Program Stacking Technique

Michiharu TSUKAMOTO*

Abstract

A technique named 'program stacking' is introduced and its applications are described. The technique pushes programs into parallel stacks, transfers control to them, and pops them up when completed their execution. The concept of program stacking extends 'execute instruction' to a program. Programs are protected and executed in parallel stacks. Therefore, this technique enables even a program with program modification to be reentrant or recursive.

1. Introduction

Present day computers adopt what is called the stored program method, in which procedures and data take the same form of representation and storage. In the early days of the computer, however due to the small amount of available main storage, programs consisting of procedures and data were treated as one and could be modified by themselves. This technique called 'program modification,' was abandoned soon after, for to the following reasons:

(1) In a multiprocessing or on-line real time processing environment programs must have reentrancy. For this purpose, such processes as disabling and the enabling interrupts and saving and resumption of the modified information are required.

(2) Recursiveness required in high-level language processors is hard to implement.

(3) Programs are prone to bugs.

(4) Debugging is difficult due to dynamic modification of the program.

It is therefore said to be a "bad programming technique"[5].

However, for those programs which treat the programs in object code as data, such as trap handlers, simulators and tracers, the method of program modification is still an useful technique. The technique has no recursiveness and reentrancy in itself, but in most cases of its use reentrancy is particularly required.

In this article, assuming a hardware stack feature, we propose and generalize a

---

This paper first appeared in Japanese in Joho－Shori (Journal of the Information Processing Society of Japan), Vol. 18, No. 3 (1977), pp. 237～244.

* Information and Control Section, Control Division, Electrotechnical Laboratory

technique called 'program stacking' which puts the modified portions of a program or the portions to be modified onto a stack and executes them.

This technique is an extension of the instruction to a program in a form which permits interruption, or the extension of the use of a stack or dynamic store to a program. As related method, we have the mixed code approach[1,2] which uses this technique as part of the program.

2. Program modification

Program modification is required when the procedure and its data are not separated regionally as well as functionally. In this case, the program-modified portions must not be repeatedly modified again until they become unnecessary. Program modification causes the loss of reentrancy and recursiveness. In order to keep them, disabling and enabling of reentry, and the saving and resumption of the modified portion are necessary. However, the former blocks the other processes and even the latter becomes difficult if there are many portions to be modified or a portion to be modified varies in size.

3. Principle of program stacking

We have devised a method of scheduling, similar to multi-level interruption in which each process is assigned a priority and processing takes place in order of priority, the process which has no preassigned priority being used. In section 3.2 we discuss the former and in section 3.3 the more general cases of the latter.

3.1 Stack

The stack feature is implemented on a computer as follows[3]:

(1) The stack is composed of a stack area(SA) to store information and a stack pointer word(SPW) to control it.

(2) SA is a contiguous area in the main store and occupies from low to high addresses.

(3) SPW is composed of the following elements:

Stack pointer(SP), space counter(SC), and work counter(WC).

During stack operation, if another process interrupts it and uses the same stack, inconsistency will result in the relation between SP and its object information or in the relation between SC and WC. So, the operations must be the hardware instructions.

If the stack is implemented as above, it has the following four properties:

Property 1 Storage of information.

Property 2 Protection of the stored information from damage.

Property 3 Information in the stack may be refered directly using SP and the index register, without push and pop.

Property 4 SA is ordinary storage, so that we may load object programs in the stack and execute in it.

The feature to push object programs in the stack and execute them is called 'program stacking.'

3.2 Basic program stacking

A portion cut out of a program with an entry at the top and an exit at the bottom is called a basic hunk*. Program stacking of a basic hunk is called the basic program stacking.

Algorithm 1 Basic program stacking

(1) Place the program which is executable in the stack area at the moment onto the stack and form a basic hunk in the stack.

(2) Put into the stack the instruction to return to the original sequence.

(3) Load in the index register, X, the complement of the number one less than the number of words stacked in the process of (1) and (2).

(4) Using the following instruction, transfer control to the top of the basic hunk formed in (1):

```
B    *SP,X      ; branch to (SP)+(X)
```

(5) Execute the basic hunk, and return to the original sequence when executing the instruction created in (2).

(6) Pop up the codes of (1) and (2).

