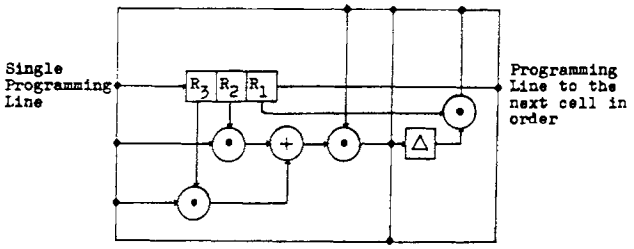

Fig. 8. A linear select programmable cell structure.

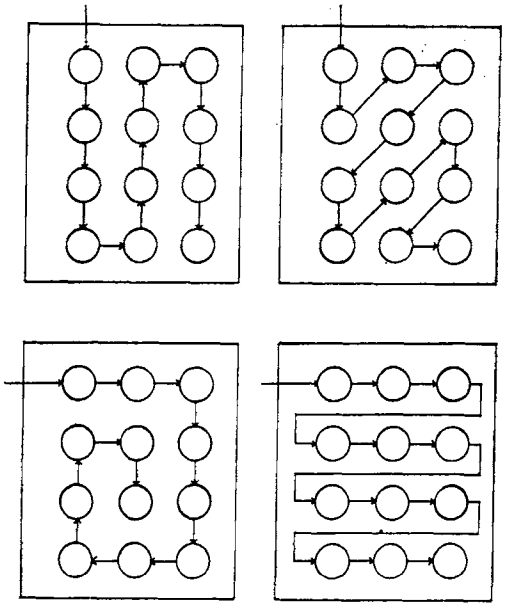

Fig. 9. Some possible arrangements of the programming bus in linear select scheme.

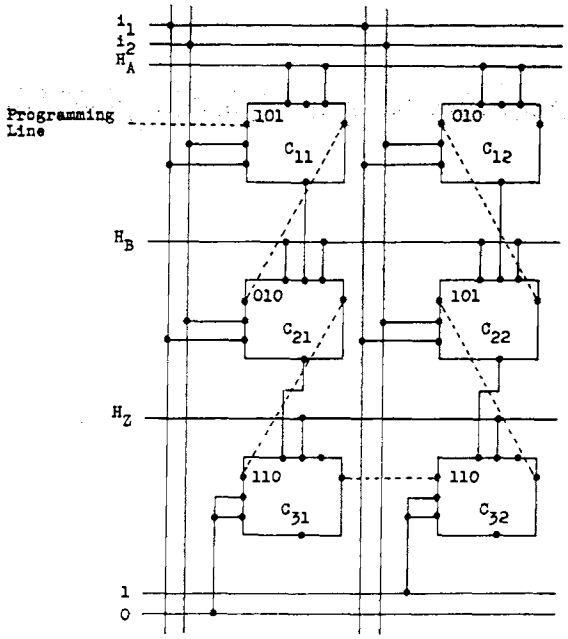

Fig. 10. A linear select programmable cellular array.

and q outputs can realize any n or less state sequential machine with p or less inputs and q or less outputs.

Conclusions

The purpose of this study has been to develop cellular methods of synthesizing sequential machines. From the production point of view, cellular array employing LSI technology represents a great saving in cost, space, and weight. From the designer's point of view, cellularization does away with the arduous task of state assignment. A further advantage of cellularization lies in its programmability. This is important in designing reconfigurable digital systems.

Acknowledgment

The author wishes to thank Dr. R. A. Short for inspiring interest in this area and for his helpful guidance throughout this work.

References

[1] K. K. Maitra, "Cascaded switching networks of two-input flexible cells," *IRE Trans. Electron. Comput.*, vol. EC-11, pp. 136–143, Apr. 1962.
[2] R. A. Short, "Two-rail cellular cascades," in *1965 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 27. Montvale, N. J.: AFIPS Press, 1965, pp. 355–369.
[3] R. C. Minnick, "Cutpoint cellular logic," *IEEE Trans. Electron Comput.*, vol. EC-13, pp. 685–698, Dec. 1964.
[4] ——, "Survey of microcellular research," Stanford Res. Inst., Menlo Park, Calif., Rep. AFCRL-66-475, 1966.
[5] R. A. Short, "The attainment of reliable digital system through the use of redundancy—A survey," *Comput. Group News*, vol. 2, pp. 2–17, 1968.
[6] R. H. Wilcox and W. C. Mann, Ed., *Redundancy Techniques for Computer Systems*. Washington, D. C.: Spartan, 1962.

The Inhibition of Potential Parallelism by Conditional Jumps

EDWARD M. RISEMAN AND CAXTON C. FOSTER

***Abstract*—This note reports the results of an examination of seven programs originally written for execution on a conventional computer (CDC-3600). We postulate an infinite machine, one with an infinite memory and instruction stack, infinite registers and memory, and an infinite number of functional units. This machine will execute a program in parallel at maximum speed by executing each instruction at the earliest possible moment.**

Manuscript received March 24, 1972; revised June 21, 1972. This work was supported in part by a study grant from Control Data Corporation.

The authors are with the Department of Computer and Information Sciences, University of Massachusetts, Amherst, Mass. 01002.

The manner in which conditional jump instructions are treated is the primary concern of this note. One possibility is to assume that when a conditional jump is encountered, no further instructions may be issued until that condition is resolved and the subsequent path is determined. Under this assumption, the seven programs, even on this infinite machine, ran only 1.72 times as fast as they did on a conventional machine. On the contrary, if it is assumed that one knows in advance which path will be taken at each branch, conditional jumps do not impede the execution of the program. This results in the program running 51 times as fast as in a conventional machine. The implications of these results are discussed.

