Software methods for virtual storage of executable code

P. J. Brown

Computing Laboratory, The University, Canterbury, Kent

Virtual storage systems were originally based on swapping to and from a disc or drum. More recently, however, the same techniques have been used in new areas, where the constraints imposed by electro-mechanical devices do not apply. This necessitates a fresh evaluation of storage management strategies. An area where new strategies may reap specially good rewards is the management of code areas by software.

(Received November 1977)

Virtual storage is concerned with transferring information between *secondary storage* and *primary storage* with the purpose of giving the illusion that the primary storage is bigger than its actual physical size. Traditionally secondary storage is pictured as a drum or disc and primary storage as 'core'. We will call this *disc/core virtual storage*. More recently, virtual storage techniques have been used in two other areas:

1. *Machine/machine linkage*
Minicomputers or microcomputers are frequently connected to mainframes, and in some cases the mainframe provides the smaller machine with a virtual storage system. Here the 'secondary storage', that provided by the mainframe, may even have faster access times than the primary storage, though its effective speed to the small machine will be controlled by the speed of the interconnecting link. (Moreover the secondary storage may itself be subject to a disc/core virtual storage mechanism.) In the future, machines are likely to be linked in much more complex ways, and thus it is best to think of the big machine/small machine situation as a peephole to a much larger and more general picture of co-operating machines. Among the facilities that one machine may supply to another will be virtual storage.

2. *Procedure/procedure linkage*
An example of this use of virtual storage is where a 'compiling' procedure supplies 'pages' of code to an 'executing' procedure running in the same machine. If the executing procedure discards a page and subsequently needs it again, then the compiling procedure regenerates it (see, for example, Brown, 1976).

Cases 1 and 2 are logically similar, and we will use the term *process/process virtual storage* to cover both of them.

Purpose of this paper

Process/process virtual storage normally needs to be implemented exclusively by software methods. Disc/core virtual storage is sometimes aided by hardware mechanisms such as automatic paging; sometimes it is provided exclusively by software. The purpose of this paper is to examine such software mechanisms, and to see how process/process virtual storage might change traditional ideas.

We shall henceforth use the neutral term *unit* to describe the blocks in which storage is allocated. A unit may be a fixed sized 'page' or a variable sized 'segment'.

Software virtual storage

The main aid that hardware can provide for virtual storage is the translating of virtual addresses into real addresses, using such devices as page tables. Software virtual storage systems are faced with performing this *address translation* as an extra overhead. If every single address reference has to be interpreted by software then this overhead may be immense, and thus the cornerstone of most strategies is to try to minimise it.

Separating code and data

One way of minimising the address translation overhead is to separate code areas from data areas and take advantage of the special properties of each. Most software virtual storage mechanisms do this, and indeed many only deal with code areas, since these tend to be the simpler. In addition, assuming code areas to be read-only, discarded units do not need to be preserved; hence (as a gross simplification) you get twice as much for your money with code areas as against read/write data areas.

Another great advantage of code is that it is normally executed sequentially. Only perhaps one instruction in ten is a jump instruction which changes the sequence of execution. It is only on the virtual addresses that are the operands of jumps that address translation is needed. (Address translation may also be needed if a sequence of code runs over a unit boundary. However if each unit is made to end with a jump to the next unit this problem is covered by the jump mechanism.)

This paper concentrates on code areas.

Naur's method

The best description of a software system for virtual storage of code areas is still, without much doubt, Naur's (1963) paper on GIER ALGOL. Naur worked in fixed size units of 40 words, the block size of the drum on the GIER machine. The code for each unit was position independent—a feature available on most modern hardware—and thus could be loaded anywhere in primary storage without modification. Jumps within units presented no problem since they were relative and hence position independent. Jumps between units needed to be interpreted to perform the correct address translation. At the end of each unit a jump to the next unit was added, and these jumps also needed to be interpreted. Naur put all constants within the code, so there was no need to have a potentially large 'constant table' in primary storage. A constant was included in every unit that referenced it.