If a part of a program can at any time be reentered, the program is called always reentrant in that part. We have the following:

Theorem 1 If processes are processed in order of priority, the technique of basic program stacking is always reentrant.

The theorem guarantees that the program modification and its execution are always reentrant if the control is the nesting.

3.3 Extension to general processes

Since the control structure handled by a stack is restricted to the LIFO

---

* Dawson calls by basic hunk the program unit which has one entry and exit at the last or whose last instruction is jump, call or return[2]. We restrict ourselves to the first type of Dawson's here.

structure, it is impossible to implement program stacking for many processes on a single stack. It is solved by providing a program stacking stack for each process. The parallel stacks are implemented as follows:

(1) Include SPW in the process control block.

(2) Arrange the stack area by a dynamic storage mechanism without relocation.

Now we have the following:

Theorem 2 In the processing of general processes, the technique of basic program stacking is always reentrant.

The theorem is an extention of the use of dynamic storage to a program. Hereafter, a stack means one of the parallel stacks implemented by the dynamic storage mechanism.

4. Extension of program stacking

Since a program contains various types of branching instructions such as BRANCH, CALL and RETURN, it is often difficult to extract a portion of a program as a basic hunk. In this chapter, we extend the concept of program stacking to general programs.

4.1 Control structure of a program required in program stacking

In program stacking, transfer of control between modules and the management of the related stacks must be done correctly. In regard to the transfer of control between modules, the control structure is classified into the four types:

(1) Stagnant type (2) Call type (3) Return type (4) Jump-out type

In type (4) the control, exiting from the executing module, never returns to the same module again, and a program is remained in the stack. In order to apply program stacking, the program must be enclosed in a call and the corresponding return, such as subroutine.

4.2 Method of program construction

Programs of closed structure as described in the previous section are generally large and their program stacking requires much time to push and pop. Moreover, it requires large capacity. However, if we ensure that no jump-out occured in the stack, then the parts without program modification can be executed outside the stack, and the above inconvenience could be avoided by appropriate partitioning and reconstruction of the program. An aggregate of program portion every of which has a branching instruction at the bottom is called an extended hunk. In the following we describe a method to reconstruct a program and form extended hunks, in order to effectively apply program stacking. We assume, however, that the entry is at the top of program and

we can use at least one index register for work.

Algorithm 2 Method of program construction (Fig. 1)

(1) Construct a program P_{r0} without regard to program stacking (Fig. 1-a).

(2) Divide the program P_{r0} into program-modified parts and not program-modified parts. In this partitioning we do not split the portions which are logically indivisible, such as macro instruction consisting of several words and the portions which save the status of the computer. We denote the partitioned program by $\text{hunk}^i_{i=1,2,..}$ (Fig. 1-b).

(3) Add a label(L^i) to the top of each partition(hunk^i), and a branch instruction(B L^{i+1}) to its end. The program now takes the form L^i: hunk^i; B L^{i+1} $_{i=1,2,...}$(Fig. 1-c).

(4) Concatinate the partitions that are program-modified on the one hand and those not program-modified on the other, and get the two extended hunks P_{r1} and P_{r2}, respectively(Fig. 1-d):

$$P_{r1} = \underset{i=1,3,5,..}{L^i: \text{hunk}^i; \text{ B } L^{i+1}} \qquad P_{r2} = \underset{i=2,4,6,..}{L^i: \text{hunk}^i; \text{ B } L^{i+1}}$$

where p_{r1} is the extended hunk to be program-modified and p_{r2} the extended hunk not to be program-modified. If otherwise, we put (B L^1) in the top of P_{r2} and newly denote this by P_{r1} and the previous P_{r1} by P_{r2}.

(5) Rewrite the instructions which refer P_{r1} to the instructions refering it by the relative address from its top. Here we assume that P_{r1} is the extended hunk to be placed on the stack and P_{r2} the extended hunk to be remained off the stack.

It is clear that the P_{r1} is the smallest extended hunk in program stacking. The two extended hunks thus constructed call each other in the unit of partition. This has the same control structure as a 'coroutine.'