Index Terms—**Conditional jumps, CPU design, execution speed, multiple functional units, parallelism, pipelining.**

Introduction

Consider the stream of instructions presented to the control unit of a conventional CPU. There are loads, stores, adds, multiplies, 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 nonzero 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 additions and several multiplies executing concurrently, let us expand the CPU so it has very many (as many as necessary) functional units. That is to say, the dispatching of an instruction is never delayed because of lack of a piece of hardware.

Complete parallelism is still not achieved 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. This condition occurs because the add instruction needs the information fetched by the load and the store instruction needs the sum computed by the add. More formally, we may say that each instruction has a set of "sources" on which it depends and a set of "destinations" which it modifies.

For the above triplet we have the following.

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

When an instruction has a destination that is not at the same time a source for that instruction, we will say that it "redefines" that destination. Thus, LDA redefines the accumulator and STA δ redefines δ. Tjaden and Flynn call this "open effects" [5].

In keeping with our previous approach, we will say that each redefinition creates a *new* destination. Under this assumption consider the following six instructions.

LDA α
ADD β } strand 1
STA γ

LDA δ
MPY ϵ } strand 2.
STA π

This set of instructions could be executed in any of three ways: 1) as shown—first strand 1, then strand 2; 2) in parallel—strand 1 at the same time as strand 2; or 3) in reverse order—first strand 2, then strand 1.

The parallel execution (case 2) can take place because the LDA δ and the LDA α are both redefinitions of the accumulator, and each creates a *new* accumulator for use by that strand independently of the one used by the other.

There is still a limit on the speed of the program. Clearly, an instruction cannot be dispatched (begin to execute) until all its sources are available. The ADD β instruction above must await the completion of the LDA α and the availability of the data in β. Even after all its sources are available, the ADD will take a nonzero time to execute. To assume otherwise would imply that all programs, regardless of their length, run in zero seconds. Stating this somewhat more formally, we have

$$T = E + \max(S_1, S_2, \cdots, S_k)$$

where

- T Completion time of the instruction and hence the time at which its results (destinations) become available as a source for further instructions.
- E Execution time of the instruction.
- S_j Time at which source j becomes available.

We will say that a program is running at "maximum speed" when the following hold.

1) Each instruction is dispatched as soon as its sources become available.

2) There exist sufficient resources in the machine so that no execution of an instruction is delayed by lack of required resources.

3) Conditional jump instructions do not impede the flow of the program because either of the following is true.

a) One somehow knows *a priori* which path will be taken from a branch point and can proceed only down

that path (in which case the branch could effectively be removed).

b) Many tentative computational paths can be maintained simultaneously, with the eventual selection of the correct path and the discarding of the incorrect paths taking place as conditional jumps become resolved.

Thus, a program is running at maximum speed when the only remaining constraints on its speed are the execution times of the various instructions and any inherent sequential dependencies between them. Note that the problem is not necessarily being solved at the maximum possible speed. A different algorithm or more efficient coding might run much faster than the program being used. By rewriting the algorithm for the type of machine being discussed, or even allowing redundant computations if the resources are available, a serial program might be modified to show a greater speed. This aspect has not been investigated in this note.

At various points in this note, we refer to an instruction stack or dispatch stack. This stack is similar to the predecode stack presented in [5]. It differs, however, in that it may be of infinite length;[1] the dispatching of instructions occurs as soon as they are ready rather than being clocked, and the decode and dispatch times are assumed to be zero. If the length of the stack is limited (as discussed in the latter part of this note), this last assumption may still allow instructions to be dispatched at an unbounded rate. Effectively, we have assumed that the dispatch time is a vanishingly small part of the execution time of any of the functional units. Therefore, new instructions can be brought in and dispatched in zero time until there is no room left in the stack to hold presently undispatchable instructions.

Let us consider in some detail the assumptions we have made concerning the decoding and dispatching mechanism. We assume first that there exist plenty of functional units and plenty of registers, and that the compiler or assembler has been clever enough to utilize these in such a way that there are no unnecessary contention problems. This is the "renaming" or "open effects" problem, and we hereby assume it out of existence.

Now in a conventional machine like the 360/91, the ith instruction in the stack is compared with the $(i-1)$th, the $(i-2)$th, and so on, to see if any of the sources of i are destinations of a previous instruction. It is this comparison hardware that increases as S^2 where S is the stack size. Comparisons are being done *in* the stack. Therefore, uncompleted instructions must be retained in the stack until they are completed in order that their presence there may inhibit later use of those registers they are in the process of changing. Under this assumption, there cannot be more instructions in execution than there are places in the stack to hold these uncompleted instructions; and, therefore, the maximum possible speedup is limited to the stack size. Our assumptions are somewhat different from this. The fact that they do not correspond to present-day techniques was pointed out by one of our referees.

We assume that when an instruction is loaded into the stack, it already specifies exactly which registers it needs as sources and which it expects to modify (its destinations). (Thus we ignore the problem of dynamic remapping of register names, although it does not seem as if this would be an insuperable design problem.) The dispatching hardware continuously monitors the state of all the source registers of this instruction, and when these sources all become "valid," it issues the instruction to some functional unit and simultaneously marks all of the destinations of this instruction as containing "invalid" data—data that are in the process of changing and hence are unusable. As the functional unit finishes its operation, it places the results it calculated in the destination registers and marks them as now containing valid numbers. Not all interlocking hardware can be eliminated from the stack, of course, for if we look at the following sequence of code,

1) LDA α
2) ADD β
3) MUL δ

we see that when the accumulator becomes valid for the first time, it means that instruction 2 may be dispatched, but not instruction 3.

