Multiprogramming STRETCH: Feasibility Considerations

E. F. Codd, E. S. Lowry, E. McDonough, C. A. Scalzi, *International Business Machines Corporation, Poughkeepsie, New York*

Abstract.[1] The tendency towards increased parallelism in computers is noted. Exploitation of this parallelism presents a number of new problems in machine design and in programming systems. Minimum requirements for successful concurrent execution of several independent problem programs are discussed. These requirements are met in the Stretch system by a carefully balanced combination of built-in and programmed logic. Techniques are described which place the burden of the programmed logic on system programs (supervisory program and compiler) rather than on problem programs.

1. Introduction

In recent years there has been a trend towards increased parallelism in computer design. The prime aim of this parallelism is to allow more of the component units of a computer system to be kept in productive use more of the time. Two forms have clearly emerged. The first, which we shall call *local parallelism*, consists of overlapping the execution of an instruction with that of one or more of its immediate neighbors in the instruction stream.

This form of parallelism was present in a very early machine, the IBM Selective Sequence Electronic Calculator, which was capable of working on three neighboring instructions simultaneously. Such parallelism was later abandoned in the von Neumann type machines such as the 701. Now that we have once again reached a stage in which the logical elements are much faster than the memories, the need for this type of parallelism has returned. In fact, the Stretch system is capable of working on as many as *seven* neighboring instructions simultaneously.

The second form of parallelism, which we shall call *nonlocal*, provides for concurrent execution of instructions which need not be neighbors in an instruction stream but which may belong, if you please, to entirely separate and unrelated programs. It is this form of parallelism upon which we wish to focus attention.

In order for a computer system to exhibit nonlocal parallelism, it must possess a number of connected facilities, each capable of operating simultaneously (and independently, except for memory references) on programs which need not be related to one another. A facility may be an input-output unit, a file unit, an arithmetic unit, a logical unit or some assemblage of these units. In an extreme case each facility is a complete computer itself.

Stretch is a multiple facility system. The following facilities are capable of simultaneous operation on programs which need not be related:

a. one (or more) central processing unit(s)

b. each channel of the basic exchange (a switching and coordinating unit for moderate speed input-output activities)

c. each disk (seeking only)

d. the read-write channel of the high speed exchange (a switching and coordinating unit for high speed input-output activities)

The multiple facility computing system bears a close resemblance to a job shop although the analogy can be taken too far. Just as the jobs to be processed in a job shop are split up into tasks which can be handled concurrently by the available facilities, so may programs be subdivided into such tasks. At any instant the tasks being executed simultaneously may belong to the same program or to different programs. One object of concurrently running tasks which belong to different (perhaps totally unrelated) programs is to achieve a more balanced loading of the facilities than would be possible if all the tasks belonged to a single program. Another object is to achieve a specified real-time response in a situation in which messages, transactions, etc., are to be processed on-line. A third object is to expedite and simplify diagnostics and certain types of problem solving by making it economically feasible for the programmer to use a console for direct communication with (and alteration of) his program.

2. Multiprogramming Requirements

Several problems arise when concurrent execution of programs sharing a common memory is attempted. For example it is almost certain that sooner or later, unless special measures are taken, one program will make an unwanted modification in another due to a programmer's blunder. Then again, when some unexpected event occurs, it is not merely a matter of deciding whether it was due to a machine malfunction, a programming blunder or an

[1] Note: Optimizing problems associated with multiprogramming are not discussed in this paper.

operator error. It is necessary to know which of the several programs may have been adversely affected and which one (if any) was responsible.