A record was kept of the usage of each unit in primary store, and the least recently used unit was discarded when the need arose.

Naur's technique has been reinvented many times since. (Perhaps even Naur reinvented it and some earlier author deserves credit.) In one variant (Colin, McGregor and Mitchell, 1975) units are of 1,024 words and consist of an integral number of procedures. No procedure can cross a unit boundary and the user is encouraged to write related procedures adjacently so they are likely to go in the same unit. Procedure calls are interpreted in the same way as Naur's jumps are. However the authors say that this only represents an overhead of 13 instructions over a direct procedure call.

Replacement algorithms

A key to a virtual storage scheme is the replacement strategy, i.e. the strategy that selects which unit(s) to discard if primary storage becomes full. Hoare and McKeag (1972) survey some of these strategies and Denning (1970) also gives an analysis.

One radical and absurdly simple strategy that is not mentioned in these surveys is the *discard-all* strategy. This strategy, which is geared to code areas, simply involves discarding all units in primary storage and starting again from scratch. The reason this 'strategy' is not mentioned in the surveys is, of course, that it would be a stupid one for hardware virtual storage of disc/core transfers. However the discard-all strategy has advantages which, in other circumstances, may outweigh its naivity. These are as follows.

1. Being so ridiculously simple, the strategy takes few instructions to code and thus leaves more store to manage; it runs fast too, as there is so little to do.
2. Because space is allocated sequentially there is no problem if units are of variable size, and there is no wasted storage (i.e. none of Randell's (1969) *external fragmentation*) except at the very end.

The above two advantages apply generally, but there are two further advantages if there is no hardware support for address translation.

3. If one new unit flows directly into another one there is no need to insert a jump instruction in between, because the second unit is certain to follow the first one in storage and control can flow straight through.
4. Jumps from one unit in primary storage to another can be made absolute—they do not need to go through the address translation process. This is because if one unit is discarded they all are; hence there can be no dangling references. (See the next Section for further details.)

The combined effect of 3 and 4 is that if a program does completely fit into primary storage, then once the program has been completely loaded the virtual storage management system gives no overheads on the running of the program, because it is completely bypassed (save perhaps for subroutine returns). Address translation is performed once for each occurrence, not continuously.

Because of these advantages, the discard-all strategy was chosen for process/process virtual storage management by Brown (1976).

Overwriting virtual jumps by absolute ones

One way of overwriting jumps to virtual addresses by jumps to absolute addresses would be the following: as each new unit comes into primary storage, all jumps from other units to virtual addresses within the new unit are identified and overwritten by absolute jumps. This method involves a lot of housekeeping and is not attractive.

A better method is to represent each virtual jump as a call of a 'self-effacing' system subroutine, the desired virtual address being passed as an argument. If this call can be represented by a single machine instruction, such as a TRAP or extracode instruction, so much the better. When the self-effacing subroutine is entered it finds the absolute location of the required virtual address, loading a new unit if necessary, and then overwrites the call of itself (provided that this is still in primary storage) by an absolute jump to the required location.

Wasted code

Randell defines *internal fragmentation* as the storage wasted when a logical segment has to be rounded up so that it fits into an integral number of pages. There is, however, a potentially much larger source of wastage: this is code that is loaded but never executed. We call this *wasted code*. (*Wasted data* could be defined similarly, but is of less interest because, unlike wasted code, it is hard to avoid.) As an extreme example consider a procedure which has the form

```
IF BOOLEAN THEN
BEGIN ⟨100 statements⟩
END;
```

and assume that this whole procedure is loaded as a segment. If the procedure is never entered with BOOLEAN true during its sojourn in primary storage then the bulk of the procedure is wasted code. (If BOOLEAN was never true during the whole program run then it would always be wasted code; hence wasted code can arise independently of a virtual storage system. Indeed in a run of a large program it would be surprising if as much as 50% of the code was exercised, and as little as 50% wasted.)

