Quick Fit:

An Efficient Algorithm for Heap Storage Allocation

Charles B. Weinstock[1]

William A. Wulf[2]

1. Introduction

There is an old maxim that only 10% of the code is responsible for 90% of a program's execution time. This was driven home to the authors during the development of the Bliss-11 compiler [1]. Early versions of the compiler were *very* slow, and timing data showed that heap allocation was the culprit. Replacing the original allocator with the one described below improved performance by nearly an order of magnitude.

Quick Fit is a storage allocator for use in the run-time support for programming languages. A variety of such algorithms are in common use -- *First Fit*, *Best Fit*, various *Buddy Systems,* and others (see [2] or [3]). Unlike these other allocators, our goal was to allocate storage as quickly as possible; memory utilization was of secondary importance. In spite of this emphasis, Quick Fit utilizes storage nearly as well as the best of the allocation algorithms [4].

The basic idea behind Quick Fit is simple, and arose from observations of the behavior of Bliss-11 compiler. In retrospect, and with broader experience, the observations apply to the vast majority of application programs -- namely, for any particular program:

- there are only a few access types [3] (perhaps tens, but seldom hundreds), and
- the most frequently allocated types involve relatively small, fixed amounts of storage (e.g., they are records with only a few fields).

The Quick Fit algorithm exploits these observations to ensure that these common cases are handled rapidly; with a clever implementation it can approach the speed of stack allocation.

2. The Basic Algorithm

Quick Fit is actually a family of related algorithms, and we'll describe some of the variants later. However, the basic scheme divides memory into two logical parts; 1) that which either is either currently allocated or was previously allocated in the past, and 2) that which has never been allocated. We call the former part

---

[1] Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pa.

[2] University of Virginia, Charlottesville, Va.

[3] We have used the Ada terminology here; such types are also known as "pointer types" in other languages.

working storage and the latter portion the *tail* [1]. The size of working storage is the high water mark of the program's memory utilization.

Each block of free space within working storage is linked to one of a number lists. These lists are rooted either in

- one of a set of *quick lists* , one for each block size between two bounds -- *minQL* and *maxQL*; these bounds are chosen to cover the sizes commonly allocated by the program. The quick list roots are arranged as a vector indexed by size, and each quick list is either empty or contains a pointer to a list of all free blocks of that particular size.

- a single *miscellaneous list* , or *misc list* , which contains all free blocks sizes not covered by the quick lists.

The basic scheme for allocation is:

1. If the request is for a block size between minQL and maxQL, and the relevant quick list is not empty, remove the first block from the list and return its address.

2. Otherwise, if the request can be allocated from the tail without extending the address space, do so and return the address. This can be done by simply incrementing a pointer to the beginning of the tail, and returning its pre-incremented value.

3. Otherwise, use one of the more complex strategies discussed later.

The scheme for deallocation is similarly simple:

1. If the size of the storage to be deallocated is covered by one of the quick lists, link it onto the proper list.

2. Otherwise, link it onto the misc list.

Based on the observations about programs stated earlier, it should be obvious why Quick Fit is fast. In the majority of cases, allocation consists of a size comparison, indexing to the proper quick list, a nil-pointer check, and delinking. Even when the appropriate quick list is empty, most of the remaining allocations are from the tail, which merely involves incrementing a pointer. Only rarely is it necessary or desirable to invoke a more complex and inefficient strategy. Similarly, a deallocation only involves a size comparison, possibly indexing to the proper quick list, and a linking operation. Often all these operations can be performed "inline" without undue impact on program size, thus reducing the total overhead to just a few instructions.

Perhaps less obvious are the reasons why Quick Fit utilizes storage effectively. In practice we have found that if a particular sized storage chunk has been allocated and then deallocated, there is a high probability that it will be allocated again. Quick Fit exploits this situation in a way that avoids splitting free chunks into smaller sizes that may not be usable. By contrast, other algorithms tend to create small, unusable blocks.

---

[1] Because most operating systems allocate the address space to the user in chunks, the tail is more precisely that part of memory already allocated from the operating system, but never used by the program. In systems with this characteristic, when the tail is exhausted a new chunk of address space may be allocated from the operating system. In such cases, space may not be contiguous, but this presents no problem for the algorithm.

3. Refinements

A wide variety of variations on the Quick Fit theme are possible. As a trivial example -- one may wish to allocate in certain discrete "chunk" sizes -- say words instead of bytes. This is, in fact, done in the illustrative implementation later.

