A STUDY OF THE CONSTRAINTS UPON THE PARALLEL DISPATCHING AND EXECUTION OF MACHINE CODE INSTRUCTIONS

Caxton C. Foster
Edward M. Riseman
Department of Computer and Information Science
Technical Report 72A-1
University of Massachusetts
Amherst, Mass. 01002

This work was supported in part by
Control Data Corporation

TABLE OF CONTENTS

ABSTRACT

An infinite machine, one with an infinite instruction stack, infinite registers and memory, and infinite numbers of functional units is defined. This paper investigates the increase in parallel execution rate as a function of the size of an instruction decode stack with look-ahead hardware. Seven programs written for a CDC-3600 were "run" on this machine. Under the constraint that instructions are not dispatched until all preceding conditional jumps are resolved, stack sizes as small as 2 or 4 achieve most of the parallelism that a hypothetically infinite stack would. The improvement over a standard machine is only 1.72:1.

An algorithm is described which can be used to replace the look-ahead hardware of the stack by reordering the sequence of instructions prior to execution. The transformed sequence has the property that if the instruction at the top of the stack cannot be dispatched immediately, there will be no instruction below it that is ready for dispatching. Experimental results demonstrate that this method achieves 93.5% of the parallelism obtained if an infinite decode stack were available under the assumption that it takes zero time to dispatch an instruction.

If it was assumed that the manner in which conditional jumps are resolved is known, then instructions could be dispatched before the jumps are executed. Under these assumptions the infinite machine on the average ran 51 times as fast as they did on a standard 3600. To reach ten times the speed of a "one-instruction at-a-time" machine,

sixteen jumps must be "by-passed" in an actual machine which would imply 65K paths in simultaneous execution.

I. INTRODUCTION

The problem of detecting and utilizing parallelism in programs has been extensively studied. Many techniques have been developed to detect parallelism in higher level languages, particularly in arithmetic expressions [1], [2]. There have been a number of proposals for FORK and JOIN type instructions for the programmer himself to specify where and how two or more sequences of instructions are executed simultaneously [3]. The huge Illiac IV has been implemented to take advantage in hardware of array operations that can be executed in parallel [4]. However, this type of machine is used effectively only on a restricted class of problems.

A different approach is the design of a general purpose computer to automatically detect when more than one instruction in the instruction stream can be executed simultaneously in parallel. In the case of a single instruction stream - single data stream machine, Flynn pointed out in 1966 that the bottleneck is the decoding and dispatching of a single instruction per machine cycle [5]. Thus, instructions may be executed in parallel but they are dispatched sequentially as in a number of current computers: IBM 360/85, 91, 195 and CDC 6600 and 7600.

Recently, Tjaden and Flynn [6] have examined the payoff in using a hardware stack to dispatch as well as execute instructions in parallel. They determined the increase in execution rate as a function of the size of the stack. Simulations show an 86% increase with a decode stack of size 10. One problem in using such a stack is that the cost of the look-ahead hardware goes up as the square of the stack size; each instruction must be compared with all instructions preceding it in the decode stack.

The process of dispatching instructions at the maximum rate is complicated further by the presence of conditional branches. Until the conditional is resolved, it is not known which of the two instruction paths proceeding from the conditional should be fetched and executed. The Stretch computer [7] - [9] involved an attempt to partially overcome this problem by allowing the programmer to guess at the path that will be taken. This sequence of instructions was fetched and dispatched. If the guess is later determined to be wrong, the system must abort and reconstruct the state of the machine back at the branch. This process required considerable effort so that there was no improvement in execution speed unless the guess was correct 90% of the time. The result was an expensive failure. The 360/91 and 360/195 avoided this difficulty by pre-fetching the two instruction sequences but no execution until the conditional is resolved [10] - [11]. We could discover no one who has proposed deeper excursions into the undecided future.

This paper is a continuation of the examination that Tjaden and Flynn [6] started. The absolute limit of the payoff of the decode stack will be investigated by considering a hypothetically infinite decode stack. First, the concept of a program executing with maximum parallelism is developed; this is a program in which it is assumed that it is known a priori how all conditional branches will be resolved. Then the analysis will be modified to treat conditionals in a realistic manner. An algorithm will also be described to replace the look-ahead hardware of the stack by reordering the sequence of instructions prior to execution. We refer to this process as "percolation". The transformed sequence will have the property that if the instruction at the top of the stack cannot be dispatched immediately,

II. MAXIMUM PARALLELISM

Consider the stream of instructions presented to the control unit of a conventional CPU. There will be loads, stores, adds, multiplys, unconditional and conditional jumps etc. Examples of such streams may be collected by tracing actual programs with a suitable interpreter. What factors limit the rate of execution of such an instruction stream?

In the simplest type of CPU, the time required to fetch instructions and operands will limit the rate. Let us add a very large (unlimited) stack or cache to the machine so that, for all practical purposes, memory access time goes to zero. Still the program takes a finite non-zero time to execute. This is because it consists of a sequence of instructions each consuming some time. Let us, therefore, allow as many instructions to be executed in parallel (at the same time) as we can. Since at any given moment we may wish to have several adds going on or several multiplys, let us expand the CPU so it has very many (as many as necessary) functional units. That is to say we will never delay the dispatching of an instruction because of lack of a piece of hardware; thus each instruction will be dispatched at the earliest possible moment.

