Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance

PHILIP G. EMMA AND EDWARD S. DAVIDSON

Abstract—**The nature by which branches and data dependencies generate delays that degrade pipeline performance is investigated in this paper. We show that for the general execution trace, few specific delays can be considered in isolation; rather, the magnitude of any specific delay may depend on the relative proximity of other delays. This phenomenon can make the task of accurately characterizing a trace tape with simple statistics intractable. We present a set of trace reductions that facilitates this task by simplifying the corresponding data-dependency graph. The reductions operate on multiple data-dependency arcs and branches in conjunction; those arcs whose performance implications are redundant with respect to the dependency graph are identified, and eliminated from the graph. We show that the reduced graph can be accurately characterized by simple statistics. We use these statistics to show that as the length of a pipeline increases, the performance degradation due to data dependencies and branches increases monotonically. However, lengthening the pipeline may correspond to decreasing the cycle time of the pipeline. These two opposing effects are used in conjunction to derive an equation for optimal pipeline length for a given trace tape. The optimal pipeline length is shown to be characterized by $n = \sqrt{\gamma\alpha}$ where γ is the ratio of overall circuit delay to latching overhead, and α is a function of the trace statistics that accounts for the delays induced by data dependencies and branches.**

Index Terms—**Branch delay, data dependency, performance analysis, pipeline, program trace, trace reduction.**

I. Introduction

In this paper, a method is developed for deriving performance models of pipelines by using statistics obtained from trace tapes to characterize the workload. This method is presented as an alternative to trace-driven simulation, and it proves to be more flexible in the sense that a single set of trace statistics is sufficient for modeling a large class of pipelines. A set of trace reductions is presented that simplifies the statistics that are collected, and assures that the resulting statistics contain all of the significant information that is required for an accurate model.

Trace-driven simulation has been used by many as a tool for evaluating various aspects of computer design. It has been used to study paging algorithms [1], [2], cache-management algorithms [3], database-buffering strategies [4], buffered disks [5], file-migration policies [6], branch-prediction strategies [7], [8], and CPU design [9]–[11]. Trace-driven simulation facilitates the capture of very detailed information regarding the systems under study; however, the information that is captured tends to be quite specific to the particular systems that are simulated.

Some outstanding work has been done by obtaining measurements from trace tapes, and using these measurements to predict computer performance, thereby "shortcutting" the simulation process. In one of the seminal papers in this area, Peuto and Shustek measure instruction mixes for various benchmarks, and use instruction-timing models to evaluate the relative performance of the IBM 370/168 Model 1, and the Amdahl 470 V/6 [12]. Clark and Levy use a similar form of analysis to evaluate a VAX 11/780 [13]. Blake measures instruction mixes to draw general conclusions about stack machines based on the performance of the HP 3000 [14]. Kobayashi measures dynamic frequencies and nesting levels of loops in trace tapes to characterize program behavior in problem state [15], [16]. MacDougall uses hierarchical measurement to assess the composition of overall system load relative to processes, and characterizes the component processes by examining instruction mixes to arrive at a system-level performance analysis [17].

In this paper, traces are used to measure the nature of interaction between instructions. Namely, the frequency of taken branches and the proximity of data dependencies are measured to assess the degree to which a pipeline machine will suffer performance degradation due to the interlocks inherent to programs. It is demonstrated that due to the complexity of interaction among data dependencies, and between data dependencies and branches, no trace can be fully characterized by simple statistics without specific knowledge of the target pipeline. However, a set of trace reductions is presented that reduce any trace to an equivalent trace that has identical performance characteristics, such that simple statistics exactly characterize the equivalent trace independently of the target pipeline.

This paper shows that as the length of any pipeline is increased, data dependencies and branches monotonically degrade the pipeline performance (in terms of clock cycles per instruction). However, increasing the length of a pipeline for any specific architecture corresponds to reducing the basic instruction cycle, i.e., it corresponds to decreasing the cycle time of the machine. These two opposing effects are used in conjunction to derive a model for determining the pipeline length with optimal performance, given a set of trace statistics and one additional technology parameter that characterizes the

Manuscript received March 15, 1986; revised June 12, 1986. This work was supported in part by the Joint Services Electronics Program under Contract N00014-84-C-0149.

P. G. Emma is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 10598.

E. S. Davidson is with the Center for Supercomputer Research and Development, University of Illinois, Urbana, IL, 61801.

IEEE Log Number 8714103.

time overhead associated with latching between pipeline segments. For an interesting case study of how technology influences the choice of pipeline length, see Doran [18].

In Section II, the effects of data dependencies and branches on pipeline performance are discussed, and a model for these effects is developed. In Section III, the set of trace reductions that are required to obtain simple statistics that characterize the trace is derived, and these reductions are applied to two example traces in Section IV. In Section V, the performance degradation inherent in lengthening the pipeline is traded off against the reduction in cycle time to arrive at a solution for optimal pipeline length for a given set of traces. The application of these same techniques to more complex classes of pipelines is discussed in Section VI.

II. Data Dependency and Branch Degradation

Any pipeline that executes the instruction flow of a machine can roughly be separated into two sections, namely, setup and execution. The setup section corresponds to the instruction unit and is responsible for instruction fetch, instruction decode, and operand fetch. The execution section performs a functional operation on one or more input operands and produces an output operand.

If the sink operand (output) of an instruction is the same as a source operand (input) of a latter instruction, then the latter instruction is said to be *data dependent* on the former instruction. The former instruction is said to be the *resolving* instruction, and the latter instruction is said to be the *dependent* instruction. In the case of instruction pipelines, operands can be in memory locations, or in registers, or in condition-code storage-elements (e.g., branch instructions require condition codes for input).

Define the *dependency distance* between a resolving instruction and its dependent instruction as the number of instructions between the two instructions plus 1. That is, if instruction $i + k$ depends on instruction i, then the *dependency distance* is k.

For the purpose of developing the following formal theory, it is assumed that all pipeline segments are constant-time segments with unit service time, and that all instructions in the input stream traverse all pipeline segments in order, i.e., strict execution order is observed. The only reason that these two assumptions are made is that they cause a one-to-one mapping to exist between the dependency distance associated with two instructions and (in the absence of other delays) the difference between the times at which they enter the pipeline. This assumption permits the set of reductions that are developed in Section III to be treated in a simple manner for illustrative purposes. If either assumption does not hold, the validity of the trace reductions is unaltered provided that the definition of dependency distance is broadened. This extension is explained in Section VI.

Whenever a data dependency occurs, the effect that this has on the performance of the pipeline is that the dependent instruction cannot enter the execution section until the resolving instruction leaves the execution section. If out-of-order execution is not permitted, then the dependent instruction blocks the flow of all instructions behind it as well. For a pipelined execution unit having N_E segments, the maximum amount of time that the dependent instruction must wait is given by $(N_E - D)^+$, where D is the dependency distance, and the operator $(x)^+$ maps negative values of x onto the value 0, and positive values of x onto x.

Assume that in the absence of a taken branch, instructions are fetched from sequential locations in memory, and they enter setup in that order. Define a *branch target* as the instruction that follows a taken branch. The effect of a taken branch is to cause a full or partial flush of setup; the associated penalty is attributed to the *branch-target* instruction.

If a branch is not taken, then no penalty is incurred in setup (although conditional branch instructions can incur penalty in the execution section of the pipeline due to data dependencies). Since the target address of a branch instruction is an input operand for the instruction, it is assumed that the target address is known at the end of setup. If the branch is unconditional then it is always taken, and thus, the penalty associated with the target of an unconditional branch is equal to the flush depth. If the branch is conditional, then the penalty associated with it may also have an inherent data-dependency penalty, e.g., the branch may be dependent on a condition that may not have been resolved.

In a pipelined setup section having N_S segments, where each segment of the pipeline takes unit time, the maximum flush penalty is $N_S - 1$, i.e., the target instruction requires time N_S rather than time 1 to reach the end of setup. This is the flush penalty unless the branch target is fewer than N_S instructions past the branch, i.e., the only cases that do not require a full flush of setup are short forward branches.

Thus, it is assumed that unconditional branches do not require execution cycles; when an unconditional branch is encountered, the incurred penalty is equal to the flush depth. For conditional branches, the penalty is twofold: first there is a data dependency that must be resolved between a condition-code setting instruction and the branch, and next, there is a flush penalty if the branch is taken.

If one were to try to predict the performance of a pipeline by looking at the data dependencies and branches in a particular execution trace, the task would be difficult since these effects are not independent, and their associated degradations tend to cancel each other somewhat. Let BW^{-1} denote the *inverse bandwidth* of the pipeline in cycles per instruction. A simple model that does not include branches, and that considers only those data dependencies that do not interfere with each other yields the following solution for inverse bandwidth

$$BW^{-1} = 1 + \sum_{D=0}^{N_E-1} p_D (N_E - D)^+.$$

In this case, it is assumed that: instructions are executed in order, N_E is the length of the execution pipeline, and p_D is the probability that a data dependency exists at distance D for an arbitrarily selected instruction. The case $D = 0$ is included to encompass those branch instructions that compute their own conditions. This simple model assumes that no branches are taken.