Such questions make it desirable to establish a set of necessary conditions which a multiprogramming system must satisfy if it is to be generally accepted and used. We propose the six conditions described below.

a. *Independence of Preparation.* The multiprogramming scheme should permit programs to be independently written and compiled. This is particularly important if the programs are not related to one another. The question of which programs are to be co-executed with which should not be pre-judged even at the compiling stage.

b. *Minimum Information from Programmer.* The programmer should not be *required* to provide any additional information about his program for it to be run successfully in the multiprogrammed mode. On the other hand, he should be permitted to supply extra information (such as expected execution time if run alone) to enable the multiprogramming system to run the program more economically than would be possible without this information.

c. *Maximum Control by Programmer.* It may be necessary in a multiprogramming scheme to place certain of the machine's features beyond the programmer's direct influence (for example, both clocks in STRETCH). This reduction in direct control by the problem programmer must not only be held to an absolute minimum, but must also result in no reduction in the effective logical power available to the programmer.

d. *Noninterference.* No program should be allowed to introduce error or undue delay into any other program. Causes of undue delay include a program which gets stuck in a loop, and failure of an operator to complete a requested manual operation within a reasonable time.

e. *Automatic Supervision.* The multiprogramming scheme must assume the burden of the added operating complexity. Thus, instructions for handling cards, tapes, and forms should originate from the multiprogramming system. Similarly, machine malfunctions, programming errors, and operator mistakes should be reported to the responsible party in a standard manner by the multiprogramming system. Again, all routine scheduling should be handled automatically by the system in such a way that the supervisory staff can make coarse or fine adjustments at will. Further responsibilities of the system include accounting for the machine time consumed by each job and making any time studies required for operating or maintenance purposes.

f. *Flexible Allocation of Space and Time.* Allocation of space in core and disk storage, assignment of input-output units, and control of time-sharing should be based upon the needs of the programs being executed (and not upon some rigid subdivision of the machine).

These requirements are met in the STRETCH system by a carefully balanced combination of built-in and programmed logic. The hardware for multiprogramming would be far too cumbersome and expensive if an attempt were made to meet these requirements by built-in logic alone. Further, the method of meeting certain of these requirements (particularly the automatic scheduling requirement) must be variable from user to user due to variations in their objectives.

First, then, let us consider those multiprogramming features which are provided in the hardware in STRETCH.

3. Multiprogramming Features in STRETCH

Before mentioning some specific features, it is important to note that extensive use of programmed logic in a multiprogramming scheme can easily prove self-defeating because the time taken by the machine to execute the multiprogramming program may offset the gain from concurrent execution of the problem programs. However, the raw speed and logical dexterity of STRETCH are such that it is possible to employ quite sophisticated programmed logic.

We now describe four major features in STRETCH which facilitate multiprogramming.

a. *Program Interruption System.* This system has already been described in some detail; see [2]. Briefly, it permits interruption of a sequence of instructions whenever the following four conditions are satisfied:

(i) the interruption system is enabled,
(ii) no further activity is to take place on the current instruction,
(iii) an indicator bit is on,
(iv) the corresponding mask bit is on.

The indicators reflect a wide variety of machine and program conditions which may be classified into the following five types:

(i) signals from input-output units, other central processing units, etc.;
(ii) data exceptions such as data flags, zero divisors or negative operands in square root operations;
(iii) result exceptions such as lost carries, partial fields or floating point exponents without certain ranges;
(iv) instruction exceptions such as instructions which should not or cannot be completed or should signal when they are completed; and
(v) machine malfunctions.

When several problem programs are being executed concurrently, certain of these conditions are of private concern to the particular program which caused their occurrence. Other conditions, particularly types 1 and 5, are of general concern. Each of the indicators for conditions of private concern has a variable mask bit which allows the current program the choice of suppressing or accepting interruption for the respective condition. On the other hand, each of the indicators for conditions of general concern possesses a fixed mask bit (permanently set in the ON position). This feature, combined with appropriate control measures respecting the disabling of the entire interruption system, virtually eliminates the possibility that an interruption of general concern is suppressed and lost.

Another aspect of the interruption system which is of importance to multiprogramming is the interrupt table. When an interruption is taken, control is passed (without changing the contents of the instruction counter) to one of the instructions in an interrupt table. The base address of this table is variable so that several such tables may exist simultaneously in memory, for example, one table for each problem program. However, only one is active at a time. The relative location within the active table which supplies the interjected instruction is determined by the indicator (and hence by the particular condition) causing interruption.