However, these assumptions do mean that as soon as an instruction is issued (dispatched), it can be removed from the stack, making room for a new instruction to be inserted and analyzed. Thus, even with a very short stack (one or two slots), we are able to stream independent instructions through at a rate limited only by the decoding time, and achieve rates of parallelism that exceed the stack size. For convenience, we have assumed that the decoding time is vanishingly small (in fact, equal to zero). This assumption is entirely in keeping with our assumptions about unlimited registers and functional units.

The maximum speed of the seven programs we examined is shown in the last column of Table II (∞ jumps). The average of the maximum speeds is 51.2 times faster than their average speed on a conventional machine. As will be explained later, this maximum speed corresponds to bypassing an infinite number of conditional jumps.

Blocking on Conditional Jumps

In the previous section, the concept of maximum speed was defined, which, of course, can never be reached in practice. 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

[1] Actually, it never needs to be larger than the entire sequence of executed instructions.

a conditional jump was already made. But in reality, when a choice point in an instruction stream (a conditional branch) is encountered, it is not known which of the two possible paths the program is going to take until the data upon which the choice is to be made (the sources of the conditional jump) become available and the instruction is actually executed—that is, until the conditional is *resolved.*

Suppose this limitation is accepted. Then no instruction can be dispatched for execution until *all* conditional jumps preceding it have been resolved and its own sources are available. We define $L^j\{x\}$ to be the $j+1$th largest element of the set x. For example,

$L^0\{1, 2, 3, 4, 5\} = 5$ is the largest element of the set

$L^1\{1, 2, 3, 4, 5\} = 4$ is the second largest, etc.

Let the set of completion times of all conditional jumps preceding[2] the execution of the ith instruction be J_i; then the earliest possible completion time of the ith instruction will be

$$T_i^0 = E_i + \max\{S_1, S_2, \cdots, L^0\{J_i\}\}$$

where the superscript 0 on T indicates that no conditional jumps are bypassed. Using this equation, we can compute the running time R of a program that blocks on all conditional jumps to be $R = \max_i \{T_i^0\}$. That is, the running time will be equated to the completion time of the last instruction completed. The "speedup" of a program under a given set of conditions is defined to be the ratio of the running time on a conventional machine to the running time obtained under the given conditions. But suppose a machine is built that could "bypass" one conditional jump by beginning execution down both paths leading out of the jump. Once the *conditional jump* is resolved, the untaken path is discarded. Sometimes, when a conditional jump is reached, all the information necessary for its resolution will have already been computed, and it can be resolved at once. Conditionals that can be decided on the spot cause no complications, since they have only one path of successors. Thus the machine can keep going down at most two paths. Such programs may be said to "bypass" one conditional jump.

Let us consider the case of a machine that can bypass two conditional jumps. Let the first unresolved jump be called A if the jump is taken and $\overline{A}$ if not. We have two paths that must be explored. Suppose that, upon going down path A, another jump called B is encountered. If it is unresolved also, path A will be split into two paths: AB and $A\overline{B}$. Each of these paths may continue until they reach unresolved conditional jumps (D and E, respectively), at which point they must wait for the resolution of either A or B or D for path AB, and A or B or E for path $A\overline{B}$. But path $\overline{A}$ may proceed, and when it comes to an unresolved conditional jump C (not necessarily the same as B), it will split into two paths AC and $A\overline{C}$. Generalizing this concept so that up to j conditional jumps may be unresolved along the ancestral path of an instruction, we have

$$T_i^j = E_i + \max\{S_1, S_2, \cdots, L^j\{J_i\}\}$$

and

$$R_j = \max_i \{T_i^j\}$$

where R_j is the running time of a program on an infinite machine that can bypass 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 bypass j conditional jumps. Of course, these various paths may represent the same written instructions or different ones. For example, a loop ended by a conditional jump might generate the streams: iterate once and exit; iterate twice and exit; iterate three times and exit; etc. Since we do not know which is going to be the "real path" (in a real life situation), we must be prepared to explore all of them. It is clear that the number of possible paths can exceed the number of written instructions. 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.

Previous Work

The discussion presented above is by no means new. Hellerman [1] and Stone [2] have examined parallelism in higher level languages. Ramamoorthy and Gonzalez [3] review several methods of recognizing parallelism in programs. Flynn [4] pointed out in 1966 that dispatching of a single instruction per machine cycle was a serious bottleneck, and Tjaden and Flynn [5] examined the benefits of parallel execution in an IBM-7090 environment. The IBM STRETCH [6]–[8], aided by the programmer, guessed at which path from a conditional it should pursue, went ahead down that path, and then "backed up" if the guess was wrong. The IBM 360/91 and 195 do prefetching and decode of the two possible instruction paths but no execution beyond the conditional jump [9]. Stone [10] describes a machine that could proceed down two paths. We could discover no one who has carried out experiments on deep excursions into the undecided future of a program.

Our Experiment

Seven programs written for the CDC-3600 were traced. 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 7 s of real 3600 time. We found no significant differences between hand-

[2] "Preceding" refers to the order in the original code as it would be executed by a conventional machine.

and compiler-generated code, nor between numeric and symbolic programs. Since the analysis of these seven programs consumed some 40 h of machine time, it was decided to bring the data collection phase of our studies to a halt.

The seven programs traced were as follows.

1) BMD01: 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, those of the 3600 itself were chosen. Table I 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 [5] showed that for code written for the 7090, a relative improvement of 1.86:1 could be achieved with a stack length of 10 while blocking on all conditional jumps. This was considerably less than the 51:1 improvement found with maximum speed. Therefore, it was decided to let the stack length (and other parameters) go to infinity and examine the effects of bypassing various numbers of conditional jumps.