A simple model that does not include data dependencies, but

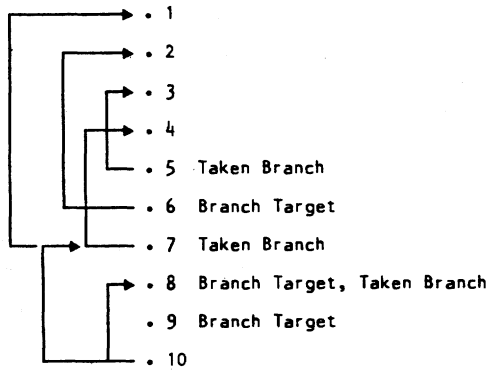

Fig. 1. A typical data-dependency graph.

assumes that a full flush of setup is required whenever a branch is taken (i.e., short forward branches are not modeled) yields the following solution for inverse bandwidth

$$BW^{-1} = 1 + p_b(N_S - 1).$$

In this case, N_S is the length of the setup pipeline, and p_b is the probability that an arbitrarily selected instruction is a taken branch.

Let all of the data dependencies in a program be represented in a dependency graph. Fig. 1 shows a dependency-graph representation of a typical run of ten instructions. A data dependency is represented by an arc that originates at the dependent instruction and terminates at the resolving instruction. For example, the arc that originates at node 7 and terminates at node 1 represents that the seventh instruction is dependent on the first instruction. Note that in the figure, nodes 6, 8, and 9 are designated as branch targets; thus, instructions 5, 7, and 8 are taken branches.

In general, data-dependency arcs are not isolated from each other; rather, they overlap one another in a complex fashion as shown in Fig. 1. Furthermore, many data-dependency arcs span branch targets, i.e., a taken branch can occur between the resolving instruction and the dependent instruction. If the fact that arcs may overlap is ignored, and the fact that arcs may span branch-target instructions is ignored, then the superposition of the two models above yields the following model.

$$BW^{-1} = 1 + p_b(N_S - 1) + \sum_{D=0}^{N_E-1} p_D(N_E - D)^+.$$

Recall that dependency distance was defined so as to reflect the actual distance (in time) between instructions in the absence of delay. Whenever a data dependency or a taken branch causes a delay in the pipeline, that delay increases the time between instructions on either side of the delay point by an amount equal to the delay itself. Thus, the "effective distance" associated with an arc that spans a delay point is increased, i.e., the arc is lengthened from a temporal point of view.

In the equation above, the term $(N_E - D)^+$ reflects the delay associated with an arc whose dependency distance is D, under the assumption that the arc does not span other delays. However, if the arc does span other delays, then the delay associated with the effectively lengthened arc is lessened. Thus, the model above yields a pessimistic estimate of performance. In order to model an execution trace effectively, one must derive a set of sufficient statistics from the trace, i.e., a set of statistics that encompasses all of the necessary information in the trace.

Define an ith-order statistic as a statistic that contains information that characterizes the overlap of i arcs. First-order statistics can be derived from any trace by counting the number of branch targets, and by keeping a histogram of dependency distance for all data dependencies in the trace. For example, the first-order statistics for the trace segment shown in Fig. 1 are

$$p_2 = 2/10,\ p_3 = 2/10,\ p_4 = 1/10,\ p_6 = 1/10, \text{ and } p_b = 3/10,$$

i.e., out of ten instructions, there are two arcs with dependency distance 2, two arcs with dependency distance 3, etc., and three branch-target instructions. The model that was derived above assumes that first-order statistics are sufficient for predicting the performance of a trace for a given N_E and N_S. For example, if $N_E = N_S = 5$, then the model above yields an inverse bandwidth of

$$\begin{aligned} BW^{-1} &= 1 + \frac{3}{10}(5-1) + \frac{2}{10}(5-2)^+ + \frac{2}{10}(5-3)^+ \\ &\quad + \frac{1}{10}(5-4)^+ + \frac{1}{10}(5-6)^+ \\ &= 1 + 1.2 + 0.6 + 0.4 + 0.1 + 0 \\ &= 3.3. \end{aligned}$$

It is simple to demonstrate via hand simulation that this model is pessimistic. For example, let *Si* represent the setup processing for instruction i, and let *Ei* represent the execution processing for instruction i. The four dependent nodes and three branch targets in Fig. 1 impose seven constraints:

1) $E5$ cannot start until $E3$ has finished,
2) $E6$ cannot start until $E2$ has finished,
3) $E7$ cannot start until $E1$ and $E4$ have finished,
4) $E10$ cannot start until $E7$ and $E8$ have finished,
5) $S6$ cannot start until $S5$ (and $E3$) have finished,
6) $S8$ cannot start until $S7$ (and $E1$ and $E4$) have finished, and
7) $S9$ cannot start until $S8$ has finished.

The pipeline flow for these ten instructions subject to these seven constraints can then be mapped onto a pipeline for which $N_E = N_S = 5$ in a straightforward manner as shown in Fig. 2. Since the tenth instruction experiences 15 cycles of delay prior to entering setup (i.e., in the absence of delays, it would enter on the tenth cycle), the inverse bandwidth of the pipeline is (10 + 15)/10 = 2.5 cycles per instruction.

Thus, a model that is based on simple first-order statistics is insufficient for accurately predicting performance. A more accurate model can be constructed based on second-order statistics by considering all pairs of arcs that overlap in some way, and by deriving a histogram of the dependency distances of the arcs that reflects the way in which the arcs overlap.

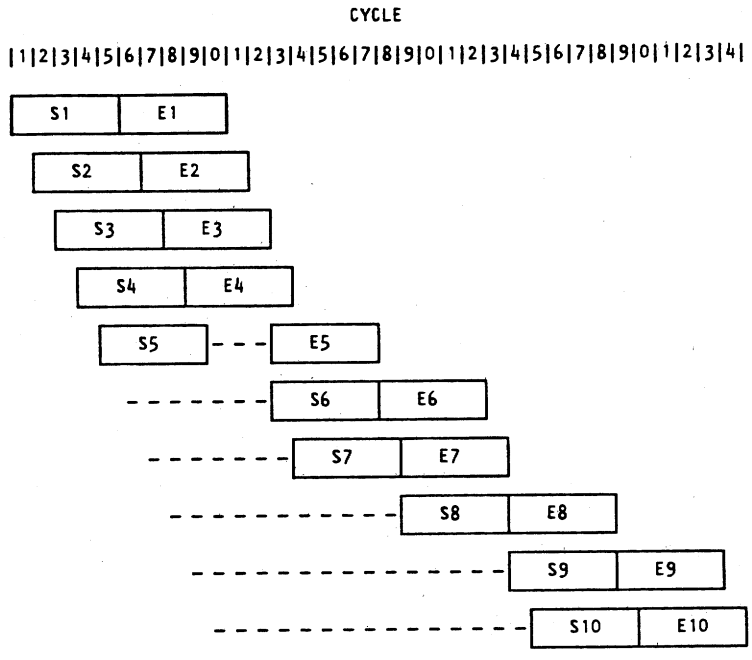

Fig. 2. Pipeline flow of ten instructions with $N_E = N_S = 5$.

Although such a model would be more accurate than a model that is based on first-order statistics, it would still be a pessimistic model, i.e., it would contain no information about the way in which triples of arcs, quadruples of arcs, etc., interact. Instead, consider reducing the trace prior to taking statistics.

A trace is said to be *separable* at a node if the dependency graph for that trace can be cut at that node without removing any of the dependency arcs. Note that the node at which the trace is separable may appear in both subgraphs. Consider some instruction j in an ordered execution trace of length N whose nodes are $(1, \cdots, j-1, j, j+1, \cdots, N)$. If the trace is separable at node j, then there is no dependency arc that originates at any of the nodes $(j+1, \cdots, N)$ and terminates at any of the nodes $(1, \cdots, j-1)$. For example, the trace shown in Fig. 1 is separable at node 7.

Any segment of a dependency graph that originates at one separable node and terminates at another separable node is said to be a *chain*. Note that for a trace of length N, the corresponding dependency graph is separable at nodes 1 and N. A chain that contains no separable nodes except at its endpoints is said to be an *inseparable chain*. For an inseparable chain of length i, the only set of sufficient statistics are ith-order statistics.

In general, an entire execution trace may form an inseparable chain. Thus, the only set of sufficient statistics that are guaranteed to characterize a trace of length N are Nth-order statistics, i.e., the sufficient statistics have the same complexity as the trace itself. If one were to use an entire trace in an analytic model, one would be doing simulation rather than analysis. In fact, this is exactly what has been done heretofore when an exact measure of performance was desired.

If simulation is not convenient, then ith-order statistics can be used in an ith-order model for some small value of i (typically $i = 1$). Such an approximation can be fairly good for short pipelines, since $(N_E - D)^+$ is only sensitive to values of D that are less than N_E. Thus, if N_E is small, then the delays that result do not tend to have global effects. The problem with using small values of i is that the sensitivity to i is unknown, yet it is known that for $i \ll N_E$, the resulting performance estimators can be very inaccurate.

However, the solution to workload modeling that is derived in this paper is an exact solution that uses a first-order model. This is accomplished by using a set of trace reductions that breaks the general complex inseparable chain into small simple chains from which first-order statistics can easily be derived. These reductions preserve the way in which data dependencies and branches interact, and both the static and the dynamic performance characteristics of a reduced trace are identical to those characteristics of the original trace.

Thus, a reduced trace contains all of the relevant performance information, yet the corresponding dependency graph consists of simple chains that are easily characterized. Hence, the set of first-order statistics that is collected from any reduced execution trace is a sufficient set, and a first-order model is thus an exact performance predictor. Since the model must handle branch degradation as well, the statistics are two-dimensional; the first dimension is the dependency distance of an arc, and the second dimension is the number of taken branches that are spanned by the arc. In the next section, the set of reductions that are used is derived, and an algorithm is presented that performs the reductions, and subsequently renders first-order statistics from the reduced trace.