Exploitation of this interruption system depends upon programmed interrupt procedures. This aspect will be taken up when we deal with programmed logic for multiprogramming.

b. *Interpretive Console.* It has been customary in general purpose computers to provide a single console at which an operator can exercise sweeping powers over the whole machine. For example, by merely depressing the STOP button the operator has been able to bring the entire activity of the machine to a halt. On the other hand, the normal requirement in multiprogramming is to communicate with a particular program and at the same time allow all other programs to proceed. Pursuing the same example, we now desire to stop *a program* rather than *the machine.*

For this reason and because it is required that several consoles be concurrently operable with varying objectives, the STRETCH console is not directly connected to the central processing unit. Instead, it is treated as an input-output device. Its switches represent so many binary digits of input and its lights so many binary digits of output. No fixed meaning is attached to either. By means of a console defining routine one can attach whatever meaning one pleases to these switches and lights.

c. *Protection System.* References by the central processing unit to memory are checked. If the address falls within a certain fixed area or within a second variable area, the reference is suppressed and an interruption occurs. The boundaries of the variable area are specified by two addresses stored within the fixed area. These addresses may be changed only if the interruption system is disabled.

This system allows any number of programs sharing memory to be effectively protected from each other. At any instant, the central processing unit is servicing only one program (logically speaking). Suppose this is a problem program P. The address boundaries are set so that P cannot make reference outside of its assigned area. Before any other problem program Q acquires the CPU, the address boundaries are changed to values which will prevent Q from making reference outside of the area assigned to Q. The task of changing address boundaries is one of the programmed functions of the Stretch multiprogramming system.

d. *Clocks.* There are two clocks in STRETCH which are usable by programs.

The first, referred to as the elapsed time clock, is a 36-bit binary counter which is automatically *incremented* by unity once every millisecond. This clock may be read by a program under certain conditions but cannot be changed by a program under any condition whatsoever. It is intended for measuring and identifying purposes, particularly in accounting for machine use, logging events of special interest and identifying output. It takes more than two years for this clock to go through a complete cycle.

The second clock, referred to as the interval timer, is a 19-bit binary counter which is automatically *decremented* by unity once every millisecond. Under certain conditions the interval timer may not only be consulted but may also be set to any desired value by a program. Whenever the interval timer reading reaches zero, an interruption occurs (if the interruption system is enabled). The main purpose of this device is to provide a means for imposing time limits without requiring programmed clock-watching: that is, frequent inspection of the elapsed time clock.

There are several other features in STRETCH which facilitate multiprogramming. To avoid going into too much detail, we shall merely make a brief reference to one of these. The exchanges assume all the burden of word assembly on input and disassembly on output. Once an input-output operation has been started, the responsible exchange is sufficiently autonomous that it does not need to interrupt the central processing unit in order to borrow some of its logical abilities (and time) for the purpose of completing the operation. It is capable of conducting the entire operation itself even though, for example, transmission of several variable length blocks from tape is involved and the channel instructions are scattered in memory. As a result of this degree of autonomy, the frequency of input-output interruptions is considerably reduced.

4. Programmed Logic

Now we turn our attention to the programmed logic and discuss how we propose to exploit the built-in logic by programming techniques in order to meet the six requirements for acceptable multiprogramming. Three tools are at our disposal: the supervisory program, the compiler, and the source language.

The supervisory program is assumed to be present in the machine whenever multiprogramming is being attempted. It is assigned the job of allocating space and time to problem programs.

Allocation of space includes determining which areas of memory and disk storage and which input-output units are to be assigned to each of the programs. The space requirements (including the required number of input-output units of each type) are produced by the compiler as a vector whose components are quantities dependent in a simple way upon one or more parameters which may change from run to run. Any space requirements depend-

ing on parameters are evaluated at loading time when the particular values of the run parameters are made available.