We still do not achieve complete parallelism because of the inherently sequential nature of parts of the instruction stream. For example, the triplet: "load accumulator, add, store accumulator" must be executed sequentially. Each instruction in a computer (with the exception of no-ops) has a set of sources and a set of destinations, not necessarily disjoint. Thus:

Instruction	Meaning	Sources	Destinations
LDA α	load acc.	memory location α	acc.
ADD β	add to acc.	acc. and memory location β	acc.
STA γ	store acc.	acc.	memory location γ

The registers (both "high speed" and "storage") containing the information that will be needed during execution of the instruction will be referred to as the sources of the instruction; the registers that must be available to store the results of the instruction are called the destinations. An instruction that has a destination which is not at the same time a source with respect to that instruction is said to be an <u>open effects</u> on that destination [6],[10]. Thus, LDA is open effects on the accumulator and STA δ is open effects on δ.

In keeping with our previous approach we will say that each open effect creates a <u>new</u> destination. Under this assumption consider the following string of instructions:

$$\left.\begin{array}{l}\text{LDA } \alpha\\ \text{ADD } \beta\\ \text{STA } \gamma\end{array}\right\}\ \text{Chunk 1}$$
$$\left.\begin{array}{l}\text{LDA } \delta\\ \text{MPY } \varepsilon\\ \text{STA } \pi\end{array}\right\}\ \text{Chunk 2}$$

This set of instructions could be executed in any of three ways:

a. as shown - first chunk 1, then chunk 2

b. in parallel - chunk 1 at the same time as chunk 2

or c. in reverse order - first chunk 2, then chunk 1

The parallel execution, case b), can take place because the LDA δ and the LDA α are both open effects on the accumulator and each creates a new accumulator for use by that chunk independently of the one used by the other.

We still have a limit on the speed of the program. Clearly, we cannot dispatch (begin to execute) an instruction until all its sources are available. The ADD β instruction above must await the completion of the LDA α and whatever instruction it was that put data into β. Even after all its sources are available the ADD will take a non-zero time to execute.

We are considering a program not as a set of instructions that must be executed in any particular order; rather, we will be looking at a program after it has executed as the set of instructions executed, each one having some earliest dispatch time. Thus, the sequentially executed program is just one possibility in which many instructions are not dispatched as early as they can be. The program executed with the maximum parallelism (minimum execution time) is the one that dispatches each instruction at the earliest moment.

There still remains the difficulty of the conditional branch. Should an instruction be dispatched before a conditional is resolved since it is not known whether the instruction will be executed? For the development of the concept of maximum parallelism, we are looking at the program a posteriori, after it has executed. Under these conditions, we know in advance how every conditional instruction, in the entire sequence of instructions executed, is resolved. Certainly this is an absurd assumption but it will determine the limit on the payoff due to parallelism by any of the hardware that was discussed earlier. In effect, a conditional instruction is being viewed as nothing more than another instruction whose execution is dependent upon certain information.

We will examine, in a more formal fashion, the constraints upon the dispatching of an instruction. The set of sources and destinations of the i^{th} instruction, I_i (numbered in the sequence in which they were originally executed), will be denoted by S_i and D_i, respectively. There are three ways in which the i^{th} instruction can involve the j^{th} register, r_j:

1. r_j is a source but not a destination ($r_j \epsilon S_i, r_j \not\epsilon D_i$). The contents of r_j will be used but not changed by I_i.
2. r_j is both a source and a destination ($r_j \epsilon S_i, r_j \epsilon D_i$). The contents of r_j are used and changed by I_i. Thus, the new contents of r_j are dependent upon the old contents of r_j.
3. r_j is a destination but not a source ($r_j \not\epsilon S_i, r_j \epsilon D_i$). The contents of r_j are changed by I_i independently of the previous value stored in r_j. This case is the opened-effects mentioned earlier.

It is clear that an instruction I_i cannot begin execution until all its sources are available. Additionally, I_i cannot complete execution until its destinations are free (storing information that will be required no longer). The following assumption is not unreasonable and shall be made: I_i cannot be <u>dispatched</u> until all its sources and destinations are available.

Let Td_i = earliest dispatch time of I_i,

$T(S_i)$ = max (times at which sources in S_i are available)

$T(D_i)$ = max (times at which destinations in D_i are available)

Then, $Td_i = \max [T(S_i), T(D_i)]$

It is possible to improve on this dispatch time if additional resources are allocated in the open-effects case cited earlier. It is possible that the lack of availability of a register as a destination can delay the dispatching of an instruction. All delays due to open-effects instructions can be removed by providing as many copies of registers as are needed, both high speed and memory registers. Since we are developing an upper limit on parallel execution speed here, we will assume that we have infinite such resources. Now, the program exhibiting maximum parallel execution is the one that dispatches each instruction the moment its sources are available; $Td_i = T(S_i)$ for all I_i in the program. The program is finished when all instructions complete execution.

The machine that was simulated by Tjaden and Flynn [6] was an IBM 7094 with the following characteristics:

a. there is a stack of finite length

b. there are infinitely many copies of the special registers such as the accumulator.

c. no instruction will be dispatched earlier than any conditional instruction preceding it in the stack.

The machine that has been described here and which was simulated is a CDC 3600:

a. there is a stack of infinite length; i.e. the entire sequence of instructions is in the stack at once.

b. there are infinite resources; thus, there are infinitely many copies of both special registers and storage registers, and no instruction is delayed due to the availability of any hardware.

c. conditionals do not block the dispatching of other instructions since the manner in which they will resolve is assumed to be known.