III. Execution-Trace Reductions

An execution trace can be considered to be a data-dependency graph, where each node in the graph represents an instruction in the trace, and each directed arc in the graph represents a precedence (ergo *directed* arc) based on an existing data dependency in the trace. A general graph may be

an inseparable chain consisting of many arcs. The trace reductions use known precedence information to remove those arcs that do not represent delays, i.e., those arcs that have no performance implications because of their temporal proximity to other delays.

If the removal of arcs is done in the manner specified in this section, then any dependency-graph representation of an execution trace can be reduced to a graph that is composed of simple inseparable chains. Furthermore, this can be done in such a manner that the reduced graph is "equivalent" to the original graph in the sense that all delays inherent to the original graph are present in the reduced graph. The simple chains in the reduced graph can then be analyzed in a straightforward manner to produce a set of first-order statistics that accurately characterizes the original trace.

In Section III-A, a set of three reductions is presented, each being relevant to a specific case in which the removal of an arc is appropriate. The permissibility of arc removal is proved based on the surrounding precedence relationships present in each of these cases. In Section III-B, an algorithm is presented that uses these reductions to break the general data-dependency graph into small simple chains, and it is shown how each of these chains can be rendered into a simple set of first-order statistics.

Preliminary to these subsections, it is helpful to establish a consistent notation as follows. Let the instructions of an execution trace be labeled in temporal order, i.e., instruction i precedes instruction $i + 1$ by exactly one instruction for all i. Define a data-dependency graph that corresponds to the trace, such that each node in the graph has the same label as its corresponding instruction in the trace. Thus, if a data dependency exists between two instructions i and j, then the label of the dependent instruction must have a larger value than the label of the resolving instruction. That is, if $i > j$, then it cannot be the case that instruction j depends on instruction i, but it may be the case that instruction i depends on instruction j. Note that any instruction may be the target of a taken branch, and should be marked as such if this is the case. For example, in Fig. 1, nodes 6, 8, and 9 are nodes that correspond to taken-branch targets. In a trace, taken-branch targets can be identified with specific branches; in an architecture not using the delayed branch, if instruction i is a taken-branch target, then instruction $i - 1$ is a taken branch.

If a data dependency exists such that instruction i is dependent on instruction j, then $i > j$, and the corresponding dependency graph has a directed data-dependency arc that originates at node i and that terminates at node j. Any node may originate zero or more arcs, and may also terminate zero or more arcs. The pipeline through which the instruction stream flows is assumed to have a setup section consisting of N_S segments, and an execution section consisting of N_E segments. Thus, the pipeline comprises $N_S + N_E$ segments.

In the following subsections, all node labels are designated by subscripted letters, e.g., i_1. All label designations with the same letter, but different subscripts, are assumed to be relevant to a particular dependent instruction. In this case, the numbered subscripts dictate the ordering on the nodes, e.g., i_0 precedes i_1 by at least one instruction. If i_0 and i_1 are the only nodes that have the label i, then because of precedence, it must be the case that i_1 is the dependent node and that i_0 is the resolving node.

If more than two nodes share the same letter label, then the node having the largest subscript represents the dependent instruction, and all other nodes represent resolving instructions for at least that node. Note that any of these nodes can also be dependents or resolvers of other nodes, but the labelings are chosen so as to focus attention on one specific dependent node.

All dependency arcs are given by an ordered pair of labels where the first label represents the dependent node, and the second label represents the resolving node. For example, the pair (i_1, i_0) represents a dependency arc that originates at the dependent node i_1, and terminates at the resolving node i_0. The dependency distance of any arc (i_x, i_y) is denoted by $D(i_x, i_y)$, and it represents the distance in instructions, $i_x - i_y$, between the nodes i_x and i_y. If i_y is the only resolving node of the dependent node i_x, then the dependency distance $D(i_x, i_y)$ can be abbreviated as D_i.

A. Basic Reductions

D_i is a dependency distance (measured in number of instructions) between a dependent instruction, generally i_1, and its resolving instruction, generally i_0, in an execution trace. In the absence of pipeline delays, D_i happens to correspond to the temporal distance (measured in clock cycles) between these two instructions as observed at any fixed segment of the pipeline. Since the goal of trace reduction is to obtain statistics that characterize the temporal aspects of the trace in the presence of delays, a method that infers accurate time delays based on the D_i is required.

Let Δ_x denote the *delay of instruction x,* defined as 1 less than the difference in machine cycles between the time that instruction $x - 1$ enters the first segment of the execution section of the pipeline and the time that instruction x enters the same segment of the pipeline. (The quantity 1 is subtracted since the nondelayed time between these events is 1 cycle.) Define the *effective dependency distance,* δ_i, as the temporal distance associated with the scalar distance D_i. Since in the absence of delays, the scalar distance D_i is equal to the temporal distance δ_i, then in the presence of delays,

$$\delta(i_1, i_0) = D(i_1, i_0) + \sum_{x=i_0+1}^{i_1} \Delta_x.$$

Thus, in general, the temporal distance δ associated with an arc i is equal to the scalar distance D of the arc plus the sum of all Δ delays associated with nodes x that are spanned by dependency arc i. This definition is broadened in Section VI.

Reduction 1: In a dependency graph, if there exists a node i_n such that $n > 1$, where i_n depends on n nodes, $i_0, i_1, \cdots, i_{n-1}$, then regardless of any other delays that are inherent to the graph, arcs (i_k, i_n) can be removed from the graph for all $0 \leq k \leq n - 2$. (See Fig. 3 for a depiction of this reduction.)

Proof: Since i_n is dependent on $i_0, i_1, \cdots, i_{n-1}$, instruction i_n cannot begin execution until all instructions i_x, $0 \leq x < n$, have completed execution. Since the instruction

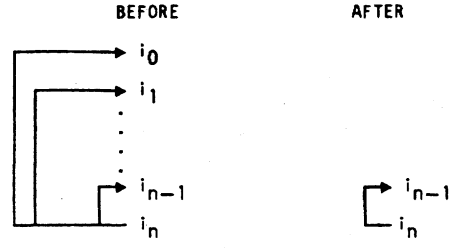

Fig. 3. Depiction of Reduction 1.

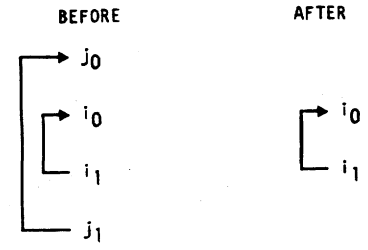

Fig. 4. Depiction of Reduction 2.

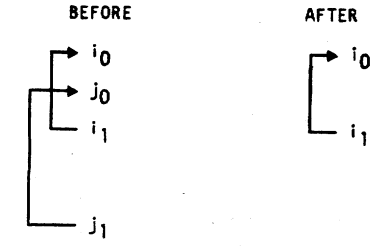

Fig. 5. Depiction of Reduction 3.

stream is executed in order at each pipeline segment, the last of the resolving instructions to complete execution will always be i_{n-1}, i.e., the closest to i_n of the resolving instructions. Thus, the penalty paid by i_n due to its data dependencies is completely determined by instruction i_{n-1}. ■

Reduction 2: If there is a nested dependency, i.e., arcs (i_1, i_0) and (j_1, j_0) exist where $j_0 \leq i_0 < i_1 \leq j_1$, then regardless of any other delays that are inherent to the graph, arc (j_1, j_0) can be removed from the graph. (See Fig. 4 for a depiction of this reduction.)

Proof: Since $j_0 \leq i_0$, the dependency (j_1, j_0) will be resolved no later than the time at which the dependency (i_1, i_0) is resolved. In order for instruction i_1 to be executed, the dependency (i_1, i_0) must be resolved. Since (j_1, j_0) is resolved no later than the time at which (i_1, i_0) is resolved, and since the resolution of (i_1, i_0) must occur prior to the time that i_1 begins execution, the dependency (j_1, j_0) will necessarily be resolved prior to the time at which i_1 begins execution. Since $i_1 \leq j_1$, instruction j_1 can begin execution no earlier than instruction i_1. Thus, (j_1, j_0) will have been resolved by the time that j_1 is ready to begin execution. Thus, j_1 will suffer no penalty due to the dependency (j_1, j_0), and hence, the arc (j_1, j_0) can be removed from the dependency graph. ■

Note that Reduction 2 subsumes Reduction 1, but it is convenient to retain Reduction 1 as a special case.

Reduction 3: If two dependency arcs intersect, i.e., if arcs (i_1, i_0) and (j_1, j_0) exist where $i_0 < j_0 < i_1 < j_1$, and if $D(i_1, i_0) \leq D(j_1, j_0)$, then if no delays (e.g., dependent nodes or branch targets) occur in the interval $[i_0 + 1, j_0]$, then the arc (j_1, j_0) can be removed from the graph. (See Fig. 5 for a depiction of this reduction.)

Proof: The proof is in three parts. First, the reduction is shown to hold when there are no other delays present. Next, the reduction is shown to hold when delays appear in the interval $[j_0 + 1, i_1]$. Finally, the reduction is shown to hold when delays appear in the interval $[i_1 + 1, j_1]$. Note that the interval $[i_0 + 1, j_0]$ is not considered, since no claim is made that Reduction 3 holds when delays are present in this interval.

As outlined above, first consider the simple case where, except for the delays induced by the dependencies (i_1, i_0) and (j_1, j_0), there are no delays of any sort represented in the entire interval $[i_0 + 1, j_1]$. In the absence of other delays, the temporal distance δ_i is equal to the scalar distance D_i plus the delay experienced at node i_1 due to the dependency arc (i_1, i_0), i.e., $\delta_i = D_i + \Delta_{i_1}$ where $\Delta_{i_1} = (N_E - D_i)^+$. Since the node i_1 is spanned by arc (j_1, j_0), the delay experienced at node j_1 is $\Delta_{j_1} = (N_E - D_j - \Delta_{i_1})^+$.