A more substantive example involves the "more complex strategies" to be used when the two simple schemes fail, and specifically the policy for allocation from the misc list. The original Bliss-11 implementation used a First Fit strategy for management of the misc list, but any of the well-known algorithms would do. Standish [3], in particular, developed a variant of Quick Fit that he called *Fast Fit* by implementing a balanced AVL tree on the misc list. He found a slight improvement in both overall allocation speed and storage utilization.

The Bliss-11 implementation was done for a machine with much smaller address space than is common today, and that led to two other refinements: (1) a policy for periodically coalescing adjacent free blocks, and (2) a policy for *splitting* larger free blocks in order to satisfy a smaller request whose quick list is empty. Our later implementations on large address-space machines have eliminated these refinements, but they still are appropriate for smaller, possibly embedded machines. However, neither should be done indiscriminately, as doing so can cause the benefits of the Quick Fit algorithm to disappear.

4. Brief Analysis and Commentary

There are two obvious ways to evaluate an algorithm such as Quick Fit: (1) in relative terms (how does it compare to other algorithms such as First Fit, etc.), or (2) in absolute terms (the time it takes to perform an allocation).

The relative performance of the Quick Fit algorithm and many others were analyzed in [4], and these results were summarized by Standish [3]. The reader can refer to either of these for a more detailed discussion. The analysis involved simulating the behavior of many storage allocation algorithms, using both theoretical and observed size and lifetime distributions for the request strings.

The results of such experiments are sensitive to the request strings used, however in general the analysis showed that our intuition was correct, namely most allocations *do* come from the Quick Lists and the Tail -- 80% of the time allocation is from the quick lists, and 96% of the time it is either from the Quick Lists or the Tail.

The results also showed that one variant of the Buddy System is slightly faster, and that one form of the Best Fit allocation strategy utilizes storage slightly more efficiently than Quick Fit. Quick Fit, however, is *significantly* better overall -- it is nearly as fast as the fastest, and nearly as storage efficient as the most efficient.

The absolute performance can be evaluated by examining the code compiled by an optimizing compiler[1]. Because nearly all requests are for blocks whose size is known at compile time (that is, they are records), and this size is within the quick fit range, we'll look at that case. Suppose that the requested block size is in register r0, and that the resulting address is to be returned in r1, then the generated code is:

---

[1] This example was compiled with an early experimental version of the Tartan Ada/VAX/VAX compiler, and then hand edited for readability.

```
        movl    QL[r0], r1          ; get quick list root and set CC's
        beq     L1                  ; is root pointer nil?
        movl    (r1), QL[r0]        ; no, delink 1st free block
        br      done
L1:     cmpl    r0, tailremaining   ; is there enough space in the tail?
        bgtr    L2                  ; no, use the complex allocator
        movl    tailbegin, r1       ; get beginning of the tail
        addl3   r0,r1,tailbegin     ; increment it
        subl2   r0, tailremaining   ; decrement space remaining in tail
        br done
L2:     pushl   r0                  ; push the size request
        calls   $1, hardalloc       ; call the complex strategy routine
done:
```

Thus, 80% take four instructions and 16% take eight. The remaining 4% take longer, but just how much longer depends upon the choice of the "hardalloc" algorithm(s).

The expense of heap allocation is often cited as a reason to avoid certain data structures -- and there are some applications for which this is a valid concern. However, the cost of heap allocation need not be as high as many people believe, nor, indeed, as high as it actually is in many systems. Algorithms such as Quick Fit make heap allocation almost as inexpensive as stack allocation, and consequently enable use of more natural, better structured program structures.

5. References

1. Wulf, William A., Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, Charles M. Geschke, **The Design of an Optimizing Compiler**, American Elsevier, 1975.