III. EFFECT OF STACKS AND SOFTWARE PERCOLATION ON PARALLELISM

Blocking on Conditional Jumps

In the previous section, we defined the concept of maximum percolation which, of course, can never be reached in practice. Each of the three assumptions about the hypothetical machines are unrealistic. Stack sizes are finite and functional units are limited in number, as are central registers and memory locations. An even more severe limit is the effect of conditional branching on the parallel execution of instructions. In the above, we were looking at traces of instruction streams, at the a posteriori history of a program. There, the choice of which path to take from a conditional jump was already made. But in reality when we come up to a choice point in an instruction stream (a conditional branch) we don't know which of the two possible paths the program is going to take until the data on which the choice is to be made (the sources of the conditional jump) become available and the instruction is actually executed - until the conditional is resolved.

Suppose we decide to accept this limitation. Then no instruction can be dispatched for execution until all conditional jumps preceding it have been resolved and its own sources are available. It is a trivial matter to modify the previous analysis to handle conditionals in this fashion. Modifying the dispatch time of an instruction, so that it is blocked by the last conditional, we have

$$Td_i = \max\,[T(S_i),J_i]$$

where J_i = completion time of the last conditional jump preceding I_i.

PARALLEL DECODE STACKS

Two factors that limit the length of the decode stack involve the look-ahead hardware: the square law increase of the circuitry and the increasing delay due to the increasing number of logical levels as fan-in and fan-out limits are exceeded.

The look-ahead hardware in the decode stack is necessary to determine dependencies in the instruction stack. If an instruction has a source that is a destination of an instruction above it in the stack, this instruction must be delayed until the instruction it is dependent upon completes execution. Similarly, if single copies of registers are all that is available, an instruction must be delayed if it has a destination that is a source of an instruction above it. However, suppose the instructions are reordered into the sequence in which they become ready to be dispatched. If we assume that the time it takes to dispatch an instruction is zero, the next instruction which is to be dispatched will be at the top of the stack. If the top instruction of the stack is not ready, there is no need for look ahead hardware to see if any other instruction is ready; all other instructions in the stack must have an earliest dispatch time that is greater than or equal to the top one. Thus, if the dispatch time of an instruction can be kept negligible, the equivalent of an infinite decode stack can be obtained by using the following algorithm.

SOFTWARE PERCOLATION

The following algorithm determines the time at which each instruction can be dispatched. It is then a simple matter to reorder instructions by the size of these values. One can think of this as letting instructions percolate by each other until they reach instructions they are dependent upon.

At any point in the execution of a sequence of instructions, say when I_i is executing, let each register, r_j, in the machine have a last changed time, $C(r_j)$. This time will be defined to be the last time at which the contents of register r_j were altered. This is equivalent to the last time that r_j appeared as a destination. Therefore, the time at which all sources become available for I_i is

$$Td_i = T(S_i) = \max (C(r_j) | r_j \varepsilon S_i).$$

Each instruction has its own execution time, Te_i, which depends on the type of the instruction. The time at which an instruction finishes execution is termed the completion time, Tc_i; it is determined by $Tc_i = Td_i + Te_i$. This completion time defines the time at which the destinations D_i become available as sources for any instructions appearing after I_i in the instruction sequence. We will store all these values and keep updating them in a "last changed" table. Hence, all registers in D_i have their entries in the last changed table updated with Tc_i: $C(r_j) = Tc_i$ for all $r_j \varepsilon D_i$. In addition, the completion time of the last conditional branch will be saved and updated whenever a new conditional branch is encountered. Modifying the dispatch time of instructions to block on conditionals, we have $Td_i = \max [T(S_i), J] = \max [\max[C(r_j) | r_j \varepsilon S_i], J_i]$.

Thus, by making one pass on the sequence of executed code, updating one table, and renaming registers when necessary, the dispatch times of all instructions are determined. There is one significant difficulty; the preceding discussion assumed that the entire sequence of instructions is available beforehand. All conditionals are not resolved prior to execution. Therefore, if this algorithm is to be applied to code in a meaningful way, it should be applied to the static code as it is written. This implies that an instruction in the original sequence of code cannot be percolated by any conditional that appears before it in the code. Thus, blocking on conditionals, we can think of the sequence of instructions as a set of segments, each segment being the condition instruction and the code that follows up to, but not including, the next conditional. One must only determine the dispatch times for the set of instructions between a pair of conditionals. The instructions within a segment can then be statically reordered into the sequence in which they would be dispatched dynamically by the decode stack if one difficulty is overcome. In the case of the stack, instructions are being fetched and the dispatching order is determined dynamically. Suppose the case shown in Figure 1(a) is encountered.

Even under the assumption of a zero dispatching interval, the percolation algorithm still only approximates an infinite stack. In a stack the conditional beginning segment 2 might be resolved before all the instructions of segment 1 have completed execution. This might allow the instructions in segment 2 to be dispatched in parallel with the remaining instructions of segment 1. However, the percolation algorithm must reorder instructions prior to execution.

Thus, the conditional must not be percolated ahead of any instructions in segment 1 lest the logic of the program be altered. Consequently, this reordering will prevent the partial parallelism of segments 1 and 2 when the second conditional branch could be resolved early. Nevertheless, it appears that much of the parallelism should be realized.