For zero jumps bypassed, we found an average improvement of 1.72 to 1 (see Fig. 1 and Table II). That is, the average program examined ran 1.72 times as fast with an infinite stack, infinite registers, infinite storage, and infinite functional units as it did in an ordinary everyday 3600. Clearly, conditional jumps were preventing any substantial amounts of parallelism. If we allow bypassing of one conditional, the average program runs 2.72 times as fast as when run sequentially.

The relative speed increases as the $\sqrt{j}$ where j is the number of jumps bypassed. That is, if we bypass four jumps, the program runs twice as fast as if we bypass only one jump. Similarly, 16 jumps bypassed 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

If we can assume that the programs examined are representative of programs in general, then an average program will run 1.72 times as fast (0 jumps in Table II) on a machine with infinite resources as on a conventional machine. The observed range is between 1.22:1

TABLE I

Relative Speed of Various Instructions in Various Machines with 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

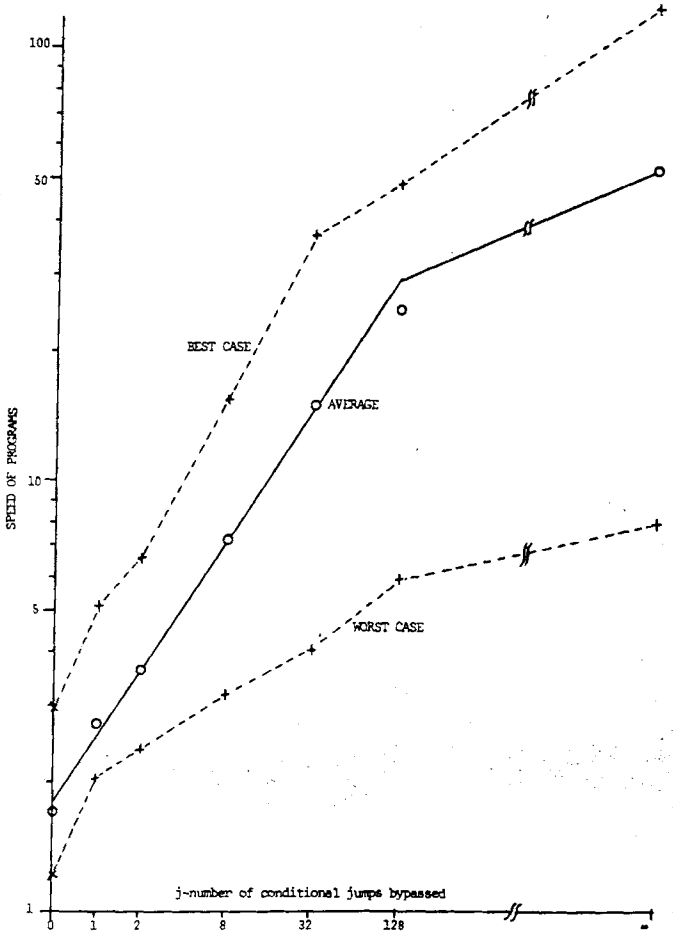

Fig. 1. Average speed as a function of number of conditional jumps that are bypassed—infinite stack machine.

TABLE II

Speedup of Seven Programs (in a Machine with an Infinite Stack) as a Function of the Number of Conditional Jumps Passable

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	120.5
AVERAGE	1.72	2.72	3.62	7.21	14.8	24.4	51.2

and 2.98:1. While it must be admitted that there is some improvement, and while it may be a cost-effective idea to apply to designing large-scale machines, it is not the sort of dramatic breakthrough one might have hoped to find. Indeed, Goode [11] used to urge that system engineers not concern themselves with redesigns that promise a payoff of less than a hemibel (factor of 3) and should preferably look first for order of magnitude (factor of 10) improvements. On his scale, this represents barely a hemi-semi-bel.

The relative speed of execution goes up only as the square root of j, the number of conditional jumps bypassed, and the number of paths that must be maintained simultaneously may go up as fast as 2^j. The authors' attention has been drawn to recent work by Kuck *et al.* [13]. By substantial preprocessing of programs during compilation, several levels of conditional jumps can be collapsed into one level, and their results are comparable to ours.

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. However, 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 latter direction is 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 effectively "unfolded" all the loops in BMDO1 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 percent faster than with them left in. Thus, we conclude on the basis of this very limited experiment that this approach does not appear to offer much help.

An investigation was carried out to determine how long a stack would be required to reach the theoretical speedup of 51 times if we ignored the problem of conditional jumps. Fig. 2 and Table III show the average speed of our seven programs as a function of the dispatch stack length under the assumption that any number of conditional jumps may be bypassed. The important things to be noted in Fig. 2 are, first, that even with a stack length as short as two, bypassing all conditional jumps allows a program to run twice as fast as if it had an infinite stack and blocked on conditionals. It appears that stack length is not nearly as important as the effect of conditional jumps. Second, it should be noted that even with a stack of length 64, the machine is still a factor of four slower than with an infinite stack. This implies that instructions must be moved a long way from their original locations (past more than 64 instructions) in order to achieve maximum speed.

