Allocation without Locking

Andrew W. Appel*

Department of Computer Science

Princeton University

Princeton, NJ 08544

November 1988

Keywords: garbage collection, dynamic memory allocation, concurrency

Abstract

In a programming environment with both concurrency and automatic garbage collection, the allocation and initialization of a new record is a sensitive matter: if it is interrupted halfway through, the allocating

---

*Supported in part by NSF Grants DCR-8603453 and CCR-8806121 and by a Digital Equipment Corp. Faculty Incentive Grant

process may be in a state that the garbage collector can't understand. In particular, the collector won't know which words of the new record have been initialized and which are meaningless (and unsafe to traverse).

For this reason, parallel implementations usually use a locking or semaphore mechanism to ensure that allocation is an atomic operation. The locking significantly adds to the cost of allocation. This paper shows how allocation can run extremely quickly even in a multi-thread environment: open-coded, without locking.

Copying garbage collection[6][5] can be extremely efficient in large memories. The amortized cost of reclaiming a cell can approach zero[1]. Generational garbage collection[7][8] is even more efficient: the cost per cell approaches zero even in more reasonably-sized memories. In particular, it is not atypical to see an amortized cost per reclaimed cell of less than one instruction.

Since the number of cells reclaimed is similar to the number allocated, we can say that there is an implicit garbage-collection cost of approximately one instruction attributable to each allocation. When the cost of garbage collection is that low, it makes sense to try to minimize the number of instructions necessary to perform an allocation. This is particularly important in languages that perform very frequent allocations; the author's ML compiler[3] typically performs one allocation for every 80 instructions executed. If the cost of the allocation can be reduced from 20 instructions to 5, that's about a 15% improvement in overall system performance.

In LISP, an allocation is typically expressed as **(cons A B)** meaning "allocate a two-word record containing the values **A** and **B** and return a pointer to it." With a compacting garbage collector, the unallocated memory is always a contiguous region. That is, there is no "free list;" instead, there is a free area of memory. The function **(cons A B)** could be implemented with these machine instructions:

1. Test free-space pointer against free-space limit.
2. If the limit has been reached, call the garbage collector.

3. Store A into new record.
4. Store B into new record.
5. Add 2 (the size of a cons cell) to the free-space pointer.
6. Return the previous value of free-space pointer.

We can use the virtual memory hardware of the computer to accomplish the test in line 1. If an inaccessible page is mapped to the region just after the free space, then any attempt to store there (in line 4) will cause a page fault. The operating system maintains data structures telling it which pages are valid, and therefore which page faults are to be handled by fetching from disk and which are to be handled by sending a "segmentation violation" signal (as it's called in Unix) to the user process. This signal can be handled by the user's run-time system, which will initiate a garbage-collection.

On systems that do a lot of allocation, it is worth dedicating a register to hold the free-space pointer to simplify access to it. Therefore, the instruction sequence for **(cons A B)**, moving a pointer for the new cons cell into "dest," might look something like this (expressed in Vax assembly language):

```
movl    B,4(fsp)

movl    A,0(fsp)

movl    fsp,dest

addl2   $8,fsp
```

This sequence of four quick instructions implements the creation of a new cell in only twice the time it would take to fetch all the contents!

We rely on the virtual memory to give a page fault trap when *fsp* reaches the end of the free space. A record may cross a page boundary, however; and it would get very messy if the trap occurred halfway through allocation. Therefore, we ensure that the first store causes the trap by storing the *last* word of the record first. If that store succeeds, then all the rest are guaranteed to succeed (as long as the record is not larger than a page). If a record larger than a page is created, then there should be enough inaccessible pages in a row so their total size is larger than the size of the record; or, locking can be used for large records on the assumption that they are rare.

What happens if this allocation procedure is used in a system where a thread of control may be suspended between any pair of instructions? An allocation may be halfway completed when the interruption occurs, which would cause problems both for other threads of control and for the garbage collector.