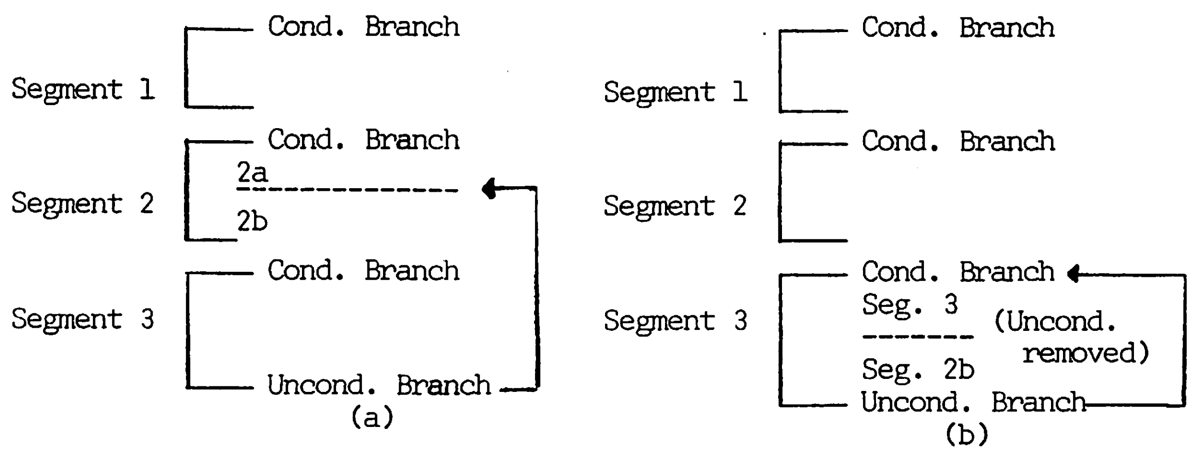

FIGURE 1

When segment 2 is in the decode stack, instructions from parts 2a and 2b can be dispatched simultaneously and in a reordered sequence. Similarly, when segment 3 and the unconditional are encountered, segment 2b can be fetched and the instructions dispatched and executed in parallel with those of segment 3. Now examine the static percolation algorithm. Instructions in segment 2 may be reordered but then the program may not be logically correct because segment 2b is no longer defined to follow segment 3; parallelism between segments 2b and 3 is also prevented. Figure 1(b) shows that this is overcome by duplicating the code of 2b so that complete parallelism in 2a and 2b as well as in 3 and 2b can take place. Consequently, with some modest increase in the size of the code one can remove the look-ahead hardware.

EXPERIMENTAL RESULTS

Data was collected by tracing seven programs which included both compiled code and hand-generated code and amounted to almost 2 million instructions. The compiled code consisted of the object code of three Fortran programs to calculate means and variances (BMO1D), analyze patterns of symbols in text strings (CONCORDANCE), and eigenvalues of matrices (EIGENVALUE). The hand coded programs included the COMPASS assembler for the CDC-3600 translating a short program, the FORTRAN compiler translating a short program, analysis of patterns of op-codes in programs (DECALIZE), and finally our interpreter itself (INTERIT).

First, we examined the increase in the execution rate as a function of stack size. The amount of parallelism will be defined to be the ratio of the normal sequential execution rate to the parallel execution time. Thus, if on the average, two instructions are executing at the same time, the parallel execution time would be half that of the sequential time, and the parallelism would be 2. Figure 2 is a graph presenting the relationship of increasing stack size on parallelism with the limit being the maximum parallelism achievable. All of these results were derived knowing how all conditionals would resolve and thereby allows us to examine the effect of stack size as an independent factor. One can see that the size of the stack is a critical factor in determining the amount of parallelism gained and that fairly large stacks are necessary to obtain all the potential parallelism.

