athlon processor of the copying garbage collector up to 3 percent, and the execution of programs up to 22 percent.

Chilimbi and Larus [4] describe a copying garbage collector that uses real-time profiling information about data access patterns in object oriented languages to produce a cache-conscious object layout to improve program performance.

Luk and Mowry [6] discuss a compiler that inserts prefetch instructions in code using linked data structures. They also discuss history-pointer prefetching that adds new pointers (called history-pointers) to data structures that contain the address of the node that should have been prefetched during the last traversal. Subsequent traversals of the linked data structure use these pointers for prefetching. This is of course only effective if the traversal patterns are similar. To update these pointers during each traversal a FIFO queue is used which contains the pointers to the last nodes visited. We also use such a FIFO queue, but we do not store these pointers in the nodes to optimize subsequent traversals. Instead we prefetch the node when we store it in the FIFO and change the order in which the graph is traversed so that we can mark other nodes first.

Roth and Sohi [10] discuss several prefetching techniques for linked data structures. They also add extra pointers to some, or all, nodes that point to nodes that are likely to be accessed in the near future. These jump-pointers are used for prefetching.

8 Conclusions

The speed of the mark phase of a garbage collector for large heaps can be improved by adding a FIFO queue with nodes that are prefetched. This reduces the cost of the high main memory access time.

We implemented this algorithm and measured a speedup between 24 and 81 percent using AMD Athlon processors, between 10 and 26 percent using an AMD Opteron processor and about 20 percent using a Motorola G4 processor.

We also used this prefetch method for the mark phase of a compacting garbage collector. This did not improve performance for one of the programs (linker), but still improved performance between 13 and 51 percent for the other programs, except for one: the compiler using an Opteron processor (3 percent).

A queue of four or five prefetched nodes gave the best results.

This algorithm can be used to improve the performance of mark-scan garbage collectors, many compacting garbage collectors and major collections of generational collectors for several processors.

References

1. Hans-J. Boehm, Reducing garbage collector cache misses, *Proceedings of the second international symposium on Memory management* , Minneapolis, Minnesota, United States, Pages: 59 64, 2001, ACM Press
2. David Callahan, Ken Kennedy and Allan Porterfield, Software prefetching, *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* , p.40-52, April 08-11, 1991, Santa Clara, California, United States
3. Cuppu, V. and Jacob, B. (2001). Concurrency, latency, or system overhead: which has the largest impact on uniprocessor DRAM-system performance? In *Proc. 28th annual Int. Symp. on on Computer Architecture*, pages 6271, Goteborg, Sweden.

4. Trishul M. Chilimbi and James R. Larus, Using generational garbage collection to implement cache-conscious data placement, *Proceedings of the first international symposium on Memory management* , p.37-48, October 17-19, 1998, Vancouver, British Columbia, Canada
5. Groningen, J.H.G. van (1995), Optimising Mark-Scan Garbage collection, *The Journal of Functional and Logic Programming* , Volume 1995, MIT Press.
6. Chi-Keung Luk and Todd C. Mowry, Compiler-based prefetching for recursive data structures, *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, p.222-233, October 01-04, 1996, Cambridge, Massachusetts, United States
7. Davis, B., Mudge, T., Jacob, B., and Cuppu, V. (2000). DDR2 and low latency variants. In *Solving the MemoryWall Problem Workshop* , Vancouver, Canada. In conjunction with 26th Annual lnt. Symp. on Computer Architecture.
8. Nicholas Nethercote and Alan Mycroft, The Cache Behavior of Large Lazy Functional Programs on Stock Hardware, *Proceedings of the workshop on Memory system performance*, Berlin, Germany, Pages: 44 55, 2003, ACM Press
9. David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, Katherine Yelick. A Case for Intelligent RAM: IRAM. *IEEE Micro* , vol 17, no. 2, March/April 1997, pages 34 - 44.
10. Amir Roth and Gurindar. S. Sohi, Effective jump-pointer prefetching for linked data structures, *Proceedings of the 26th annual international symposium on Computer architectur*, Atlanta, Georgia, United States, Pages: 111 121, 1999, IEEE Computer Society