If $D_i \geq N_E$, then $\Delta_{i_1} = 0$, and $\Delta_{j_1} = (N_E - D_j)^+$. However, in this case, since $D_j \geq D_i$ and $D_i \geq N_E$, it follows that $D_j \geq N_E$, and thus, $\Delta_{j_1} = 0$ also. Therefore, in this case, the arc (j_1, j_0) does not represent a delay, and it can be removed from the graph. On the other hand, if $D_i < N_E$, then $\Delta_{i_1} = N_E - D_i$, and therefore, the delay experienced at node j_1 is $\Delta_{j_1} = (D_i - D_j)^+$. Since $D_j \geq D_i$, $\Delta_{j_1} = 0$, and hence, arc (j_1, j_0) can be removed from the graph.

Now consider the case where other delays exist in the interval $[j_0 + 1, i_1]$, and let the sum of these delays be Δ_α. Since this interval is spanned by both of the arcs (i_1, i_0) and (j_1, j_0), Δ_α appears as a component in both of the temporal distances δ_i and δ_j. Since $D_j \geq D_i$, it follows that $\delta_j \geq \delta_i$, and the same arguments (as used in the previous case) apply. Thus, arc (j_1, j_0) can be removed from the graph.

Finally, consider the case where other delays exist in the interval $[i_1 + 1, j_1]$, and let the sum of these delays be Δ_β. This interval is spanned by the arc (j_1, j_0), but it is not spanned by the arc (i_1, i_0). Thus, Δ_β is a component of the temporal distance δ_j, but it is not a component of the temporal distance δ_i. Since $D_j \geq D_i$, it follows that $\delta_j > \delta_i$, and the arguments used above still apply. Thus, arc (j_1, j_0) can be removed from the graph. ■

B. Trace Rendering Algorithm

For a new trace tape, the algorithm that is presented in this section makes one pass through the tape to construct an equivalent data-dependency graph for the trace. It then makes three successive passes through the dependency graph to apply Reductions 1, 2, and 3, respectively. After all reductions have been applied, it makes a final pass to collect statistics. The particular algorithm presented here is not the most efficient algorithm that can accomplish this task. It was chosen for this presentation only because it is simple to understand.

On the final statistical pass, the algorithm counts the number of taken branches in the graph, and collects information about the chains formed by the data-dependency arcs. For those chains that are composed of a single arc, no information about N_E or N_S is required, i.e., these chains can be represented by a

two-dimensional histogram. The dimensions of the histogram represent the dependency distance associated with the arc, and the number of taken branches spanned by the arc, respectively. For those chains that are composed of multiple arcs, a statistical histogram cannot be constructed without specific knowledge of N_E and N_S. Instead, this algorithm saves information about multiple-arc chains in the form of a list of stacks; a stack is constructed for each such chain, where each element of a stack represents an arc in the chain. This list of stacks can be constructed without specific knowledge of N_E or N_S.

After the statistical pass is made, all of the statistics are written to a save file. These statistics are: the length of the trace, the number of taken branches in the trace, the two-dimensional histogram that represents the single-arc chains, and the list of stacks that represent multiple-arc chains. These statistics characterize the trace tape; once they are collected, this process need not ever be repeated for the same tape, regardless of changes made to N_E or N_S.

Note that if the list of stacks is empty, then the entire trace is characterized by the two-dimensional histogram, and the number of taken branches in the trace. However, if the list of stacks is not empty, then the algorithm prompts the user to input the values of N_E and N_S, and it processes the stacks with these values, and increments the appropriate bins in the two-dimensional histogram. The final histogram characterizes the trace subject to specific values of N_E and N_S. Probabilities are estimated by dividing these final statistics by the length of the trace (in number of instructions).

Thus, the complexity of this particular algorithm is linear in the length of the trace the first time that the trace is processed (but note that five passes are made in this process). However, for subsequent runs, the algorithm merely reads in the intermediate statistics from the save file, and processes the list of stacks subject to new values of N_E and N_S. Thus, for subsequent runs, the complexity of the algorithm is linear in the number of multiple-arc chains that cannot be reduced by Reductions 1, 2, or 3. Brief descriptions of the steps used in the algorithm follow.

Step 1: Construct the data-dependency graph. The graph is implemented as a linear linked list, where each node in the graph represents an instruction on the trace tape. Each node has four attributes: a node name, a flag that indicates whether the node is the target of a taken branch, a list of pointers to resolving nodes, and a list of pointers to dependent nodes.

Scan the execution trace starting with the first instruction on the trace, and for that instruction, create the first node of the graph. Give the first node the name 1. For each new instruction record encountered, create a new node, and append it to the data dependency graph. If the name of the last node in the graph is i, then the next node to be appended is named $i + 1$. The name of the last node in the graph is the number of instructions in the trace. For each instruction record that is the target of a taken branch, mark the corresponding instruction node accordingly.

Meanwhile, maintain a list of all sink-operand locations that are encountered. With each sink-operand location, keep a pointer to the most recent instruction node that set the operand. For all source operands encountered, find the name of the source operand in the operand list, and use the associated pointer to establish the relevant dependency arcs between the dependent and the resolving instruction nodes. ■

For example, Fig. 1 is a graphic representation of the set of data dependencies and branches that is obtained by applying Step 1 to the associated trace of ten instructions.

Step 2: Apply Reduction 1. Since Reduction 1 is merely a statement of the fact that the delay experienced at a dependent node depends only on the proximity of the nearest resolving node, application of Reduction 1 is straightforward. Namely, for each node in the graph that has more than one resolving node in its resolving-node list, remove the dependency arcs associated with all but the closest resolving node. The closest resolving node is the node with the largest name. ■

Fig. 6 is a graphic representation that shows the extent to which Step 2 reduces the graph in Fig. 1.

Step 3: Apply Reduction 2. The purpose of this reduction is to remove nested dependency arcs. This step is performed with a stack since it is straightforward to identify nesting with a stack. Scan the nodes of the dependency graph in order. When a resolving node is found, first perform the analog of Step 2, i.e., if the node has more than one dependent node, then remove all of the associated arcs, except for the one from the nearest (lowest numbered) dependent node. Then push a pointer to the resolving node onto the stack.

When a dependent node is found, locate its respective resolving node on the stack. Remove from the graph all dependency arcs associated with resolving nodes that are older in the stack than this resolving node. (Since these "older" nodes are still stacked, their dependent nodes have not yet been encountered, hence, they represent outer levels of nesting.) Finally, remove from the stack the pointer to this resolving node, i.e., its dependent node (the current node) has been encountered.

After the application of Reductions 1 and 2, no node in the list will point to more than one dependent node, no node in the list will point to more than one resolving node, and there will be no nested arcs in the graph. ■

Fig. 7 is a graphic representation that shows the extent to which Step 3 further reduces the graph in Fig. 1.

Step 4: Apply Reduction 3. This step is also easily performed with a stack algorithm and a single in-order scan of the trace. As each resolving node is encountered, a pointer to it is stacked along with its associated dependency distance and a tag that is set when its associated dependent node is reached. As each resolving node j_0 is encountered, search the stack to find an "older" resolving node i_0 with the following properties: i) the tag at i_0 is not set (arcs i and j overlap), ii) the dependency distance of i is no longer than that of j, and iii) no other delays (i.e., branch targets or dependent nodes) occur between i_0 and j_0. If such an i_0 is found, then remove arc j from the dependency graph. When the tags for all elements of the stack have been set, then the stack can be purged, since all dependent nodes have been reached; this marks the end of an inseparable chain. ■

Fig. 8 is a graphic representation that shows the extent to which Step 4 further reduces the graph in Fig. 1.

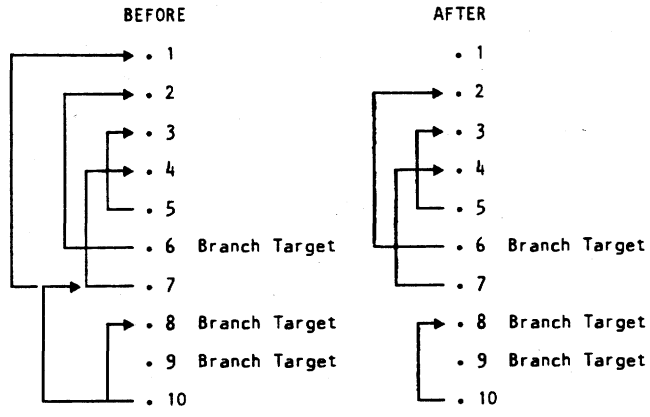

Fig. 6. Application of Reduction 1 to Fig. 1.

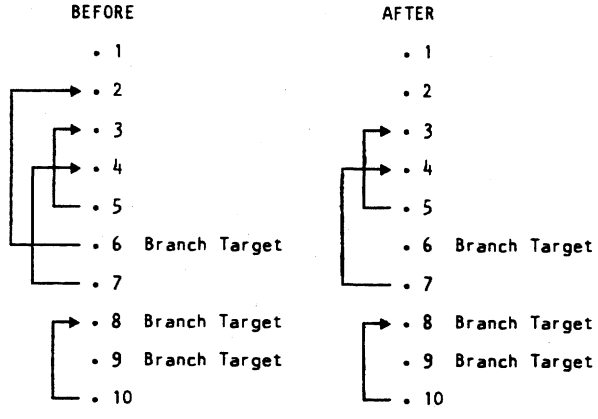

Fig. 7. Subsequent application of Reduction 2.