For this reason, many implementations use a lock or semaphore in the allocation procedure to prevent two threads from allocating at the same time, and to prevent the garbage collector from running while one thread is allocating. The use of a locking mechanism is very costly, compared to the four- or five-instruction cost (including amortized garbage collection overhead!) of allocating a cell. This paper shows how locking mechanisms can be avoided.

We can easily solve the contention problems between different allocating threads: each thread will be given its own free space. Allocations by thread A in space a won't affect allocations by thread B in space b. When thread A is pre-empted by thread B, the half-finished allocation in space a won't cause problems for any allocations performed in space b.

This leaves the problem of interference between an allocating thread and the garbage collector. If thread A is halfway through an allocation when it is pre-empted, and the garbage collector is invoked before A resumes, then a dangerous situation arises. The garbage collector won't know which fields of the new cell have already been initialized and which haven't. The initialized fields must be traversed (and updated, if the cells that they point to have been moved); but the uninitialized fields are garbage, and should not be traversed.

The solution is to have the garbage collector finish the allocation and initialization of the new cell. The instruction sequence for an allocation (as shown in the previous section) is simple and stereotyped; therefore, it is easy for the allocator to recognize when a thread has been suspended during an allocation. There are only three kinds of instructions allowed during an allocation, and these instructions never occur in any other context:

- Storing to an offset from the free-space pointer (fsp).
- Moving the fsp to a destination register.
- Adding to the fsp.

It is a simple matter to recognize these instructions when they occur after the suspended thread's program counter. In fact, since "adding to the fsp" always occurs at the very end of an allocation, it is possible for the garbage collector to finish the allocation by interpreting the machine-instructions until the add is found:

```
    /* an allocation-interpreter for the Vax */
interp()
{ while (1)
        if ( pc points to "movl rn, k(fsp)" )
                        fsp[k] = reg[n]; advance pc;
        else if (pc points to "movl fsp, rn")
                        reg[n] = fsp; advance pc;
        else if (pc points to "addl2 $k, fsp")
                        fsp = fsp+k; advance pc; return;
        else return;
}
```

In some compilers, the number of fields in a record can be larger than the number of registers, so it is necessary to execute fetch instructions during an allocation, but these too can be stereotyped and recognized.

Interpreting machine instructions is slower by a factor of approximately 20 than executing them directly. An allocation takes approximately 5 instructions (for small records, which are most common), so the overhead of interpreting an allocation is about 100 instructions. However, since a typical process spends only a tenth of its time in the middle of an allocation, then it will usually be suspended outside of any allocation; it takes just the interpretation of one instruction for the interpreter to realize that the program counter is not at one of the stereotyped allocation instructions, and that no more interpretation need be done. And when the process *is* found inside an allocation, then on the average half of the allocation will already be completed. If a process is allocating only 10% of the time, then the average overhead per suspended process is only about $20 \cdot (5 \cdot \frac{1}{2} \cdot \frac{10}{100} + 1 \cdot \frac{90}{100}) = 23$ instructions.

It is important to understand that this interpretation does not take place on every pre-emption. Most pre-emptions are between allocating threads; only rarely does a garbage collection begin. When there is a concurrent garbage collector [4][2], the interpretation need be done only at a “flip” (the beginning of a new cycle). Most of a concurrent collector’s time-slices do not involve a “flip,” and do not need to touch any newly-allocated or partially-allocated cells.

Suppose that a thread allocates 10,000 cells between garbage-collections, and then incurs an (average) 23 instruction overhead for “cleaning up” a partially allocated cell. This is much cheaper than locking, which would require several instructions for each of those 10,000 cells, especially if there was much contention between different allocating threads.

Thus, dynamic allocation with garbage collection can be very cheap even when there are several threads sharing memory. Instead of using a locking mechanism on allocations, each thread can allocate in its own area of memory without regard to interruption. Occasionally the garbage collector will have to clean up an incomplete allocation, but this is not too expensive.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *SIGPLAN Notices (Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation)*, pages 11–20, 1988.

[3] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–324, Springer–Verlag, 1987.

[4] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[5] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[6] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[7] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 23(6):419–429, 1983.

[8] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments)*, pages 157–167, 1984.