2. Knuth, Donald, **Sorting and Searching**, The Art of Computer Programming, Volume III, Addison-Wesley, 1973.

3. Standish, Thomas A., **Data Structure Techniques**, Addison-Wesley, 1980.

4. Weinstock, Charles B., **Dynamic Storage Allocation**, Ph.D. Thesis, Carnegie Mellon University, 1976.

Appendix: The Basic Quick Fit Algorithm

```
------------------------------------------------------------------
------------------------------------------------------------------
--
--          The Package Specification for Quick Fit
--
------------------------------------------------------------------
------------------------------------------------------------------

with SYSTEM;
generic
   minQL:  integer := 0;              -- minimum quick list item size
   maxQL:  integer := 32;             -- maximum quick list item size
   OSchunk:  integer := 4096;    -- How much we get when we go to the OS.
package QuickFit is
   function  Alloc(n: natural) return system.address;
   procedure DeAlloc(s: system.address; n: natural);
end QuickFit;



------------------------------------------------------------------
------------------------------------------------------------------
--
--          The Package Body for Quick Fit
--
------------------------------------------------------------------
------------------------------------------------------------------

with UNCHECKED_CONVERSION;
package body QuickFit is

------------------------------------ Types For Storage on a Free Lists
type free_cell;                              -- a free block (see below)
type freePtr is access free_cell;       -- pointer to a free block
type free_cell is
   record
          next:freePtr;                      -- pointer to the next free block on a list
          size:natural;                     -- size of the current free block
   end record;

------------------------------------ Constants Related to Allocation Size
Chunk:     constant integer := natural'size/system.storage_unit;
MinChunk:   constant integer := free_cell'size/system.storage_unit;

------------------------------------ The Free Space Pointers
QL:array(minQL..maxQL) of freePtr;          -- One list for each size
MiscList:    freePtr;                          -- A single list for all other sizes.
TailBegin:    system.address;                -- Beginning of space in the tail
TailRemaining: natural := 0;                 -- Amount of space left in the tail
```

```
---------------------------------- Unchecked Conversion Between Types
function ConvertToFreePtr is new UNCHECKED_CONVERSION(natural,freePtr);
function ConvertToFreePtr is new UNCHECKED_CONVERSION(system.address,freePtr);
function ConvertToAddress is new UNCHECKED_CONVERSION(freePtr,system.address);
function ConvertToAddress is new UNCHECKED_CONVERSION(natural,system.address);
function ConvertToNatural is new UNCHECKED_CONVERSION(system.address,natural);


---------------------------------- Support Routines

function "+"(s: system.address; i: natural) return system.address is
   -- addition between addresses and naturals
   begin
     return ConvertToAddress(ConvertToNatural(s) + i);
   end "+";

function ComputeSize(size: natural) return natural is
   -- round  a size request up to (a) an integral number of "chunks, and
   -- (b) at least MinChunk (so that there is always space for the free
   -- space information).
   begin
     if size < minChunk
       then return minChunk;
            else return (size + (Chunk-1))/Chunk;
     end if;
    end ComputeSize;


---------------------------------- Allocation Strategy Routines

function QuickAlloc(size: natural) return system.address is
   -- perform an allocation from one of the quick lists
   cell: freePtr;
   begin
     cell := QL(size);
     QL(size) := cell.next;
     return  ConvertToAddress(cell);
   end QuickAlloc;

function TailAlloc(size: natural) return system.address is
   -- perform an allocation from the tail
   locn: system.address;
   begin
     locn := TailBegin;
     TailBegin := TailBegin+size;
     TailRemaining := TailRemaining-size;
     return locn;
   end TailAlloc;
```

```
function MiscAlloc(size: natural) return system.address is
   -- perform an allocation from the miscellaneous list
   begin
     -- Use whatever method pleases you.
     -- Return the null address if allocation is not possible.
     -- The next line is only for syntatic correctness.
     return ConvertToAddress(null);
   end MiscAlloc;

procedure ExtendTail is
   -- obtain new storage from the operating system
   begin
     if TailRemaining > 0 then DeAlloc(TailBegin,TailRemaining); end if;
   -
     -- TailBegin := <<space obtained from the operating system.>>
     -- <<raise storage_error if the OS fails>>
     -- TailRemaining := OSchunk;
   -
   end ExtendTail;

function HardAlloc(size: natural) return system.address is
   -- control routine for the expensive, out-of-line allocations
   locn: system.address;
   use SYSTEM;
   begin
     if size > OSchunk then raise storage_error; end if;
     locn := MiscAlloc(size);
     if locn /= ConvertToAddress(null) then return locn; end if;
     extendTail;
     return TailAlloc(size);
   end HardAlloc;


---------------------------------- Entry Points to the QuickFit Allocator

function Alloc(n: natural) return system.address is
   -- allocate a chunk of size (at least) n storage units
   size:  natural;
   begin
     size := ComputeSize(n);
     if size in minQL..maxQL then
           if   QL(size) /= null then return QuickAlloc(size);
           elsif size <= TailRemaining then return TailAlloc(size);
           else return HardAlloc(size);
           end if;
     elsif size <= TailRemaining then return TailAlloc(size);
     else  return HardAlloc(size);
     end if;
   end Alloc;
```

```
procedure DeAlloc(s: system.address; n: natural) is
  -- deallocate the chunk at s, of length n
  size: natural;
  cell: freePtr := ConvertToFreePtr(s);
  begin
    size := ComputeSize(n);
    cell.size := size;
    if not (size in minQL..maxQL)
      then cell.next := MiscList; MiscList := cell;
           else cell.next := QL(size); QL(size) := cell;
    end if;
  end DeAlloc;

end QuickFit;
```