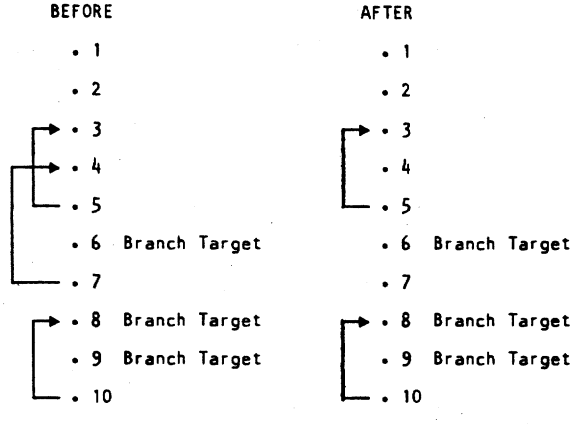

Fig. 8. Subsequent application of Reduction 3.

Step 5: Collect statistics. Define $\omega_{i,j}$ as the number of data-dependency arcs in the trace that span i nodes (excluding the resolving node but including the dependent node) of which j nodes are targets of taken branches. Let $p_{i,j}$ be the probability of such an arc, and estimate $p_{i,j}$ as the ratio of $\omega_{i,j}$ to the number of instructions in the trace. Similarly, define ω_b as the number of taken branches in the trace. Let p_b be the probability of a taken branch, and estimate p_b as the ratio of ω_b to the number of instructions in the trace. Initialize $\omega_{i,j} = 0$ for all reasonable values of i and j, and set $\omega_b = 0$.

Collecting statistics can once again be done with a stack in a two-step process. While scanning through the dependency graph, the steps are to find inseparable chains, and then to collect whatever data are suitable from each chain. As the dependency graph is scanned, the statistic ω_b is incremented whenever a node is found that is marked as the target of a taken branch.

When resolving nodes are encountered, they are stacked as was done in Step 4. When each dependent node is encountered, the corresponding tag in the stack is set. When all of the tags in the stack are set, the stack represents an inseparable chain. Note that the last dependent node of an inseparable chain can also be the first resolving node of the next inseparable chain (i.e., the graph is separable at such a node). In this event, a copy of the node is added to the graph, i.e., the node is split into two nodes: one that only reflects resolving nodes, and one that only reflects dependent nodes.

For each stack, the "oldest" element of the stack is used to increment the statistic $\omega_{i,j}$ according to its associated dependency distance (i), and the number of taken branches that are spanned by the corresponding dependency arc (j). This is only done for the "oldest" element of the stack, since it is only this element whose corresponding delay is unaffected by the dependency arcs represented by other elements of the stack.

If the stack contains exactly one element, then it can be discarded, since the dependency histogram has already been made to reflect the presence of the associated dependency arc. If the stack contains multiple elements, then it is saved on a list of stacks for further processing subject to N_E and N_S. ■

As observed from experiments, the vast majority of inseparable chains contain only a single arc once the trace has been reduced. Thus, the list of stacks that results from a given trace tape is typically small. If the list of stacks is empty, then Step 6 (as follows) is not required.

Step 6: Render statistics from the list of stacks. For this step, the specific values of N_E and N_S must be known. For each stack in the list, proceed as follows.

For each entry in the stack, let D represent the associated dependency distance (in number of instructions), let δ represent the corresponding temporal distance (in cycles), and let Δ represent the delay experienced at the dependent node (in cycles). Initially, assign $\delta = D$ for all entries in the stack.

Starting with the "oldest" entry in the stack, and continuing until all entries have been processed, proceed as follows. Increment the statistic $\omega_{\delta,j}$ on behalf of the entry, unless the entry is the "oldest" entry in the stack (recall that the statistic for the "oldest" entry was collected in Step 5). Next, if j is the number of taken branches spanned by the dependency arc represented by the current entry, then assign $\Delta = (N_E - \delta - j(N_S - 1))^+$ for the entry. For each entry in the stack that spans the dependent node of this entry, increment its temporal distance δ by this value of Δ. Another entry spans the dependent node of the current entry if its resolving node is "older" than the dependent node of the current entry, and the dependent node of the current entry is "older" than the dependent node of the other entry. ■

Within our experience, most traces fully reduce to inseparable chains, each composed of a single arc. For such traces, the statistics $\omega_{i,j}$ are independent of N_E and N_S, i.e., the statistics

that characterize such traces are independent of the hardware under study. However, for traces that do not fully reduce, it is only necessary to reprocess the list of stacks obtained in Step 5 when different values of N_E and N_S are being investigated.

For any pipeline that is characterized by N_E and N_S, and for any execution trace that is characterized by $p_{i,j}$ and p_b, the inverse bandwidth of the pipeline running that trace is

$$BW^{-1} = 1 + p_b(N_S - 1) + \sum_{i=0}^{N_E-1} \sum_{j=0}^{i} p_{i,j}(N_E - i - j(N_S - 1))^+.$$

For example, the fully reduced trace-segment in Fig. 7 is characterized by the statistics

$$p_b = 3/10,\ p_{2,0} = 1/10,\ \text{and}\ p_{2,1} = 1/10$$

independently of N_E and N_S. For the values $N_E = N_S = 5$, the inverse bandwidth as computed from the formula above is

$$\begin{aligned} BW^{-1} &= 1 + \frac{3}{10}(5-1) + \frac{1}{10}(5-2)^+ + \frac{1}{10}(5-2-1(5-1))^+ \\ &= 1 + 1.2 + 0.3 + 0.0 \\ &= 2.5. \end{aligned}$$

This value agrees with the value obtained via hand simulation (see Fig. 2).

In the next section, the effectiveness of these reductions is demonstrated on two sample execution traces. Statistics are given on the complexity of the dependency chains (i.e., the number of arcs in the chains) after each reduction is applied, and the resulting statistics that characterize the traces are shown.

IV. Examples of Trace Reduction

To demonstrate the effectiveness of trace reduction, two execution traces taken from an IBM System 370 processor were reduced in the manner specified above. One of them is an eigenvalue kernel which was monitored over 54 693 instructions, and the other is a Gaussian elimination kernel which was monitored over 63 612 instructions. In these cases, memory locations, registers, and condition codes are considered to be the data items that give rise to dependencies. The tracer provided the address, the opcode, and the data references made by each instruction during execution.

The first set of statistics, presented in Table I, shows the effectiveness of each reduction. The dependencies in each of the traces initially formed one inseparable chain. Even after applying Reduction 1, neither of the chains was separated. In the case of the eigenvalue kernel, the chain comprised 39 852 dependency arcs. In the case of the Gaussian elimination kernel, the chain comprised 49 174 dependency arcs.

The application of Reduction 2 separated the eigenvalue kernel into 32 253 inseparable chains, only 66 of which comprised more than a single arc. The Gaussian elimination kernel was separated into 41 361 inseparable chains, only 323 of which comprised more than a single arc. For both traces, Reduction 2 left no chains composed of more than three arcs. Reduction 3 completely reduced the eigenvalue kernel into

TABLE I
EFFECTIVENESS OF TRACE REDUCTIONS

	EIGEN		GAUSS	
Trace Length	54693		63612	
Reduction 1	1 chain of 39852		1 chain of 49174	
Reduction 2	# arcs	# chains	# arcs	# chains
	1	32187	1	41038
	2	65	2	322
	3	1	3	1
Reduction 3	# arcs	# chains	# arcs	# chains
	1	32254	1	41346
	2	0	2	16

32 254 chains, each consisting of a single arc. For this kernel, the statistics $\omega_{i,j}$ are independent of N_E and N_S. The Gaussian elimination kernel was separated into 41 346 chains each consisting of a single arc, but 16 chains each consisting of two arcs were left. Thus, in addition to the statistics $\omega_{i,j}$ and ω_b, the Gaussian elimination kernel requires a list of 16 stacks, each with two elements, to characterize it independently of N_E and N_S.

The second set of statistics, presented in Table II, is the set of first-order sufficient statistics for the kernels. The upper part of the table is a histogram of dependency distance versus the number of taken branches spanned, i.e., these are the statistics $\omega_{i,j}$. For example, the reduced eigenvalue dependency graph contains 304 dependency arcs of distance 3 that span a single taken branch. The statistic $p_{i,j}$ is obtained by dividing the entries in this histogram by the trace length.

The "extra delays" shown in the table for the Gaussian elimination kernel correspond to those 16 arcs that are the second arcs of the inseparable chains composed of two arcs. For any specific N_E, these delays could appear in the table of $\omega_{i,j}$ statistics, however, they are kept separate to make the statistics more general. Note that for chains composed of two arcs that cannot be split by Reduction 3, if the dependency distances of the arcs are D_i and D_j, and if no taken branches are spanned, then the delay experienced at the second dependent node is equal to $(N_E - D_j - (N_E - D_i)^+)^+$.

In all 16 such cases that appear in the Gaussian elimination kernel, the first arc is characterized by $D_i = 7$, the second arc is characterized by $D_j = 6$, and no taken branches are spanned. Since the delays associated with these first arcs are independent of the presence of the second arcs, the first arcs appear in Table II in the distance = 7 row, branches = 0 column. The penalties associated with the second arcs appear as "extra delays" in the table. Since for these arcs, $(N_E - 6 - (N_E - 7)^+)^+$ is 0 for $N_E \leq 6$ and is 1 for $N_E \geq 7$ these 16 arcs contribute 16 cycles of delay over-and-above the delays attributable to the $\omega_{i,j}$ statistics in pipelines having $N_E \geq 7$. The net effect of these irreducible chains is thus to add 16/63 612 to the BW^{-1} calculated from the $\omega_{i,j}$ and p_b in the table for cases where $N_E \geq 7$.