The supervisory program uses its precise knowledge of the space requirements of a problem program together with any information it may have regarding the expected execution time and pattern of activity to determine the most opportune time to bring that program into the execution phase. It is not until the decision to execute is made that specific assignments of memory space, disk space, and input-output units are put into effect. By postponing space allocation until the last minute, the supervisory program maintains a more flexible position and is thus able to cope more effectively with the many eventualities and emergencies which beset computing installations no matter how well managed they are.

Allocation of time includes not only determining when a loaded program should be put into the execution phase but also handling queues of requests for facilities from the various programs being concurrently executed. The fact that both pre-execution queueing and in-execution queueing are handled by programming rather than by special hardware results in a high degree of flexibility. Thus, at any time the supervisory program is able to change the queue discipline in use on any shared facility and so cope more effectively with the various types of space and time bottlenecks which may arise. On interruptible facilities, such as the STRETCH CPU, which allow one program to be displaced by another, changes in queue discipline may be expected to have very considerable effect upon the individual and collective progress of the programs being co-executed.

These allocating powers of the supervisory program have several implications. Most important of these is that the compiler must produce a fully relocatable program—relocatable in memory and in disk storage, and with no dependence on a specific assignment of input-output units. A further consequence is that the supervisory program is responsible for all loading, dumping, restoring, and unloading activities, and will supply the operator with complete instructions regarding the handling of cards, tapes and forms.

In order to meet the requirements of independent preparation of problem programs and noninterference with one another, it is necessary to assign the following functions to the supervisory program:

a. direct control of the enabled/disabled status of the interruption system,

b. complete control of the protection system and clocks,

c. the transformation of I/O requests expressed in terms of symbolic file addresses into absolute I/O instructions (a one-to-many transformation) followed by the issuing of these instructions in accordance with the queue disciplines currently in effect,

d. the initial and (in some cases) complete handling of interruptions from I/O units and other central processing units.

By convention, whenever a problem program is being serviced by the central processing unit, the interruption system is enabled. On the other hand, when the supervisory program is being serviced, either the enabled or the disabled status may be invoked according to need. Adherence to this convention is assisted by the compiler which

a. refrains from generating in problem programs the instruction BRANCH DISABLE (an instruction which completely disables the interruption system), and

b. whenever it encounters this instruction in the source language itself, substitutes a partial disable (a pseudo instruction) in its place, flagging it as a possible error.

So long as the interruption system is enabled, the protection system is effective. Problem programs are therefore readily prevented from making reference to the areas occupied by other programs (including the supervisory program itself). They are further prevented from gaining direct access to the address boundaries, the interrupt table base address, and the clocks, all of which are contained in the permanently protected area.

For the sake of efficient use of the machine, one further demand is made of the programmer or compiler. When a point is reached in a problem program beyond which activity on the central processing unit cannot proceed until one or more input-output operations belonging to this program (or some related program) are completed, then control must be passed to the supervisory program so that other problem programs may be serviced.

It is important to observe that we do *not* require the programmer or compiler to designate places in the program at which control may be taken away if some higher priority program should need servicing. We believe this to be an intolerable requirement when unrelated programs are being concurrently executed, especially if all arithmetic and status registers at such places must contain information of no further value.

It is the interruption system (particularly as it pertains to input-output) which makes this requirement unnecessary. It allows control to be snatched away at virtually any program step, and the supervisory program is quite capable of preserving all necessary information for the displaced program to be resumed correctly at some later time.

In removing certain features of the machine from the direct control of the problem programmer, we may appear to have lost sight of the requirement that he should have a maximum degree of control. However, for every such feature removed, we have introduced a corresponding pseudo feature. Take for example, the pseudo-DISABLE and pseudo-ENABLE instructions. When a problem program P issues a pseudo-DISABLE, the supervisory program effectively suspends all interruptions pertaining to P (by actually taking them and logging them internally) until P issues a pseudo-ENABLE. Meanwhile, the interruptions pertaining to other programs not in the pseudo-disabled state are permitted to affect the state of the queue for the CPU.