U.S. Postal Service

STATEMENT OF OWNERSHIP, MANAGEMENT AND CIRCULATION

Required by 39 U.S.C. 3685)

1A. TITLE OF PUBLICATION	1B. PUBLICATION NO	2. DATE OF FILING
ACM SIGPLAN NOTICES	6 0 4 - 4 9 0	9/15/88

3. FREQUENCY OF ISSUE	3A. NO. OF ISSUES PUBLISHED ANNUALLY	3B. ANNUAL SUBSCRIPTION PRICE
Monthly	12	Member $22- Non-Member $35-

4. COMPLETE MAILING ADDRESS OF KNOWN OFFICE OF PUBLICATION *(Street, City, County, State and ZIP+4 Code) (Not printers)*

11 West 42nd. St., New York, N.Y. 10036

5. COMPLETE MAILING ADDRESS OF THE HEADQUARTERS OF GENERAL BUSINESS OFFICES OF THE PUBLISHER *(Not printer)*

11 West 42nd. St., New York, N.Y. 10036

6. FULL NAMES AND COMPLETE MAILING ADDRESS OF PUBLISHER, EDITOR, AND MANAGING EDITOR *(This item MUST NOT be blank)*

PUBLISHER *(Name and Complete Mailing Address)*

Association For Computing Machinery, 11 West 42nd. St., N.Y., N.Y. 10036

EDITOR *(Name and Complete Mailing Address)*

Richard Wexelblat, Philips Labs, 345 Scarborough Rd, Briarcliff Manor, NY 10510

MANAGING EDITOR *(Name and Complete Mailing Address)*

7. OWNER *(If owned by a corporation, its name and address must be stated and also immediately thereunder the names and addresses of stockholders owning or holding 1 percent or more of total amount of stock. If not owned by a corporation, the names and addresses of the individual owners must be given. If owned by a partnership or other unincorporated firm, its name and address, as well as that of each individual must be given. If the publication is published by a nonprofit organization, its name and address must be stated.) (Item must be completed.)*

FULL NAME	COMPLETE MAILING ADDRESS
Association For Computing Machinery, Inc. (a non-profit organization)	11 West 42nd. St., New York, N.Y. 10036

8. KNOWN BONDHOLDERS, MORTGAGEES, AND OTHER SECURITY HOLDERS OWNING OR HOLDING 1 PERCENT OR MORE OF TOTAL AMOUNT OF BONDS, MORTGAGES OR OTHER SECURITIES *(If there are none, so state)*

FULL NAME	COMPLETE MAILING ADDRESS

9. FOR COMPLETION BY NONPROFIT ORGANIZATIONS AUTHORIZED TO MAIL AT SPECIAL RATES *(Section 423.12 DMM only)*
The purpose, function, and nonprofit status of this organization and the exempt status for Federal income tax purposes: *(Check one)*

(1) [X] HAS NOT CHANGED DURING PRECEDING 12 MONTHS
(2) [ ] HAS CHANGED DURING PRECEDING 12 MONTHS *(If changed, publisher must submit explanation of change with this statement.)*

10. EXTENT AND NATURE OF CIRCULATION *(See instructions on reverse side)*	AVERAGE NO. COPIES EACH ISSUE DURING PRECEDING 12 MONTHS	ACTUAL NO. COPIES OF SINGLE ISSUE PUBLISHED NEAREST TO FILING DATE
A. TOTAL NO. COPIES *(Net Press Run)*	12,400	12,326
B. PAID AND/OR REQUESTED CIRCULATION 1. Sales through dealers and carriers, street vendors and counter sales	0	0
2. Mail Subscription *(Paid and/or requested)*	12,000	11,990
C. TOTAL PAID AND/OR REQUESTED CIRCULATION *(Sum of 10B1 and 10B2)*	12,000	11,990
D. FREE DISTRIBUTION BY MAIL, CARRIER OR OTHER MEANS SAMPLES, COMPLIMENTARY, AND OTHER FREE COPIES	100	98
E. TOTAL DISTRIBUTION *(Sum of C and D)*	12,100	12,088
F. COPIES NOT DISTRIBUTED 1. Office use, left over, unaccounted, spoiled after printing	300	238
2. Return from News Agents	0	0
G. TOTAL *(Sum of E, F1 and 2 - should equal net press run shown in A)*	12,400	12,326

11. I certify that the statements made by me above are correct and complete

SIGNATURE AND TITLE OF EDITOR, PUBLISHER, BUSINESS MANAGER, OR OWNER

Fred Hamm — Publisher

PS Form 3526, July 1984 *(See instruction on reverse)*