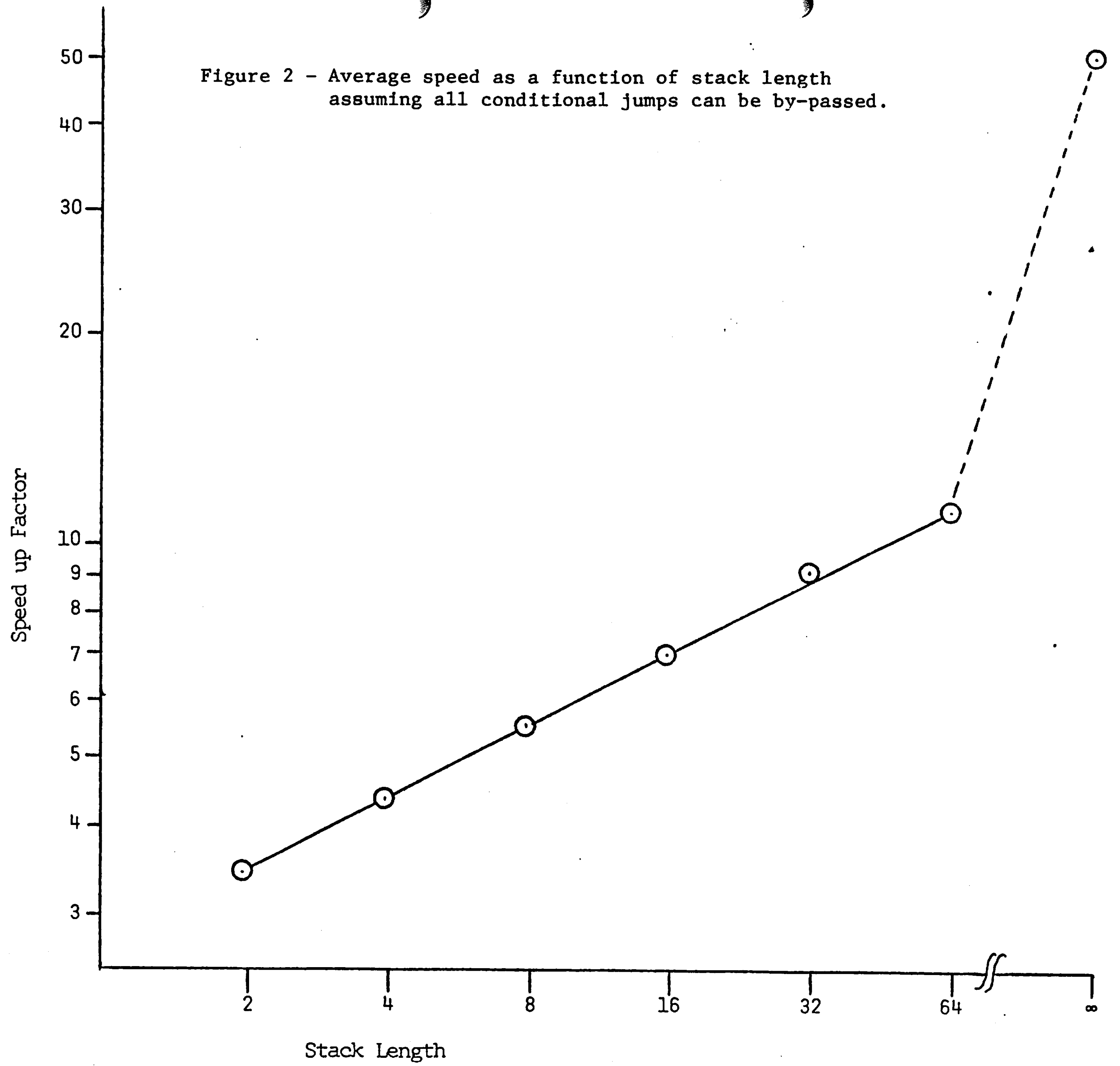

Figure 2 - Average speed as a function of stack length assuming all conditional jumps can be by-passed.

Table 1

Parallelism as a Function of Various Stack Sizes

Stack Sizes

	2	4	8	16	32	64	∞
BMD01	1.368	1.401	1.417	1.429	1.430	1.431	1.431
CONC.	1.431	1.500	1.516	1.523	1.527	1.527	1.527
EIG.	1.545	1.655	1.695	1.710	1.720	1.722	1.722
COMPASS	1.208	1.215	1.219	1.220	1.220	1.220	1.220
FTN. COMP.	1.376	1.390	1.392	1.392	1.392	1.392	1.392
DECALIZE	1.610	1.708	1.752	1.775	1.781	1.781	1.781
INTERIT	2.440	2.648	2.882	2.975	2.975	2.975	2.975
AVE.	1.568	1.645	1.696	1.718	1.721	1.721	1.721

Table 2

Stack sizes necessary to achieve various percentages of the parallelism in an infinite stack.

	100%	99%	90%
BMD01	64	8	2
CONC.	32	8	2
EIG.	64	16	4
COMPASS	32	2	2
FTN. COMP.	>64	4	2
DECALIZE	32	16	4
INTERIT	16	16	8

With instructions blocking on conditionals, Table 1 presents the resultant parallelism for stack sizes varying from 0 to 64. The parallelism that would be obtained from an infinite size stack is also included. This data is not graphed in Figure 2 because of the very slight increase in parallelism as a function of stack size. On the average, a sizeable portion of the parallelism obtained with an infinite stack is realized by a stack of size two; almost all the parallelism is obtained by a stack of size 8. Stack sizes that would be necessary to achieve 90, 99 and 100% of the parallelism of an infinite stack are given in Table 2.

All of the experimental results just discussed were carried out under the assumption that there are as many extra copies of all types of registers as needed. Experiments were run to determine whether this was a necessary assumption by examining the effect upon parallelism of limiting registers to a single copy each. There was an insignificant decrease in the resultant parallelism if memory registers were limited to a single copy each. If the number of special high speed registers (A, Q and D registers in the CDC 3600) are limited to a single copy of each type, the parallelism was reduced by slightly more than 10% with a stack of size 32.

DISCUSSION AND CONCLUSIONS

This section has examined the use of a decode stack which dispatches each instruction as early as possible. Under the constraint that conditional branches delay instructions until they are resolved, very little of the maximum parallelism is obtained. The limit on the parallelism that is achieved with an infinitely large stack was found to be slightly more than 1.7; this means that the usual sequential machines would take 70% longer to execute the set of 7 test programs than this parallel machine. These results are somewhat worse than those given by Tjaden & Flynn [6], 86% for a stack size of 10. Most of the parallelism is achieved with very short stacks. In all but one case, a stack size of 4 would achieve 90% of the parallelism of an infinite stack. Little parallelism is gained by supplying extra copies of registers. These results imply that parallelism between conditional branches is quite limited in the object and hand code of typical programs run on current machines.

Under the assumption of a zero dispatching interval, we have determined the upper bound on the parallelism derived for various stack sizes. One should note the implications of this assumption. Suppose we have 10 instructions in a row which could be dispatched and executed in parallel and a stack of size one: all the instructions would be executed in parallel because they would be sequentially brought into the stack and dispatched in zero time. Nevertheless, it is a desirable goal to approach this assumption by decreasing the dispatch time and we do determine a limit on the parallel execution speed in this structure. Given the relatively small amount of parallelism obtained even with zero dispatch time, this assumption does not appear to affect any of the conclusions.