Another example of a pseudo feature is the pseudo interval timer: one of these is provided for each problem program. The supervisory program coordinates the resulting multiple uses of the built-in interval timer.

The need to detect that a program has become stuck in a loop, or that an operator has not responded to an instruction from the supervisory program, is met by allotting a reasonable time limit for the activity in question. When this interval expires without the supervisory program receiving a completion signal, an overdue signal is sent to an appropriate console. The interval timer is, of course, used for this purpose and expiration of the interval is indicated by the time signal interruption.

5. Concluding Remarks

In order to provide a practical demonstration of the feasibility of multiprogramming STRETCH, the authors are developing an experimental supervisory program with general multiprogramming capabilities. The experiments to be performed with this system are aimed at exploring conditions under which multiprogramming is profitable to the user.

REFERENCES

STRETCH

1. S. W. DUNWELL, "Design Objectives for the IBM Stretch Computer", *Proceedings of EJCC* (December 1956).
2. F. P. BROOKS, JR., "A Program-Controlled Program Interruption System", *Proceedings of EJCC* (December 1957).
3. W. BUCHHOLZ, "The Selection of an Instruction Language", *Proceedings of WJCC* (May 1958).
4. F. P. BROOKS, JR., G. A. BLAAUW, W. BUCHHOLZ, Processing data in bits and pieces, *IRE Transactions on Electronic Computers* (June 1959).
5. G. A. BLAAUW, Indexing and control word techniques, *IBM Journal of Research and Development* (July 1959).
6. H. K. WILD, "The Input-Output Devices of the Stretch Computer", paper presented at Auto-Math 59 Exhibition, Paris (June 1959).

MULTIPROGRAMMING

7. S. GILL, Parallel programming, *The Computer Journal* (April 1958).
8. C. STRACHEY, "Time Sharing in Large Fast Computers", *Proceedings of International Conference on Information Processing* (June 1959).
9. W. F. SCHMITT, A. B. TONIK, "Sympathetically Programmed Computers", *Proceedings of ICIP* (June 1959).
10. J. BOSSET, "Sur Certains Aspects de la Conception Logique du Gamma 60", *Proceedings of ICIP* (June 1959).
11. A. L. LEINER, W. A. NOTZ, J. L. SMITH, R. B. MARIMONT, "Concurrently Operating Computer Systems", *Proceedings of ICIP* (June 1959).
12. J. W. FORGIE, The Lincoln TX-2 Input-Output System, *Proceedings of WJCC* (1957).

Flow Outlining—A Substitute for Flow Charting

W. T. GANT, *Shell Oil Company, Midland, Texas*

The computing group with Shell Oil Company, Midland, Texas, has been using a method of describing problems that replaces the need for flow charts where coding must be done with a symbolic assembler instead of a compiler. We call it flow outlining and consider it superior to flow charting because it is less time consuming to prepare, easier to code from, and permits more detailed remarks where needed. Probably the most attractive advantage is the ease with which one can make additions or cross out errors and then hand a very marked-up page to a typist and get back a neat corrected copy.

The form is quite simple, consisting of a symbolic identification column and a statement of what is to be performed. The same symbolic identification is used in the coding, thus making it very easy to check the coding with the flow outline. Additional statements can be added by using any convenient subscripts desired. A sample example follows.

AA	Reset SUM Y to zero, eject page.
BB	Read a card, branch to BBi if i = 1, 2, 3, 4 where i is digit in column 1. If i ≠ 1, 2, 3, 4, go to CC.
BB1	Alphabetic comments card, print columns 11–80 alphabetically, go to BB.
BB2	Data card, containing X_1, X_2, X_3, compute $Y = X_1 + X_2X_3 + X_1X_2$.
BB2A	Print X_1 X_2 X_3 Y.
BB2B	SUM Y + Y → SUM Y, go to BB.
BB3	Total card, double space, print SUM Y, go to AA.
BB4	End of job, start next program.
CC	Error condition, print error message, pass remaining data cards, start next program.