The final statistic shown in the table is the number of taken branches that appear in the traces. The statistic p_b is obtained by dividing the total number of branches that appear in the

TABLE II
SUFFICIENT STATISTICS FOR EXAMPLE TRACES

		EIGEN			GAUSS		
		Number of Branches			Number of Branches		
Dependency Histogram		0	1	2	0	1	2
Distance	1	28494	115	0	35989	520	0
	2	1020	991	254	3522	916	17
	3	333	304	1	26	106	1
	4	376	21	0	22	1	0
	5	78	0	0	209	0	0
	6	13	0	0	0	0	0
	7	0	0	0	16	0	0
Extra Delays		0			16 if $N_E \geq 7$		
Taken Branches		4027			7688		

trace by the trace length. This set of statistics fully characterizes the traces.

V. Optimal Pipeline Length as a Function of Trace Statistics

Thus far, it has been shown that any execution trace can be reduced in a manner that allows a set of first-order statistics to characterize the trace exactly. Given that a processor designer has a set of traces that is believed to characterize a workload, it is possible to use trace statistics to choose the optimal pipeline length for that workload with a given technology in mind.

Since the performance of a pipeline is measured in terms of instructions per second, performance is decomposable into aspects of machine organization (which determine instructions per cycle), and aspects of technology (which determine cycle time). For the purposes of this paper, it is assumed that the machine organization is fixed as specified above, i.e., it is a pipeline comprising a setup section and an execution section.

The one unspecified parameter is the length of the pipeline, or the granularity into which tasks are to be decomposed. As is apparent from the equation for inverse bandwidth pertaining to this machine organization, the cycles per instruction increases monotonically with the length of the pipeline due to data dependencies and branches. However, as the length of the pipeline is increased (or equivalently, the granularity into which tasks are decomposed becomes finer), the cycle time of the pipeline decreases. Thus, as the length of the pipeline increases, extrinsic delays degrade cycles per instruction, while intrinsically, cycles per second is improved. Recent work by Kunkel and Smith reports and analyzes such a tradeoff for vector machines [19]. In the following sections, it is shown how to trade off these two aspects of performance to arrive at a pipeline length that delivers maximum bandwidth for a given set of trace statistics.

In Section V-A, the impact of pipeline granularity on cycles per instruction is analyzed. For any specific pipeline having specific values of N_E and N_S, the pipeline can be characterized by the ratio of these lengths, $E/S = N_E/N_S$, and by the value n where $N_E = nE$ and $N_S = nS$. With this view, it is assumed that changing the pipeline granularity is merely a matter of varying n, i.e., it is assumed that the characteristic length ratio E/S is invariant in n for any base design.

The inverse bandwidth for a pipeline that is characterized by E/S and n is denoted $BW_{E,S}^{-1}(n)$. Inverse bandwidth is the sum of three terms: the constant 1, a term for branch delay, and a term to account for data dependencies in the presence of branch delay. Of these three terms, it is only the data dependency term that is unwieldy; it contains a double summation, and the nonlinear operator $(x)^+$. For convenience, the data-dependency term is denoted $D_{E,S}(n)$. In Section V-A, data from Table II are used to obtain a general estimator for this term subject to the eigenvalue benchmark. This is done by using the difference $D_{E,S}(n) - D_{E,S}(n-1)$ to obtain a recursion in n. This recursion facilitates making an estimate of $BW_{E,S}^{-1}(n)$ from an exact solution to $D_{E,S}(k)$ where $k \ll n$.

In Section V-B, the impact of pipeline granularity on cycle time is analyzed. Cycle time can be obtained as a function of n for any specific technology; however, it is more generally useful to have a function of n that is less dependent on the specific technology used. That is, a function of n is sought that is independent of the specific cycle time that is obtainable with a particular technology.

For a given technology, let C be the longest path (in nanoseconds) through a nonpipelined machine, and let L be the time (in nanoseconds) associated with latching. Let γ be the ratio C/L. The single parameter γ is sufficient for specifying all technologies that are characterized by C/L.

It is shown that the speedup of a pipelined machine, denoted $\Psi(n)$, over a nonpipelined machine is the product of the pipeline bandwidth and a function of the parameters n and γ. That is, speedup is a function that is separable into two components: the function $BW_{E,S}^{-1}(n)$ which can be estimated as shown in Section V-A, and a function that is dependent on the technology parameter γ. Since γ is independent of the specific cycle-time that is offered by a technology, the speedup function provides a convenient means for determining the optimal pipeline granularity.

In Section V-C, the two components of $\Psi(n)$ are reviewed. The first component is a function of only γ and n, and it is monotonically increasing in n. The second component, $BW_{E,S}(n)$, is monotonically decreasing in n. The product of these two components, $\Psi_{E,S}(n)$, forms a bitonic sequence. This is demonstrated by showing that the derivative of $\Psi_{E,S}(n)$ has a double root, and no other roots. The double root is a maximum (optimal design point) determined by $n_{\text{opt}}^2 = \gamma\alpha_{E,S}(k)$, where $\alpha_{E,S}(k)$ is a function of the statistics that are gathered via trace reduction evaluated at some pipeline length,

$k(E + S)$. The eigenvalue benchmark statistics are applied to this result for illustrative purposes.

Since $\alpha_{E,S}(k)$ is fixed for any specific E/S subject to any specific benchmark and an arbitrarily chosen k, the solution for n_{opt} is only a function of γ. This fact is used to derive the sequence γ_i whose elements define ranges of γ for which particular values of n are optimal. In particular, γ_n is that point at which $\Psi_{E,S}(n) = \Psi_{E,S}(n + 1)$ when $\gamma = \gamma_n$. It is shown that the sequence γ_i as derived yields exact values of γ_n for n greater than a specified threshold value.

A. Cycles Per Instruction as a Function of Pipeline Length

As derived in Section III, the formula for inverse bandwidth is

$$BW^{-1} = 1 + p_b(N_S - 1) + \sum_{i=0}^{N_E-1} \sum_{j=0}^{i} p_{i,j}(N_E - i - j(N_S - 1))^+.$$

This formula is the sum of three components, namely: the constant 1, a term associated with branch delay, and a term to account for data dependency degradation in the presence of branch delay. The constant 1 is inherent to any pipeline that decodes at most 1 instruction per cycle, i.e., the upper bound on pipeline performance is assumed to be 1 cycle per instruction. The branch-delay term is a function only of the branch probability, and the length of setup N_S. The most complex of the terms is the data-dependency degradation. This term involves the two-dimensional statistic, $p_{i,j}$, as well as the lengths of the setup and the execution pipeline sections, N_S and N_E, respectively. It is this last term that is the primary focus of this section.

Let E/S be the length ratio N_E/N_S. Assume that any increases in the length of the pipeline (i.e., any refinements in the pipeline granularity) are distributed in both sections of the pipeline so as to preserve this ratio. Without loss of generality, make the restriction that either $E = 1$ and/or $S = 1$, or that E and S are relatively prime. Thus, when a pipeline length is chosen, n is defined so that the length of setup is $N_S = nS$, the length of execute is $N_E = nE$, and the overall length of the pipeline is $N = N_S + N_E = n(S + E)$. Let the inverse bandwidth for the pipeline be denoted as $BW_{E,S}^{-1}(n)$, so that all pipelines having the characteristic E/S can be described by the single parameter n. Similarly, denote the data-dependency component for the pipeline as $D_{E,S}(n)$. Thus,

$$BW_{E,S}^{-1}(n) = 1 + p_b(nS - 1) + D_{E,S}(n)$$

where

$$D_{E,S}(n) = \sum_{i=0}^{nE-1} \sum_{j=0}^{i} p_{i,j}(nE - i - j(nS - 1))^+.$$

Table III shows the number of penalty cycles associated with data dependencies in the eigenvalue benchmark over a range of values of nS and nE. These data are plotted in Fig. 9. These numbers were computed by using the statistics in Table II; any particular value of $D_{E,S}(n)$ can be obtained by dividing the appropriate table entry by the tracelength 54 693.

TABLE III
DATA DEPENDENCY PENALTY CYCLES FOR EIGENVALUE BENCHMARK

nE \ nS	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	28609	28494	28494	28494	28494	28494	28494	28494	28494
3	59483	58123	58008	58008	58008	58008	58008	58008	58008
4	90995	89076	87970	87855	87855	87855	87855	87855	87855
5	122904	120963	119299	118193	118078	118078	118078	118078	118078
6	154891	152950	151010	149600	148494	148379	148379	148379	148379
7	186891	184950	183009	181324	179914	178808	178693	178693	178693
8	218891	216950	215009	213069	211638	210228	209122	209007	209007
9	250891	248950	247009	245068	243383	241952	240542	239436	239321

Note that for $E/S \leq 1$, the value $D_{E,S}(n)$ is independent of nS. This property occurs because no data dependency that spans a taken branch has performance implications when $E/S \leq 1$ since all such data dependencies are obviated by branch delays. Thus, $D_{E,S}(n)$ is determined by the statistic $p_{i,0}$ if $S \geq E$. Since a closed form estimator for $BW_{E,S}^{-1}(n)$ is sought, and since it is only the $D_{E,S}(n)$ component that is not trivial, the normalized difference $(D_{E,S}(n) - D_{E,S}(n - 1))/D_{E,S}(n)$ was computed for various fixed values of the ratio E/S using the numbers in Table III to obtain the approximations shown in Table IV. These approximations are plotted in Fig. 10. As demonstrated in this table, an approximate estimator for this difference given any E/S is

$$\frac{D_{E,S}(n) - D_{E,S}(n-1)}{D_{E,S}(n)} \simeq \frac{E}{nE - 1}.$$

Thus, an approximation to the data-dependency penalty can be expressed recursively as follows.