One may still feel that there are cases in which the addi-

Table 3

The speed of percolated programs relative to the speed of unpercolated programs with an infinite dispatching stack.

Program Name	Relative Speed
BMD01	.955
CONC.	.947
EIG.	.835
COMPASS	.978
FTN. COMP.	.955
DECALIZE	.937
AVERAGE	.935

tional expense that is required to achieve this parallelism is justified. This paper has described an alternative to achieving this parallelism strictly in hardware. The percolation algorithm presented in this paper approximates the decode stack by reordering instructions prior to execution. This method achieves 93.5% of the parallelism of an infinite stack; results for individual programs is presented in Table 3. Thus, one can effectively replace the hardware stack by additional processing during compilation.

The critical factor in the limitation of parallelism is not the stack size or multiple copies of functional units and registers. Rather, the limiting factor to be focussed upon is the problem of conditional branches.

IV. EFFECT OF BYPASSING CONDITIONAL JUMPS ON PARALLELISM

Since it has been established that conditional jumps inhibit parallelism, let us consider ways to overcome this problem. Suppose we built a machine which could "get by" one conditional jump by beginning execution down both paths leading out of the jump. Once the jump is resolved the untaken path is discarded. Conditionals that can be decided on the spot cause no complications since they have only one path of successors. Thus, we can keep going down at most two paths. Such programs may be said to "by-pass" one conditional jump.

Let us generalize this concept so that up to j conditional jumps may be unresolved along the ancestoral path of an instruction. We define $L^j\{x\}$ to be the j+1th largest element of the set x. For example,

$$L^0\{1,2,3,4,5\} = 5$$
$$L^1\{1,2,3,4,5\} = 4$$
etc.

Let the set of completion times of all conditional jumps preceding* the execution of the ith instruction be J_i, then the earliest possible completion time of the ith instruction will be

$$T_i^0 = E_i + \text{Max}\,\{S_1, S_2, \ldots, L^0\{J_i\}\}$$

where the superscript 0 on T indicates that no conditional jumps are bypassed. Using this algorithm we can compute the running time, R, of a program which blocks on all conditional jumps to be:

$$R = \underset{i}{\text{Max}}\,\{T_i^0\}\ .$$

*"preceding" refers to the original code - as it would be executed by a one-at-a-time machine.

That is, the running time will be equated to the completion time of the last instruction completed. Expanding to bypass j conditional jumps

$$T_i^j = E_i + \text{Max}\,\{S_1, S_2, \ldots, L^j\,\{J_i\}\}$$

and

$$R_j = \text{Max}\,\{T_i^j\}$$

where R_j is the running time of a program on an infinite machine which can by-pass j conditional jumps. One should note that the number of paths that must be maintained may be as large as 2^j, if the program can by-pass j conditional jumps. Since the complexity of a CPU must grow at least linearly with the number of paths maintained, we hope to find dramatic improvements in speed for small j, since even a j as small as 8 implies up to 256 paths executing simultaneously.

OUR EXPERIMENT

We traced seven programs written for the CDC-3600. These included compilers, compiled code, hand generated code, numeric programs and symbol manipulating programs. A total of 1,884,898 instructions were traced representing very nearly seven seconds of real 3600 time. We found no significant differences between hand and compiler generated code, nor between numeric and symbolic programs. Since the analysis of these seven programs consumed some forty hours of machine time, we decided to bring the data collection phase of our studies to a halt.

The seven programs we traced were as follows:

1. BMD01D. A FORTRAN program for the calculation of means and

variances.

2. CONCORDANCE. A FORTRAN program written to analyze text strings for repetitions of patterns of symbols.
3. EIGENVALUE. A FORTRAN program to compute EIGENVALUES of matrices.
4. COMPASS. The COMPASS assembler itself translating a short program. An example of hand-coded symbol manipulation.
5. FORTRAN. The FORTRAN compiler itself translating a program. Another example of hand-coded symbol manipulating program.
6. DECALIZE. A hand-coded program to analyze patterns of op-codes up to ten-tuples.
7. INTERIT. Our interpreter itself. Hand-coded.

Since we had to choose some set of execution times, we chose those of the 3600 itself. Table 4 shows that their ratios are not far from the 360/91 or the CDC 6600, two of the fastest computers currently available.

Tjaden and Flynn [6] showed that for code written for the 7090, a relative improvement of 1.86:1 could be achieved with a stack length of eight and blocking on all conditional jumps. This was considerably less than the 51:1 improvement found with maximum percolating so we decided to let the stack length (and other parameters) go to infinity and examine the effects of getting by various numbers of conditional jumps.