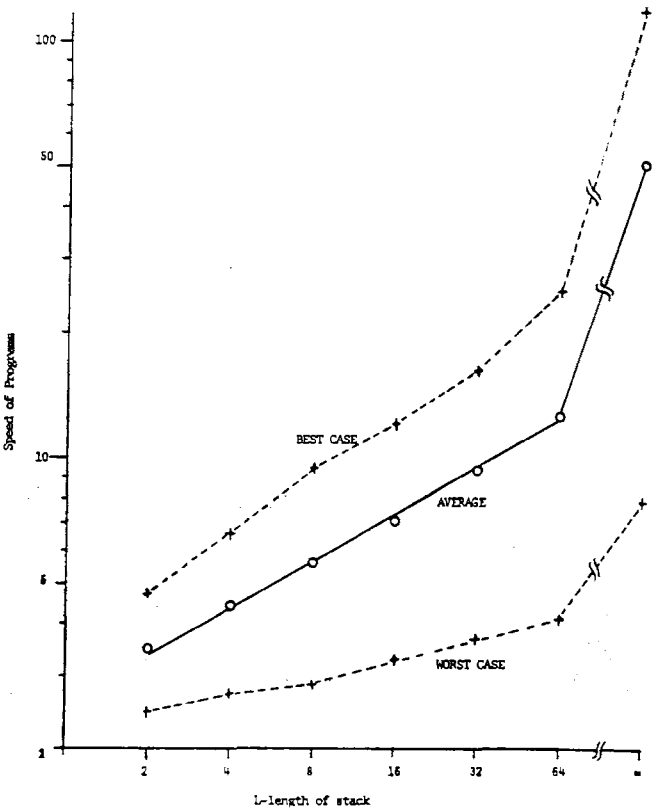

Fig. 2. Average speed as a function of stack length assuming all conditional jumps can be bypassed.

TABLE III

SPEEDUP OF SEVEN PROGRAMS AS A FUNCTION OF LENGTH OF THE 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	13.20
AVERAGE	3.45	4.40	5.52	7.00	9.20	12.49

CONCLUSIONS

Within the programs that were examined, there is a potential parallelism of 51:1. Even given all the resources they might conceivably need, these programs were severely inhibited by the presence of conditional jumps. Limiting them to bypassing no more than two conditionals, we could extract less than a 4:1 improvement in speed. To run ten times as fast as a one-instruction-at-a-time machine, 16 jumps must be bypassed. This implies up to 65 000 paths being explored simultaneously. Obviously, a machine with 65 000 instructions executing at once is a bit impractical.

Therefore, we must reject the possibility of bypassing conditional jumps as being of substantial help in speeding up the execution of programs. In fact, our results seem to indicate that even very large amounts of hardware applied to programs at run time do not generate hemibel improvements in execution speed.

We are left, then, with three alternatives: extensive preprocessing of programs as suggested by Kuck *et al.* [13]; recasting algorithms to take advantage of machine parallelism as, for example, in the Goodyear STARAN or the Illiac IV; or just plain speeding up a conventional monoprocessor so it gets the job done faster.

Acknowledgment

The authors wish to express their appreciation to J. Vervaert and F. Pirz for their help in collecting and analyzing the data presented in this note.

References

[1] H. Hellerman, "Parallel processing of algebraic instructions," *IEEE Trans. Comput.*, vol. C-15, pp. 82–91, Feb. 1966.
[2] H. S. Stone, "One-pass compilation of arithmetic expressions for a parallel processor," *Commun. Ass. Comput. Mach.*, pp. 220–223, Apr. 1967.
[3] C. V. Ramamoorthy and M. J. Gonzalez, "A survey of techniques for recognizing parallel processable streams in computer programs," in *Proc. 1969 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 35. Montvale, N. J.: AFIPS Press, 1969, pp. 1–15.
[4] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.
[5] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.*, vol. C-19, pp. 889–895, Oct. 1970.
[6] E. Bloch, "The engineering design of the Stretch computer," in *Proc. 1959 Eastern Joint Comput. Conf.*, p. 48.
[7] R. T. Blosk, "The instructions unit of the Stretch computer," in *Proc. 1960 Eastern Joint Comput. Conf.*, pp. 299–325.
[8] J. Cocke and H. J. Kolsky, "The virtual memory of the Stretch computer," in *Proc. 1959 Eastern Joint Comput. Conf.*, pp. 82–94.
[9] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The model 91: Machine philosophy and instruction handling," *IBM J. Res. Develop.*, vol. 11, Jan. 1967.
[10] H. S. Stone, "A pipeline pushdown-stack computer," in *Parallel Processor Systems, Technologies, and Applications*, L. C. Hobbs, Ed. Washington, D. C.: Spartan, 1970, pp. 235–249.
[11] H. H. Goode, notes from a course on system design, Univ. Michigan, Ann Arbor, spring 1957.
[12] M. J. Flynn, personal communications.
[13] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," IEEE Comput. Soc. Repository, Publ. R72-109, May/June 1972.

Percolation of Code to Enhance Parallel Dispatching and Execution

CAXTON C. FOSTER AND EDWARD M. RISEMAN

***Abstract*—This note investigates the increase in parallel execution rate as a function of the size of an instruction dispatch stack with lookahead hardware. Under the constraint that instructions are not dispatched until all preceding conditional branches are resolved, stack sizes as small as 2 or 4 achieve most of the parallelism that a hypothetically infinite stack would.**

An algorithm is described that can be used to replace the lookahead 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 percent of the parallelism obtained if an infinite dispatch stack were available under the assumption that it takes zero time to decode and dispatch an instruction.

***Index Terms*—Dispatch stack, lookahead hardware, parallel execution, parallelism, percolation of code, software lookahead.**

Introduction

The problem of detecting and utilizing parallelism in programs has been extensively studied. A review of some of the techniques developed to detect parallelism in higher level languages, particularly in arithmetic expressions, appears in [1]. 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 [2]. The huge Illiac IV has been implemented to take advantage in hardware of array operations that can be executed in parallel [3]. 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 points out that the bottleneck is the decoding and dispatching of a single instruction per machine cycle [4]. 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.

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.