Wasted code is simple to avoid, provided that units can be of variable size. All that is necessary is to work in units that are logical units of the *execution* of the program. Systems usually work in units of the static layout of the program, such as blocks and procedures, but these are not suitable execution units. This is a point well made by Naur, who cites earlier unfortunate experience of a scheme based on static units. (Unfortunately, at the time, 1963, physical constraints prevented his using logical run-time units absolutely.) It is not clear if Naur's point has been properly taken in the intervening years.

The natural unit of program execution is the *basic block*, a term coined by those interested in flow analysis (see, e.g. Allen and Cocke, 1976). A basic block is a sequence of instructions that are not jumps, terminated with a jump. Here a 'jump' includes a conditional jump, a subroutine call or return, a stop or a 'table jump' derived from a CASE statement.

If storage is allocated in units of basic blocks and with no internal fragmentation, there will be no wasted code (save in error situations and certain possible interrupt situations).

Avoiding wasted code does not in itself guarantee good performance, but it is a good start.

Size of units

The striking facet of basic blocks as units is that they are small, perhaps averaging 10 instructions or 30 to 40 bytes.

There is still controversy over the ideal size for a unit. Hoare and McKeag report suggestions varying from 1 to 8,192 words. (Units of one instruction avoid wasted code!) Much thinking in this area is coloured by disc/core virtual storage where physical block sizes may be of fixed size and latency overheads such as disc seek times are large. These factors swamp the relative advantages of small units cited by Batson *et al.*, (1970) and Coffman and Varian (1968). In process/process virtual storage none of these physical constraints applies and thus small units of variable size are more feasible.

Apart from possible physical constraints the only two objections to small units are the size of the unit table (i.e. the page or segment table) and the time to search this table. (In the process/process case, the unit table, and the algorithms associated with it may actually reside in the secondary storage, e.g. the 'mainframe'.) The unit table is likely to represent an overhead of two words per unit. Even if this comes to 20% of a unit size it is likely to be more than compensated by the savings in wasted code. As regards speed, computer science has put great effort into table look-up techniques and has produced effective methods even for rapidly changing tables such as the unit tables might be. Nevertheless there obviously must be some speed penalty, and thus strategies that minimise address translation overheads are best.

Increasing unit size

In one realisation of basic blocks as units (Brown, 1976) the following two devices are used to make units rather bigger. Firstly calls of run-time system subroutines (other than the self-effacing one described earlier) are *not* counted as the end of a unit. This is because they always return (except in error cases) and they are locked into primary storage and thus can not cause any unit to be discarded. (The system is actually a BASIC time sharing system where the run-time package is shared between all users. That is why it is locked in.) This device at least doubled the size of units.

Secondly, if one unit ends with an unconditional jump to another, then the two can be dynamically combined into one. (If the second unit is also subsequently jumped to from a third unit, then this represents a jump into the middle of the new combined unit. In this case the 'label' at the head of the second unit needs to go into the unit table.) A similar technique can be used for subroutine jumps, which might be treated as an instruction to stack a return link followed by an ordinary direct jump.

Another strategy

So far we have examined Naur's strategy, which has fixed sized units and discards the least recently used one, and the discard-all strategy, which works with variable sized blocks and minimises address translation overheads.

An intermediate strategy is a FIFO strategy. This is used in the software virtual storage management of code areas by Bobrow and Murphy (1967). They simply put code into a circular buffer. They allow variable sized units and when a new unit is loaded it automatically overwrites one or more units at the previous head of the buffer.

In such a scheme it is possible to make forward jumps (i.e. jumps to a more recently loaded unit) absolute since the jump must disappear before the unit it references does. However backward jumps need to go through an address translation mechanism because the opposite applies; actually if the unit referenced is discarded and subsequently reloaded the backward jump will become a forward one and can be made absolute.