For zero jumps by-passed, we found an average improvement of 1.72 to 1 (see Figure 3 and Table 5). That is, the average program ran 1.72 times as fast with infinite stack, infinite registers, infinite

TABLE 4

Relative Speed of Various Instructions in Various Machines

Fixed Point Add Taken as Unity for each Machine

Instruction	CDC-3600	IBM-360/91	CDC-6600
Fixed Add	1	1	1
Fixed Multiply	3-4	7-11	no such inst.
Fixed Divide	7-8	36-37	no such inst.
Floating Add	2-3	2	1.3
Floating Multiply	3-4	3	3.3
Floating Divide	6-7	4	9.6

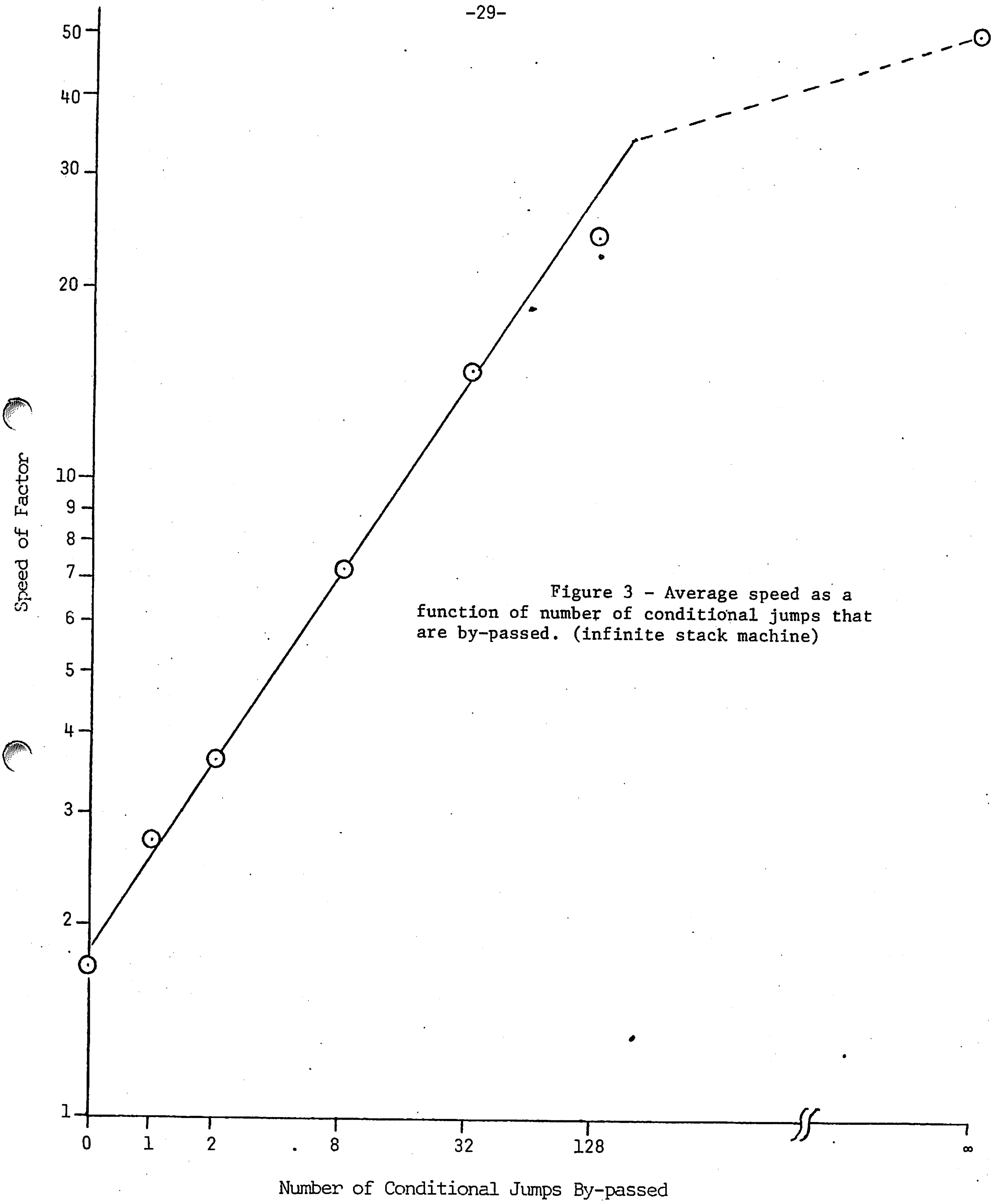

Figure 3 - Average speed as a function of number of conditional jumps that are by-passed. (infinite stack machine)

fig3

TABLE 5

Speed Up of Seven Programs as a Function of the Number of Conditional Jumps Passable

(Speed up is defined as the average number of instructions being executed in parallel)

Program	0 jump	1 jump	2 jumps	8 jumps	32 jumps	128 jumps	∞ jumps
FORTRAN	1.40	2.03	2.38	3.14	4.02	5.86	32.4
COMPASS	1.22	2.10	2.74	4.28	5.55	7.17	27.2
CONCORDANCE	1.53	2.27	3.45	8.50	20.20	47.30	100.3
INTERIT	2.98	5.11	6.60	15.10	36.70	37.70	39.8
EIGENVALUE	1.72	2.40	3.34	6.64	14.20	22.40	29.7
DECALIZE	1.79	2.76	3.44	5.23	6.15	6.53	7.8
BMD01D	1.43	2.38	3.32	7.56	16.80	43.50	39.5
AVERAGE	1.72	2.72	3.62	7.21	14.8	24.4	51.2

storage, infinite functional units as it did in an ordinary everyday 3600. Clearly, conditional jumps were preventing any substantial amounts of parallelism. If we allow by-passing of one conditional the average program runs 2.72 times as fast as when run sequentially.