$$D_{E,S}(n) \simeq \frac{nE - 1}{(n-1)E - 1} D_{E,S}(n-1).$$

This approximation is valid over any set of statistics $p_{i,j}$ that tends to be concentrated over small values of i and j. Furthermore, if $p_{i,j} = 0$ for $i \geq nE$ and $j \geq nS$, then the approximation is extremely good, i.e., it becomes more accurate as n increases. Assume that $D_{E,S}(k)$ has been computed exactly for some value k, and that an estimator for $D_{E,S}(n)$ is desired where $n > k$. Using the recursion above,

$$D_{E,S}(n) \simeq \left(\prod_{i=k+1}^{n} \frac{iE - 1}{(i-1)E - 1} \right) D_{E,S}(k) = \frac{nE - 1}{kE - 1} D_{E,S}(k)$$

where $D_{E,S}(k) > 0$, i.e., kE is greater than the smallest i for which $p_{i,j} > 0$. Note that the best estimator is the one for which $k = n - 1$, and that an exact solution is obtained in the case that $k = n$. Thus, a family of estimators in k for inverse bandwidth as a function of n is given by

$$BW_{E,S}^{-1}(n) \simeq 1 + p_b(nS - 1) + \frac{nE - 1}{kE - 1} D_{E,S}(k)$$

B. Speedup as a Function of Pipeline Length

In Section V-A, a class of estimators was built that determines the increase in cycles per instruction as a function of pipeline length for a known workload given the fixed ratio

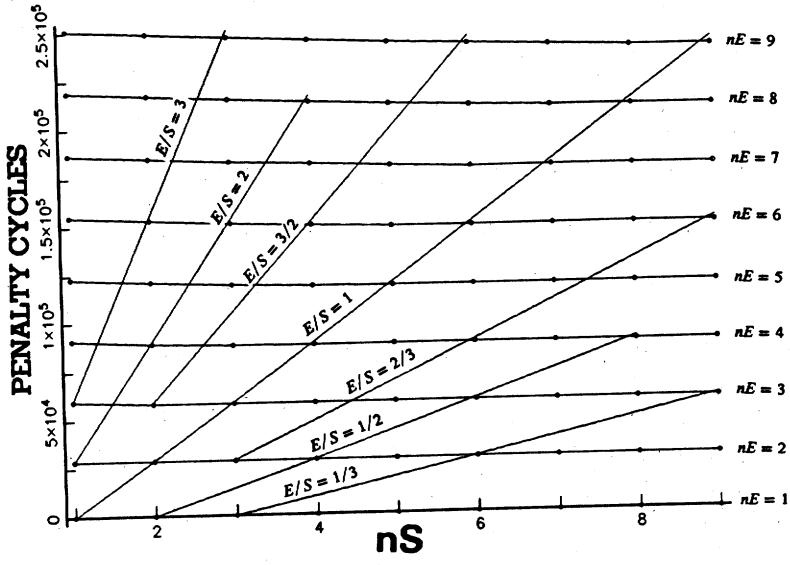

Fig. 9. Data dependency penalty cycles for various nE and nS.

TABLE IV
THE NORMALIZED DIFFERENCE $(D_{E,S}(n) - D_{E,S}(n-1))/D_{E,S}(n)$

ACTUAL

n \ E/S	1/4	1/3	1/2	2/3	3/4	1	4/3	3/2	2	3	4
2	1.000	1.000	1.000	0.676	0.609	1.000	0.582	0.611	0.679	0.611	0.581
3	–	0.509	0.509	0.408	–	0.509	–	0.382	0.410	0.381	–
4	–	–	0.340	–	–	0.340	–	–	0.291	–	–
5	–	–	–	–	–	0.256	–	–	–	–	–

APPROXIMATE

n \ E/S	1/4	1/3	1/2	2/3	3/4	1	4/3	3/2	2	3	4
2	1/1	1/1	1/1	2/3	3/5	1/1	4/7	3/5	2/3	3/5	4/7
3	–	1/2	1/2	2/5	–	1/2	–	3/8	2/5	3/8	–
4	–	–	1/3	–	–	1/3	–	–	2/7	–	–
5	–	–	–	–	–	1/4	–	–	–	–	–

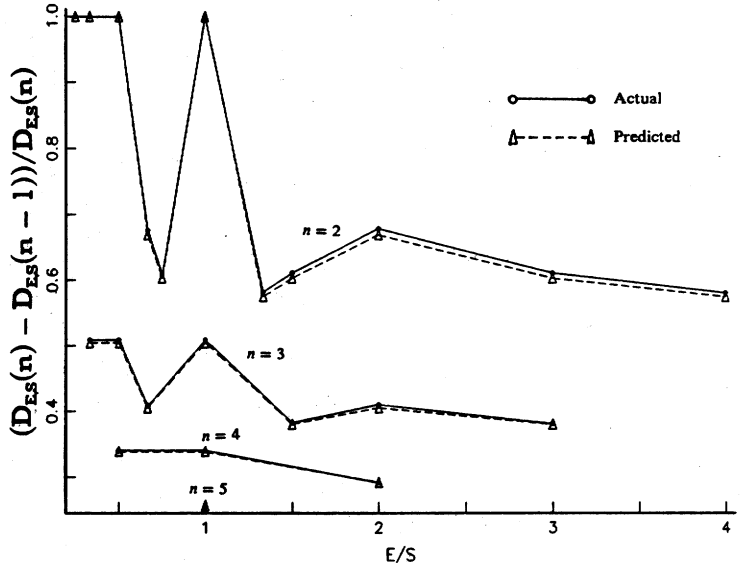

Fig. 10. Approximations to $(D_{E,S}(n) - D_{E,S}(n-1))/D_{E,S}(n)$ for various E/S and n.

E/S. In this current section, the fact that increasing the pipeline length decreases the cycle time is taken into account, and an equation for speedup as a function of pipeline length is derived subject to any given function $BW_{E,S}^{-1}(n)$. Since cycle time decreases monotonically as a function of n to an asymptote (i.e., the shortest possible cycle time must include latch delay), and since $BW_{E,S}^{-1}(n)$ increases monotonically as a function of n for any E/S, there exists a specific n that maximizes speedup for any specific workload given a value of E/S. This optimization is pursued in Section V-C.

Let BW_{ser}^{-1} denote the inverse bandwidth for a serial (nonpipelined) machine, and let BW_{pip}^{-1} denote the inverse bandwidth for a pipelined machine. Assume that $BW_{\text{ser}}^{-1} = 1$ cycle per instruction, since branches and data dependencies do not degrade the performance of such a machine. From the basic approach used by Larson and Davidson [20], let C denote the longest path (in nanoseconds) through the serial machine, i.e., C represents the longest path that would exist in a machine with all latches removed. Assume that the penalty associated with a single latch is L nanoseconds. L should include all aspects of latching, such as setup and hold time, clock skew, etc.; valuable insight is offered by Kunkel and Smith [19] and by Unger and Tan [21] as to the actual overheads associated with latching in pipelines. The clock period for a serial machine (assuming that the final output is latched on every cycle) is then $\Phi_{\text{ser}} = C + L$ nanoseconds, and the throughput of the machine is given by $T_{\text{ser}} = BW_{\text{ser}}/\Phi_{\text{ser}}$ instructions per nanosecond.

Since a pipelined machine having N stages has the same logical pathlength C, and since each stage includes a latch, the clock period for a pipelined machine is $\Phi_{\text{pip}} = (C/N) + L$ nanoseconds. The pipelined throughput is thus $T_{\text{pip}} = BW_{\text{pip}}/\Phi_{\text{pip}}$ instructions per nanosecond. Speedup, denoted Ψ, is defined to be the ratio between the pipelined and the serial throughputs, namely,

$$\Psi = \frac{T_{\text{pip}}}{T_{\text{ser}}} = \frac{BW_{\text{pip}}(C+L)}{BW_{\text{ser}}[(C/N)+L]} = BW_{\text{pip}} \cdot \left[\frac{N(C+L)}{C+LN}\right].$$

Since the exact values of C and L may not be known at the start of a design, it simplifies matters to define a technology parameter that is the ratio between them. Let $\gamma = C/L$ denote this parameter. Although C and L may not be known, it is reasonable to expect that given a technology, and given an idea of the circuit complexity that is to be designed, a designer should be able to estimate γ. As will be demonstrated later, it is not essential to know an exact value of γ in order to select the optimal pipeline length, since optimal lengths are applicable over ranges of the parameter γ. Defined in terms of γ, the equation for speedup is

$$\Psi = BW_{\text{pip}} \cdot \left[\frac{N(\gamma+1)}{N+\gamma}\right].$$

C. Optimal Pipeline Length as a Function of Job Load

Recall that the pipeline under consideration is specified by the known ratio E/S, and that the length of the pipeline is given by $N = n(E + S)$. In Section V-A, an estimator for inverse bandwidth was developed for an E/S pipeline of length N, namely $BW_{E,S}^{-1}(n)$. The speedup of an E/S pipeline as a function of n can then be estimated by

$$\Psi_{E,S}(n) = BW_{E,S}(n) \cdot \left[\frac{n(S+E)(\gamma+1)}{n(S+E)+\gamma}\right] \simeq \left(1 + p_b(nS-1) + \frac{nE-1}{kE-1} \cdot D_{E,S}(k)\right)^{-1} \cdot \left[\frac{n(S+E)(\gamma+1)}{n(S+E)+\gamma}\right]$$

for some computed value of $D_{E,S}(k)$. Since this speedup is relative to instructions per second, the best performing pipeline is that pipeline characterized by the particular n that maximizes $\Psi_{E,S}(n)$.

