ITC consists of a linear list of addresses of words which contain addresses of routines to be executed. This level of indirection is illustrated by the example shown in Figure 2.

Comparing Figures 1 and 2 shows an important qualitative difference. ITC does not involve the compilation of any code as such, only addresses. Since almost all machines implement the concept of words which can contain addresses, ITC may be generated in an essentially machine independent manner. On the other hand, DTC involves the generation of operand access routines, which makes machine independence much more difficult.

To examine space and time requirements, we compare the DTC generated by the PDP-11 FORTRAN compiler [3] to the ITC which might be generated by an equivalent compiler. Figure 3 shows that not only is the space decreased but also the time. The comparison is actually overly favorable to DTC. First, the routine \$F0001, which must be generated as part of the compiler output to obtain the short BR jumps in the \$P000*n* routines, has not been counted at all in the space since it is generated only once for several variables. Second, only the fetch (push) has been shown. Inclusion of the store (pop) adds an extra routine (DTC) or extra word (ITC) for each variable. Actually, the PDP-11 compiler generates a parametrized call for each store, which costs one extra word per assignment.

If threaded code were to be generated in register machine style as opposed to stack machine style, the saving would be even greater since in this case each register would require two addresses (ITC) or two routines (DTC) for each operand.

Several other advantages of ITC manifest themselves in specific connection with the SPITBOL system.
1. The address of the operand fetch routine which is associated with a dynamic value can serve as the type code identifying the value in dynamic memory.
2. The linking from one code block to another is easily affected by making the initial address in the new block point to the routine for switching code blocks. The SPITBOL system uses this fact to generate a separate code block for each statement leading to the interesting property that unreachable statements are garbage collected.
3. The operand fetch and store routine addresses, which are associated with a variable, may be modified in the case where the variable is input or output associated.

Received June 1974; revised September 1974

References
1. Dewar, R.B.K. SPITBOL. SNOBOL4 Doc. S4D23, Illinois Inst. of Tech., Chicago, Ill., Feb. 1971.
2. Bell, James R. Threaded code. *C.ACM 16*, 6 (June 1973), 370–372.
3. Digital Equipment Corporation, PDP-11 FORTRAN IV Programmer's Manual, DEC-11-KFDA-D, Maynard, Mass., 1971.

Programming Techniques

G. Manacher Editor

A Simplified Recombination Scheme for the Fibonacci Buddy System

Ben Cranston and Rick Thomas
The University of Maryland

A simplified recombination scheme for the Fibonacci buddy system which requires neither tables nor repetitive calculations and uses only two additional bits per buffer is presented.

Key Words and Phrases: Fibonacci buddy system, dynamic storage allocation, buddy system
CR Categories: 3.89, 4.32, 4.39

A severe problem in the Fibonacci dynamic storage allocation scheme is to locate the buddy buffer when recombining. Hirschberg [1] gives two algorithms. The first of these involves a table which grows in size rapidly with the maximum size buffer allowed. For a system allowing buffers of up to 17,717 words, this table requires 987 entries. The second algorithm involves a costly repetitive calculation. This paper presents a more efficient algorithm[1] which requires only two additional bits in some control field of each buffer.

Let the two additional bits be called the *B*-bit (for Buddy bit) and the *M*-bit (for Memory). These are in addition to the usual *A*-bit (buffer is currently in use and

This work was supported in part by Navy Contract ONR-N0014-67-A-0239-0032-01.

Authors' address: The University of Maryland, Computer Science Center, College Park, MD 20742.

[1] James A. Hinds [2] has described another scheme which has most of these advantages but requires more than two additional bits.

Fig. 1. Behavior of buffer control fields.