From here on out, the relative speed increases as the $\sqrt{j}$ where j is the number of jumps by-passed. That is, if we by-pass four jumps, the program runs twice as fast as if we by-pass only one jump. Similarly, sixteen jumps by-passed is twice as fast as four jumps. The square root relation holds quite well up to 32 jumps (some four billion paths). We have no theoretical justification of this relationship at the present time.

DISCUSSION

The relative speed of execution goes up only as the square root of j and the number of paths that must be maintained simultaneously goes up as 2^j. Hence, it does not appear that this is the proper approach to designing general purpose high speed CPU's unless minimum response time is the only criterion considered.

Naturally, the reader may be concerned with the fact that the code we examined was written for a sequential machine and not a parallel one. But we have provided for as much renaming as is necessary and, aside from recasting the algorithm completely, the only real improvement that could be made would be to eliminate conditional jumps. But Flynn [12] has mentioned an unpublished study in which fewer than half of the conditional jumps were removable even after extensive hand tailoring.

One mechanical aid in this direction would be to implement a repeat instruction for those loops where the number of iterations is known before entry (non-data dependent exits) which would not be "conditional" in the normal sense of the word.

In a very brief examination of this approach, we flagged all the "loop ending jumps" in BMD01 as being "non-conditional" and reran the program on our hypothetical machine with infinite resources but blocking on conditional jumps. We found that with DO-loop generated jumps eliminated, it ran almost exactly 1% faster than with them left in. Thus, we conclude on the basis of this experiment, that this approach does not appear to offer much help.

We, then, decided to find out how long a stack would be required to reach the theoretical speed-up of 51 times if we ignored the problem of conditional jumps.

Figure 2 and Table 6 show the average speed up of our seven programs as a function of decode/dispatch stack length under the assumption that any number of conditional jumps may be by-passed. The important things to be noted in Figure 2 are, first, that even with a stack length as short as 2 by-passing all conditional jumps allows a program to run twice as fast as if it had an infinite stack and blocked on conditionals (see below). Second, we should note that even with a stack of length 64, we are still a factor of 4 slower than an infinite stack. This implies that instructions must percolate a long distance (past more than 64 instructions) in order to achieve maximum speed.

Table 6

Speed Up of Seven Programs as a Function of the Length of the decode/dispatch Stack when all Conditional Jumps are Passable.

Program	Stack Length					
	2	4	8	16	32	64
FORTRAN	2.44	2.71	2.81	3.26	3.63	4.08
COMPASS	3.24	3.78	4.00	4.59	5.06	5.64
CONCORDANCE	4.22	6.50	9.33	11.95	15.80	20.0
INTERIT	4.43	5.63	7.39	10.59	15.80	24.8
EIGENVALUE	2.46	3.17	4.16	5.46	7.94	11.91
DECALIZE	2.66	3.46	4.33	4.85	5.28	5.78
BMDO1D	4.70	5.54	6.59	8.30	10.91	15.20
AVERAGE	3.45	4.40	5.52	7.00	9.20	12.49

CONCLUSIONS

Lurking within an average program, there is a potential parallelism of 50:1. Even given all the resources it might conceivably need, this average program will be severely inhibited by the presence of conditional jumps. Limiting ourselves to by-passing no more than two conditionals, we can extract less than a 4:1 improvement in speed. To run ten times as fast as a one-instruction-at-a-time machine, we need to get by sixteen jumps. This implies 64K paths being explored simultaneously. Obviously, a machine with 65,000 instructions executing at once is a bit impractical.

Should other studies confirm our conclusions, it would appear that the way to gain higher speed CPU's is by the brute force technique of making each instruction run faster; not by providing look ahead stacks of multiple functional units.

The only alternative seems to be tailoring of algorithms to take advantage of the parallel machine or the use of costly compilers.

References

[1] H. Hellerman, "Parallel Processing of Algebraic Expressions", IEETEC, February, 1966, pp. 82-91.

[2] H.S. Stone, "One-Pass Compilation of Arithmetic Expressions for A Parallel Processor", Communications of ACM, April, 1967, pp. 220-223.

[3] M.E. Conway, "A Multi-processor System Design", 1963 Proceedings of FJCC, pp. 139-146.

[4] G.H. Barnes, R.M. Brown, M.Kato, D.J. Kuck, D.L. Slotnick & R.A. Stokes, "The ILLIAC IV Computer", IEEETC, August, 1968, pp. 746-757.

[5] M.J. Flynn, "Very High Speed Computing Systems", Proceedings of IEE, December, 1966, pp. 1901-1909.

[6] G.S. Tjaden & M.J. Flynn, "Detection & Parallel Execution of Independent Instructions", IEETC, October, 1970, pp. 889-895.

[7] E. Bloch, "The Engineering Design of the Stretch Computer", 1959, Proc. Eastern Joint Computer Conference. pp. 48.

[8] R.T. Blosk, "The Instructions Unit of the Stretch Computer", 1960, Proc. Eastern Joint Computer Conference, pp. 299-325.

[9] J. Cocke and H.G. Kolsky, "The Virtual Memory of the Stretch Computer", 1959, Eastern Joint Computer Conference, pp. 82-94.

[10] R.M. Tomasulo, D.W. Anderson & F.J. Sparacio, "The Model 91: Machine Philosophy & Instruction Handling", IBM Journal of Research & Development, Vol. 10., November, 1966.

[11] D.W. Anderson, "The IBM System/360 Model 91", IBM Journal of Research & Development, Vol. 11, January, 1967.

[12] M.J. Flynn, Personal Communication with the Authors.