Comparison

We now have three strategies to compare, and will evaluate them from four standpoints: simplicity, ability to deal with variable sized units, address translation overheads and efficiency of unit selection. The last is a measure of how clever the strategy is at choosing the correct page to discard (see discussion of the principle of optimality in Denning, 1970). The following table summarises the respective strategies.

	Discard-all	*FIFO*	*Naur*
Simplicity	Great	Good	Reasonable
Variable sized units?	Yes	Yes	Not easy
Address translation overhead	Minimal	Small	Fairly small
Efficiency of unit selection	Terrible	Fair	Good

Not surprisingly, the table shows no 'best buy'. Any choice of strategy involves trading off a large number of factors of which the above table gives only four. Even mathematical analysis and mathematical models, as Hoare and McKeag observe, cannot give a complete guide to the best choice. However a general trend is evident: Naur's method looks good for a typical disc/core situation whereas in a process/process situation the other methods have the right properties.

Conclusions

With changing technology in both hardware and software, virtual storage strategies do not always need to be dominated by sizes of disc tracks and latency times on electro-mechanical devices. In the absence of such physical constraints it is possible to work in logical units of program execution, such as basic blocks, and thus achieve dramatic savings in wasted code. Such units are small and of variable size and may necessitate new storage management strategies. One possibility is the discard-all strategy.

References

Allen, F. E. and Cocke, J. (1976). A program data flow analysis procedure, *CACM* Vol. 19, No. 3, pp. 137-147.

Batson, A., Shy-Ming Ju and Wood, D. C. (1970). Measurements of segment size, *CACM*, Vol. 13, No. 3, pp. 155-159.

Bobrow, D. G. and Murphy, D. L. (1967). Structure of a LISP system using two level storage, *CACM*, Vol. 10, No. 3, pp. 155-159.

Brown, P. J. (1976). Throw-away compiling, *Software—Practice and Experience*, Vol. 6, No. 3, pp. 423-434.

Coffman, E. G. and Varian, L. C. (1968). Further experimental data on the behaviour of programs in a paging environment, *CACM*, Vol. 11, No. 7, pp. 471-474.

Colin, A. J. T., McGregor, D. R. and Mitchell, D. (1975). Paging STAB-1 code, *Software—Practice and Experience*, Vol. 5, No. 3, pp. 309-316.

Denning, P. J. (1970). Virtual memory, *Computing Surveys*, Vol. 2, No. 3, pp. 153-189.

Hoare, C. A. R. and McKeag, R. M. (1972). A survey of storage management techniques, in Hoare and Perrott, Eds., *Operating Systems Techniques*, Academic Press, London, pp. 117-151.

Naur, P. (1963). The design of the GIER ALGOL compiler: Part I, *BIT*, Vol. 3, pp. 124-140.

Randell, B. (1969). A note on storage fragmentation and program segmentation, *CACM*, Vol. 12, No. 7, pp. 323-333.

Book review

Fundamentals of Computation Theory, edited by M. Karpinski, 1977; 542 pages. (*Springer-Verlag Lecture Notes in Computer Science 56*, $19.80)

This volume in the Springer Series of Lecture Notes records the proceedings of the 1977 International FCT Conference held in Poland in September 1977. It gives over 60 papers organised in three sections: A. Algebraic and constructive theory of machines and computations, B. Computation theory in category, C. Computability, decidability and arithmetic complexity. The papers which range from fundamental, deeply abstract to practical, give a wide coverage of modern computability theory which will be of interest to those who work in the field of logic and foundations of mathematics and also people interested in real computational problems.

The volume is a typical product of Springer's policy of publishing rapidly produced versions of conference proceedings. It has the virtue of bringing together results from writers in America, Western and Eastern Europe faster than would be the case by any other means and as such will be found valuable by all interested in the foundations of mathematics and computing.

A. Young (Coleraine)