This problem is considered in a companion note to this one; Riseman and Foster [5] examine the relative increase in execution rate as a function of the number of conditional jumps "bypassed;" N conditional jumps can be bypassed by the execution of 2^N simultaneous parallel instruction streams.

Recently, Tjaden and Flynn [6] examined the payoff in using a hardware stack to dispatch and execute instructions in parallel. They examined the speedup in execution as a function of the stack size under the constraint that instructions are not dispatched until all preceding conditional branches are resolved. This note is a continuation of that work. It accepts the restrictions imposed by branching and explores the limit of parallelism obtained by parallel dispatching using such a hardware stack.

The resultant parallelism of seven programs written

Manuscript received November 11, 1971; revised June 21, 1972. This work was supported in part by a study grant from Control Data Corporation.

The authors are with the Department of Computer and Information Sciences, University of Massachusetts, Amherst, Mass. 01002.

```
1    LDA    α
2    LDQ    β
3    ADD    σ
4    QLS    3
5    STA    δ
6    STQ    ε
```

Fig. 1. Two strands of independent code—instructions 1, 3, 5 and instructions 2, 4, 6.

for a CDC-3600 is presented as a function of stack sizes ranging to infinity. An algorithm will also be described to replace the lookahead 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, there will be no instruction below it that is ready for dispatching.

Parallelism

The average amount of parallelism will be defined to be the ratio of the normal sequential execution time 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. This measure is comparable to that used by Tjaden and Flynn [6]. They counted the average number of instructions that could be dispatched in parallel during each cycle of their predecode stack. We differ in that we are letting the system run asynchronously; each instruction has an execution time and is dispatched as soon as there are no dependencies.

The above measures and assumptions appear to be equivalent. We also assume that the dispatching interval takes zero time.[1] Since instructions are dispatched asynchronously, many instructions can be dispatched during the execution of a single instruction. One should note the implications of this assumption. Suppose we have ten instructions in a row that 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. Clearly, this is an impossible condition to achieve. 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. One should still keep in mind that this will tend to result in a larger amount of the potential parallelism being achieved by shorter stacks.

Suppose we consider a program not as a set of instructions that must be executed in some sequential order; rather we will look at a program as a set of instructions with constraints upon the time at which instructions can be executed. 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*.

Let us examine the constraints upon the dispatching of an instruction. It is clear that no instruction can begin execution until all its sources are available. If sufficient resources are provided, an instruction need not be delayed because its destination is not available. This situation has been discussed as "open effects" instructions [6]. Each time a register is referenced as a destination but not a source for an instruction, this can be construed as a "renaming" of that register; this generates a "new edition" of the register and hence eliminates resource dependencies. There are still two types of dependency that prevent us from dispatching all the instructions in a program simultaneously and completing the program in one instruction execution time. These are the dependency of data and the dependency of flow of control.

Data dependencies will cause a program to consist of subsections we will call "strands of code," in which an on-going manipulation of data is accomplished. These strands involve a sequence of instructions in which the destinations of one instruction affect the source of a following instruction. For example, in Fig. 1, instructions 1, 3, and 5 belong to one strand, while 2, 4, and 6 belong to another. These two strands may or may not be subparts of some superstrand. At least locally they are independent and may be executed in any convenient order provided only that the within-strand-ordering is preserved.

Fig. 2 shows a block diagram of a small program that displays typical flow of control dependencies (also referred to as procedural dependencies [6]). Blocks labeled P are processes, while diamonds represent conditional jumps or branch points within the program. Conditional jumps are handled in this note as they are in [6], namely, blocking the execution of further instructions until the conditional jumps are resolved. Consequently, one can think of a program decomposed into linear (nonbranching) chunks. A linear chunk begins with an entry statement or an instruction that is a target of a conditional jump. It terminates with an exit statement or a conditional jump. We assume non-self-modifying code

[1] For a more complete discussion of this assumption, refer to the companion note [5] in this issue.

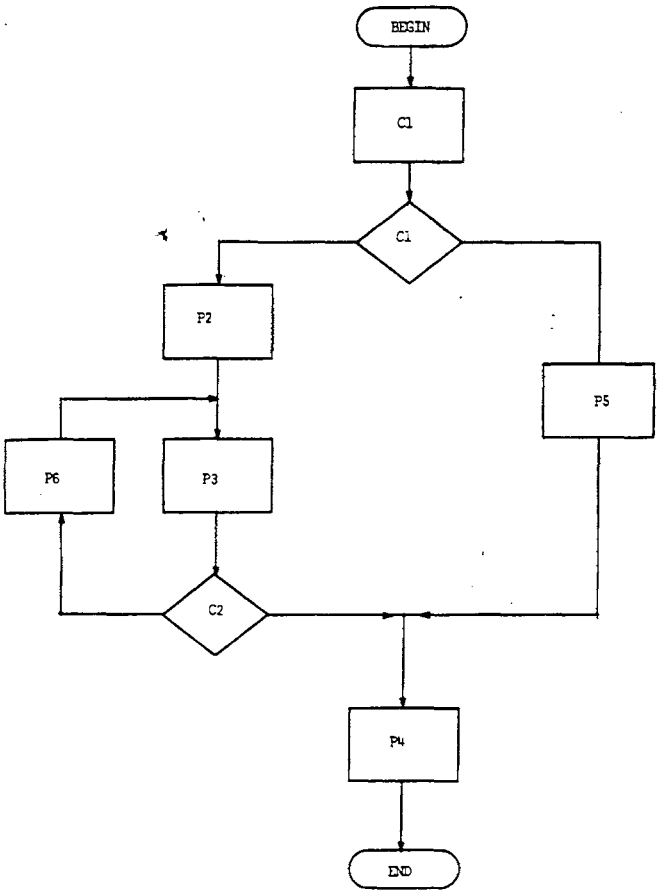

Fig. 2. Block diagram showing a small program with typical flow of control dependencies. Blocks labeled P are processes, while diamonds represent branch points.

throughout. Fig. 3 shows the example program divided into linear chunks. Note that $P3$ and $P4$ appear twice because there are two different chunks that might involve them. This is a helpful way of looking at the program; if one blocks on conditionals, it is within these chunks that strands are executed in parallel by dynamically dispatching the instructions in a different order. No instruction from one chunk can be dispatched before the conditional terminating a previous chunk. Otherwise an instruction will be executed that might not be in the instruction stream when the previous conditional is resolved.