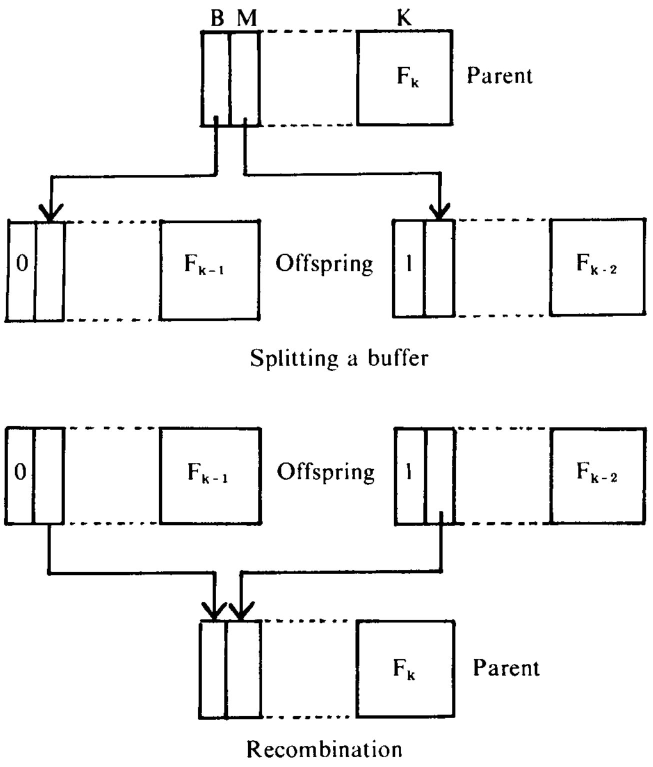
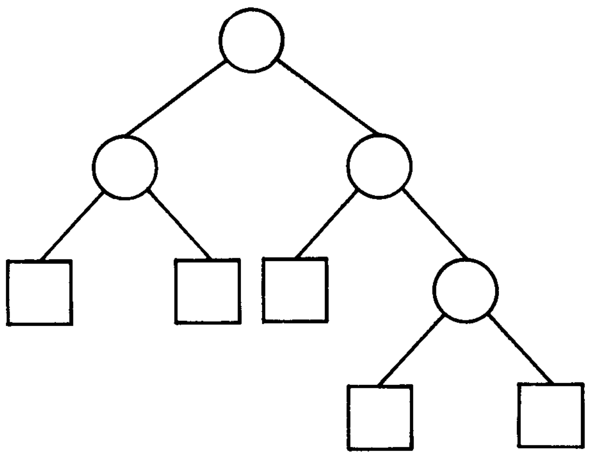


Fig. 2. Binary tree representation of the storage pool.



K-field (buffer size key—that is, the size of the buffer is F_k, where k is the contents of the buffer's *K*-field). Upon breaking up a buffer of size F_{k+1} into two smaller buffers F_k and F_{k-1} as shown in Figure 1, the *M*-bit of the parent buffer is stored as the *M*-bit of the right offspring, and the *B*-bit of the parent is stored as the *M*-bit of the left offspring. The *B*-bit of the left offspring is set to zero, and the *B*-bit of the right offspring is set to one. Upon recombination the *B*-bits of the offspring are discarded, and the *M*-bits are used to reconstruct the *M*- and *B*-bits of the parent. Since the parent buffer's *M*- and *B*-bits are preserved in the *M*-bits of the two offspring, it is clear that no information is lost in this process.

The *B* (Buddy) bit in each buffer now describes the location of the buddy buffer. If *B* is zero then the buffer is a "left" buffer and the buddy is found by adding F_k to the address of the buffer, where k is the contents of the buffer's *K*-field. If *B* is one then the buffer is a "right" buffer and the buddy is found by subtracting F_{k+1} from the address of the buffer. (This follows the convention that the left buddy is the larger of the two.)

This technique can also be used in generalized Fibonacci buddy systems [2]. A generalized Fibonacci buddy system is one in which the buffer sizes satisfy the relation $S_n = S_{n-1} + S_{n-k}$ where S_n is the *n*th buffer size available and k is some positive integer. For $k = 1$, this describes the binary buddy system [3]. For $k = 2$, this describes the Fibonacci buddy system of [1]. Such a system is completely determined by the choice of k and the first k values of S_n.

The following discussion shows that the job can be done with two bits, and no fewer. The status of buffer storage can be represented by a binary tree (see Figure 2). The squares represent buffers which are unsplit, that is, either in use or in the available storage pool, and the circles represent former buffers which have been split. The relationship between the number of unsplit buffers and the number of split buffers is given by $U = S + 1$ where U is the number of unsplit buffers and S is the number of split buffers. Adding U to both sides gives $2U = U + S + 1$. In order to recombine, every buffer must have associated with it one bit to represent its "handedness." Since this requires a total of $(S + U)$-bits, the above formula shows that two bits per unsplit buffer supply one bit more than required. Thus two bits per buffer seems to be a practical minimum for all schemes which store "handedness" directly.

Thus with only two additional bits per buffer, Fibonacci buddy buffering can be as fast as binary buddy buffering [3], and yet significantly reduce fragmentation. A dynamic storage allocation system using this Fibonacci buddy buffering scheme has been implemented as part of a compiler for the SIMPL programming language [4].

Received May 1974; revised September 1974

References

1. Hirschberg, D. A class of dynamic memory allocation algorithms. *Comm. ACM 16*, 10 (Oct. 1973), 615–618.
2. Hinds, James A. An algorithm for locating adjacent storage blocks in the buddy system (in preparation).
3. Knuth, D. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Reading, Mass., 1968, pp. 443–444.
4. Basili, V., and Turner, J. SIMPL-T, a structured programming language. CN-14, U. of Maryland, College Park, Md.