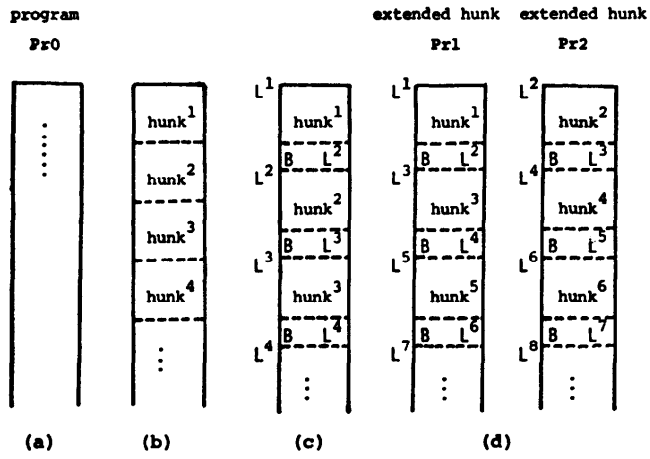

Fig.1 Construction of two extended hunks
(a) Original program
(b) Partition
(c) Add labels and branch instructions
(d) Construct two extended hunks

4.3 Extended program stacking

As described in the previous section, instructions which refer a stacked hunk must refer it by the relative address from its top. Extended hunks are dynamically allocated in the stack area. It is necessary to have something like a base register to avoid address modification at stacking time. We therefore extend the stack as follows:

Definition Extended stack

A stack with the addition of the following element to the stack pointer word(3.1).

Hunk pointer(HP): Points to the top of the last extended hunk in the stack.

Using this, program stacking is extended as follows:

Algorithm 3 Extended program stacking

It is assumed that the extended hunks to be program-modified are constructed according to Algorithm 2.

I. Preprocessing

(1) Push return address in the stack.

(2) Save HP in the stack.

(3) Update HP (i.e., HP:=SP+1).

(4) Push the extended hunk in the stack.

II. Execution

(5) Transfer control to the extended hunk in the stack by the following instruction:

```
B    *HP        ; branch to (HP)
```

III. Postprocessing

(6) Pop the stack until SP=HP.

(7) Restore HP from the stack.

(8) Fetch the return address and transfer control to the caller.

This is always reenterant, too. I and II may be considered the extension of subroutine call and III that of return.

If we notice that the extended program stacking dynamically loads the extended hunks to the stack, links and automatically releases the area, this may be considered a type of dynamic overlay.

5. Conclusion

For the hardware stack feature we have described a technique named program stacking which executes a program in parallel stacks. We have also discussed the method of reconstructing a program in order to assure effective application of the

technique. The characteristics of the technique are summarized as follows:

(1) The function of execute instruction is extended to multiple instructions, that is to programs.

(2) Use of a stack of dynamic store is extended from the one for data to the one for programs.

A stack or dynamic store, for example subroutine-link-stack, is usually used for the storage and protection of data. In the present technique, object codes are first loaded, protected in it as data and are executed afterward by jumping into it, so procedures and data are completely protected from outside.

When the size of hunk is small like the interface of existing subroutines and monitors, basic program stacking can be applied directly, and we can modify the virtual interface without modifying the interface of the existing subroutines and monitors. Since the technique renders a program reentrant and recursive, it has been effectively utilized in the implementation of drivers of monitor call in LISP processor[4]. Manual coding of extended hunks is too complicated and is likely to produce a lot of bugs. This may be solved by mechanical processing to Algorithm 2.

The author wishes to express his deep gratitude to Mr. Koichi Furukawa, Dr. Tadashi Nagata and Dr. Hirochika Inoue for their valuable advice and guidance, as well as to the members of Robot Research Group, Control Division, for their interesting discussions.

References

1) R.J. Daking et al.: A Mixed Code Approach, Comput. J., vol. 16, No. 3, pp. 219-222 (1973)

2) J.L. Dawson: Combining interpretive code with machine code, Comput. J., vol. 16, No.3, pp. 216-219 (1973)

3) Mitsubishi Electric Corp.: MELCOM 7700 system reference manual (1973)

4) M. Tsukamoto: On Foreground LISP Interpreter, Proceedings of IPSJ, vol. 15, pp. 653-654 (1974)

5) P. Wegner: Programming Language, Information Structures, and Machine Organization, McGraw-Hill, New York (1968)