In summary, the machine that was simulated by Tjaden and Flynn [6] was an IBM 7094 with the following characteristics: 1) there is a stack of finite length; 2) there are infinitely many copies of the high-speed registers, such as the accumulator; 3) no instruction will be dispatched before any conditional instruction preceding it in the stack is resolved. The machine that has been described here and that was simulated is a CDC-3600 with the same characteristics as above, except that the stack is possibly of infinite length and there are infinitely many registers and functional units of all types so that no instruction is delayed due to the availability of any hardware. Further details of this simulation are available in [7].

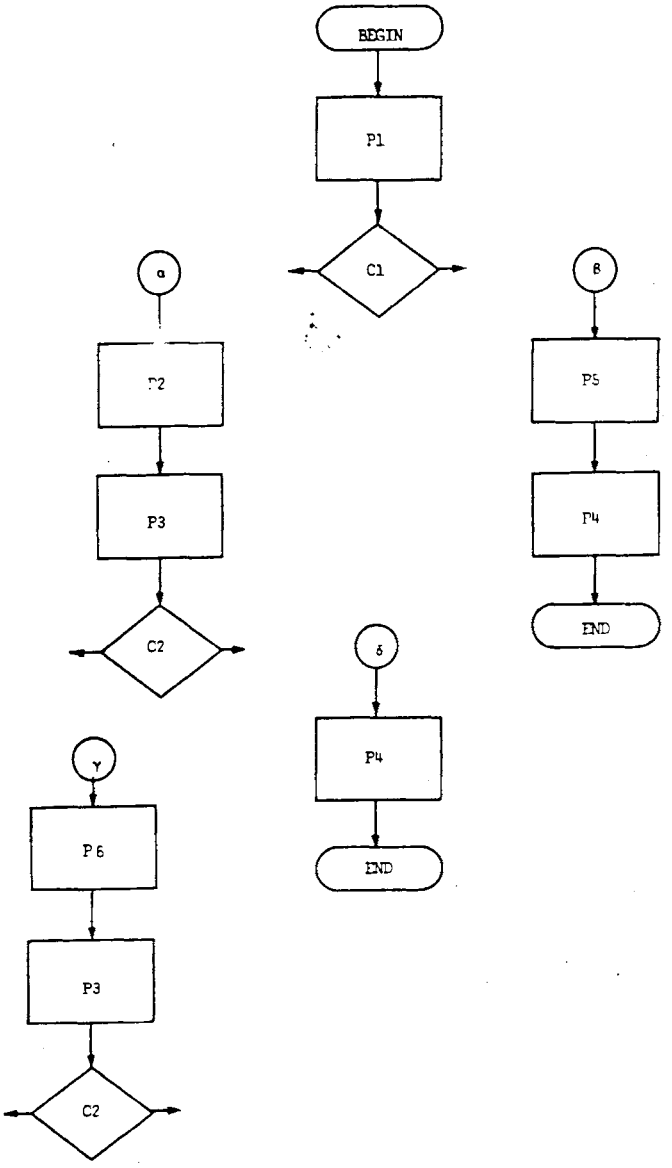

Fig. 3. Linear chunk decomposition of the example in Fig. 2.

Comparison of Experimental Results

Data were collected by tracing seven programs that included both compiled code and hand-generated code and amounted to almost 2 million instructions (see [5] for a description of these programs).

First we examined the increase in execution rate as a function of stack size. Table I presents the resultant parallelism for stack sizes varying from 2 to 64. The parallelism that would be obtained from an infinite size stack is also included. These data are not graphed 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 potential parallelism is obtained by a stack size 8. Stack sizes that would be necessary to achieve 90, 99, and 100 percent of the parallelism of an infinite stack are given in Table II. The limit on the parallelism that is achieved with an

TABLE I
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
AVERAGE	1.568	1.645	1.696	1.718	1.721	1.721	1.721

TABLE II
STACK SIZES NECESSARY TO ACHIEVE 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

infinitely large stack was found to be slightly more than 1.72; this means that the usual sequential machines would take 72 percent longer to execute the set of 7 test programs than this parallel machine. These results are somewhat worse than those given by Tjaden and Flynn [6], 86 percent for a stack size of 10. This difference is not great considering that a different set of programs for different machines were used. Also, as expected, short stacks in our simulation achieve relatively larger amounts of parallelism than they obtained, since in effect they assumed a dispatch time of $1/S$ (S being the stack size); and our zero dispatch time allows a string of independent instructions to "flush through" the stack as long as there is one stack position that is available that is *not* holding some delayed instruction.

All of our experimental results discussed so far 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 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 percent with a stack of size 32. This effect appears to be somewhat less than that reported by Tjaden and Flynn.

SOFTWARE PERCOLATION

Two factors that limit the length of the dispatch stack involve the lookahead 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 lookahead hardware in the dispatch stack is necessary to determine dependencies in the 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. However, it is possible to reorder program-code-as-written so that interlocking hardware in the stack will not be required. That is, the code will be reordered such that, if the instruction at the top of the stack cannot be dispatched (because it is waiting for computation of one of its requisite sources), then there is no instruction below it in the stack that could be dispatched at this time. If this is achieved, then each instruction will be dispatched as early as possible, and hence the program will finish as soon as possible, and it will be done without the expense of any interlock hardware.

We are going to do this reordering prior to execution, so we must not move instructions from one linear chunk to another, for we are uncertain of the order of execution of the different chunks. Further, the reordering inside a given chunk must be done so that it preserves the logic of the original code.

A one-pass algorithm can accomplish this by assigning an "earliest possible dispatch time" to each instruction that is not less than the time at which the sources of that instruction become available and not less than the time at which that linear chunk is entered. Instructions are then sorted into ascending "earliest possible dispatch times" and the reordering is complete. Details of this straightforward process are available in [7], but perhaps a brief description here might be of interest.

We begin by examining the object code in the form that would normally be generated by a compiler or assembler ready for execution. The first task is to recognize the linear chunk boundaries. We assume that we are within a chunk. If the next statement is not a conditional branch, we continue within the chunk. Unconditional branches are ignored, and the chunk continues at the destination of the unconditional branch, perhaps requiring duplication or reduplication of code (see, for example, $P3$ and $P4$ of Fig. 3). A conditional branch ends this chunk, and the next instruction begins a new one.

As each instruction is added to a chunk, we update a table called the "Most Recent Change Table." The entry in this table for each source of the current instruction is examined, and the largest value so found is assigned to the "earliest possible dispatch time" of this instruction. This time plus the execution time of the instruction is used to update the most recent change entry of all the destinations of this instruction. It is

TABLE III

SPEED OF PERCOLATED PROGRAMS RELATIVE TO 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

possible to view this as a graph theoretic procedure in which the instructions form the nodes of the graph.

Gonzelez and Ramamoorthy [8] found that a very large amount of time was necessary to discover potential parallelism in Fortran source programs. It appears that the time they require to detect parallelism goes up as the cube of the number of statements involved (50 statements require 2 s, 100 require 8 s). In the first place, our chunks are quite small, due to the high frequency of conditional jumps; and, in the second place, we do *not* attempt to discover potential parallelism *between* chunks, only within them, and it is phase II of their analysis (the construction of the permissible transition graph) that is the lengthiest. Thus our algorithm should be considerably faster than theirs. Nonetheless, it is clear that this form of preprocessing of programs would be worthwhile only for those cases in which many executions may be expected.

The reader's attention is directed to the following phenomenon. Suppose we have a chunk of code such that the conditional jump that ends the chunk is resolvable early in the execution of the chunk. Then in an infinite stack machine, the new chunk can be started as soon as this conditional is resolved. But in our software percolation scheme, we cannot allow any percolation across chunk boundaries. Thus, some potential parallelism is lost but, as we shall see below, this is quite small in magnitude.

The speed of percolated code relative to unpercolated code with an infinite dispatching stack is shown in Table III. These numbers are found by dividing the execution time of the program with an infinite stack by the execution time of the percolated program with a dispatching stack of length 1. Note that the assumption of zero dispatch time affects the percolated code in the same manner that was discussed previously. This experiment was not run for one of the programs, INTERIT, because the information on one of the tapes was accidentally destroyed. For the six programs analyzed this way, the average speed of the percolated program was 93.5 percent of the speed of the unpercolated program dispatched from an infinite stack. Stating this another way, with an infinite stack, the six programs (on the average) ran 1.51 times as fast as they would have in a conventional sequential machine. Using the percolation algorithm in place of most of the decoding hardware, programs ran 1.41 times as fast as in a conventional machine. These values are somewhat lower than they might have been because the program that was not run had the largest values of parallelism.

DISCUSSION AND CONCLUSIONS

Under the assumption of a zero dispatching interval, we have determined the upper bound on the parallelism derived for various stack sizes. Most of the parallelism achieved by using a stack to decode and dispatch instructions is obtained with very short stacks. In all but one case, a stack size of 4 would achieve 90 percent 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.

One may still feel that there are cases in which the additional expense that is required to achieve this parallelism is justified. This note has described an alternative to achieving this parallelism strictly in hardware. The percolation algorithm presented in this note approximates the dispatch stack by reordering instructions prior to execution. This method achieves 93.5 percent of the parallelism of an infinite stack. 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 focused upon is the problem of conditional branches, or additional processing to convert the code to a form that takes advantage of this parallel processing structure.

ACKNOWLEDGMENT

The authors wish to express their appreciation to J. Vervaert and F. Pirz for their help in collecting and analyzing the data presented in this note.

REFERENCES

[1] C. V. Ramamoorthy and M. J. Gonzalez, "A survey of techniques for recognizing parallel processable streams in computer programs," in *Proc. 1969 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 35. Montvale, N. J.: AFIPS Press, 1969, pp. 1–15.

[2] M. E. Conway, "A multi-processor system design," in *Proc. 1963 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 23, 1963, pp. 139–146.

[3] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.

[4] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.

[5] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," this issue, pp. 1405–1411.

[6] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.*, vol. C-19, pp. 889–895, Oct. 1970.

[7] C. C. Foster and E. M. Riseman, "A study of the constraints upon the parallel dispatching and execution of machine code instructions," Dep. Comput. and Inform. Sci., Univ. Massachusetts, Amherst, Rep. TR72A-1, Feb. 1972.

[8] M. J. Gonzalez, Jr., and C. V. Ramamoorthy, "Program suitability for parallel processing," *IEEE Trans. Comput.*, vol. C-20, pp. 647–654, June 1971.