This estimate is used from here on. For any fixed k, the function $\Psi_{E,S}(n)$ has exactly one critical point (maximum), and thus, it describes a bitonic sequence. This is demonstrated by showing that the derivative of $\Psi_{E,S}(n)$ has one double root, and no other roots, as follows. First, a separation of variables on the equation above yields

$$\Psi_{E,S}(n) = \frac{\gamma+1}{(S+E)\left(1-p_b-\frac{D_{E,S}(k)}{kE-1}\right)-\gamma\left(p_bS+\frac{ED_{E,S}(k)}{kE-1}\right)} \cdot \left[\frac{(S+E)\left[\frac{(kE-1)(1-p_b)-D_{E,S}(k)}{(kE-1)p_bS+ED_{E,S}(k)}\right]}{n+\frac{(kE-1)(1-p_b)-D_{E,S}(k)}{(kE-1)p_bS+ED_{E,S}(k)}} - \frac{\gamma}{n+\frac{\gamma}{S+E}}\right].$$

The derivative with respect to n is

$$\frac{\partial \Psi_{E,S}(n)}{\partial n} = \frac{\gamma+1}{(S+E)\left(1-p_b-\frac{D_{E,S}(k)}{kE-1}\right)-\gamma\left(p_bS+\frac{ED_{E,S}(k)}{kE-1}\right)} \cdot \left[\frac{-(S+E)\left[\frac{(1-p_b)(kE-1)-D_{E,S}(k)}{p_bS(kE-1)+ED_{E,S}(k)}\right]}{\left(n+\frac{(1-p_b)(kE-1)-D_{E,S}(k)}{p_bS(kE-1)+ED_{E,S}(k)}\right)^2} + \frac{\gamma}{\left(n+\frac{\gamma}{S+E}\right)^2}\right].$$

and the roots of the derivative, denoted n_{opt}, are determined by

$$n_{opt}^2 = \frac{\gamma((1-p_b)(kE-1)-D_{E,S}(k))}{(S+E)(p_bS(kE-1)+ED_{E,S}(k))}.$$

Thus, $\Psi_{E,S}(n)$ increases monotonically to a maximum point (the double root as determined above), and then it decreases monotonically.

Note, however, that $\Psi_{E,S}(n)$ only has a physical interpretation for integer values $n \geq 1$. Therefore, in the region $n \geq 1$, the sequence determined by $\Psi_{E,S}(n)$ is: monotonic if the double root is ≤ 1, bitonic if the double root is ≥ 2, and either monotonic or bitonic if the double root is >1 and <2. If $\Psi_{E,S}(n)$ is monotonic in the range of interest, then the best choice of n is $n = 1$. If $\Psi_{E,S}(n)$ is bitonic in the range of interest, then the best choice of n is either $\lfloor n_{opt} \rfloor$ or $\lceil n_{opt} \rceil$.

Note that n_{opt}^2 is the product of the technology parameter γ, and a function of the trace statistics subject to a given length ratio E/S. In particular, define

$$\alpha_{E,S}(k) = \frac{1}{S+E}\left[\frac{(kE-1)(1-p_b)-D_{E,S}(k)}{(kE-1)p_bS+ED_{E,S}(k)}\right]$$

so that $n_{opt} = \sqrt{\gamma\alpha_{E,S}(k)}$. For any given E/S subject to any specific workload and an arbitrarily chosen k, the value of $\alpha_{E,S}(k)$ is fixed, and hence, the value of n that maximizes speedup is only a function of γ.

For example, suppose that for a particular pipeline, $E/S \simeq 1$, i.e., it is known that the length of setup is approximately equal to the length of execution. Furthermore, suppose that the eigenvalue benchmark is believed to characterize the workload. Then,

$$\alpha_{1,1}(k) = \frac{1}{2}\left[\frac{(k-1)(1-p_b)-D_{1,1}(k)}{(k-1)p_b+D_{1,1}(k)}\right]$$

where $p_b = 4027/54\,693$ as shown in Table II, and $D_{1,1}(k)$ is the estimator obtained by dividing the (k, k) entry in Table III by the tracelength for $k > 1$. Arbitrarily, choose $k = 2$. Then from Table III, $D_{1,1}(2) = 28\,494/54\,693$, and $\alpha_{1,1}(2) = 0.34089$. The optimal value of n is then estimated to be either the floor or the ceiling of $n_{opt} = \sqrt{0.34089 \cdot \gamma}$. Note that the choice $k = 3$ yields a slightly different estimator, $n_{opt} = \sqrt{0.32790 \cdot \gamma}$.

While the methodology above provides a value of n_{opt}, it does not positively identify which of $\lfloor n_{opt} \rfloor$ or $\lceil n_{opt} \rceil$ is truly the optimal choice for n. Furthermore, since $D_{E,S}(n)$ is merely an estimate that is computed from a known $D_{E,S}(k)$, the value of k that is chosen may generate more uncertainty in the optimal choice of n depending on γ. For example, if $\gamma = 75$ in the example above, then the choice $k = 2$ yields $n_{opt} = 5.056$, and the choice $k = 3$ yields $n_{opt} = 4.959$ (i.e., the best choice of n could be 4, 5, or 6).

For small values of n, this does not pose a serious problem, since the exact values of $\Psi_{E,S}(n)$ can be readily computed and compared. However, when γ is very large, the inaccuracy inherent to $\alpha_{E,S}(k)$ may be such as to position n_{opt} far away from the actual best choice of n. Inaccuracy in $\alpha_{E,S}(k)$ arises because the nonlinear operator $(x)^+$ used in the computation of $D_{E,S}(n)$ causes the difference $D_{E,S}(n) - D_{E,S}(n-1)$ to vary with n when n is small.

For large values of n, the difference $D_{E,S}(n) - D_{E,S}(n-1)$ is a constant. This property allows for an exact solution for optimal n when n is large. Let $I = \max\{i\}$ for which $p_{i,j} \neq 0$. Then for $n \geq (I+1)/E$, the difference $D_{E,S}(n+1) - D_{E,S}(n)$ can be written as

$$D_{E,S}(n+1) - D_{E,S}(n) = \sum_{i=0}^{I}\sum_{j=0}^{i} p_{i,j}([(n+1)(E-jS) - (i-j)]^+ - [n(E-jS)-(i-j)]^+).$$

Note that since $i \geq j$ is all terms, then for a term to be nonzero, it must be the case that $j < E/S$. The difference in the equation above is nonlinear only if both of the conditions $(n+1)(E-jS) > (i-j)$ and $n(E-jS) < (i-j)$ hold for some $i \geq j$. However, if $n \geq (i-j)/(E-jS)$ for every $\{(i, j)\}$ for which $i \geq j$, $j < E/S$, and $p_{i,j} \neq 0$, then the difference $D_{E,S}(n+1) - D_{E,S}(n)$ is a constant. That is, for large values of n,

$$D_{E,S}(n+1) - D_{E,S}(n) = \sum_{i=0}^{I}\sum_{j=0}^{i} p_{i,j}(E-jS) = K, \text{ a constant.}$$

For example, in the $E/S = 1$ pipeline, the constraint $j < E/S$ dictates that only the statistics $p_{i,0}$ are relevant. In the eigenvalue benchmark, $p_{i,0} = 0$ if $i > 6$, and therefore, $I = 6$. The constraint $n \geq (i-j)/(E-jS)$ subject to $j = 0$ and $i \leq 6$ dictates that $D_{E,S}(n+1) - D_{E,S}(n)$ is a constant for all $n \geq 6$. In particular, for the eigenvalue benchmark running on an $E/S = 1$ pipeline, $K = \Sigma p_{i,0} = 30\,314/54\,693$. In the remainder of this section, K is used to solve for ranges of γ in which particular values of n maximize pipeline performance.

Define the sequence γ_i such that $\Psi_{E,S}(n) = \Psi_{E,S}(n+1)$ at the point $\gamma = \gamma_n$. That is, each particular γ_n defines the region boundary for which Ψ is maximized at the point n if $\gamma = \gamma_n - \epsilon$, and Ψ is maximized at the point $n+1$ if $\gamma = \gamma_n + \epsilon$, where ϵ is arbitrarily small. For $n \geq (i-j)/(E-jS)$ subject to the constraints on i and j as stated above, it was shown that $D_{E,S}(n+1) = D_{E,S}(n) + K$. For a particular n, the boundary γ_n is that value of γ that satisfies $\Psi_{E,S}(n+1) = \Psi_{E,S}(n)$. Equating these two values of Ψ in terms of K at the point γ_n yields

$$\frac{1+p_b(nS-1)+D_{E,S}(n)+[p_bS+K]}{1+p_b(nS-1)+D_{E,S}(n)} = \left[\frac{(n+1)(S+E)(\gamma_n+1)}{(n+1)(S+E)+\gamma_n}\right] \cdot \left[\frac{n(S+E)+\gamma_n}{n(S+E)(\gamma_n+1)}\right].$$

Straightforward manipulation results in

$$\gamma_n = \frac{(S+E)(p_bS+K)}{1-p_b+D_{E,S}(n)-nK} \cdot n(n+1).$$

Note that although the denominator in the equation for γ_n appears to be a function of n, in fact it is a constant. Since $D_{E,S}(n+1) = D_{E,S}(n) + K$ when n is sufficiently large, the

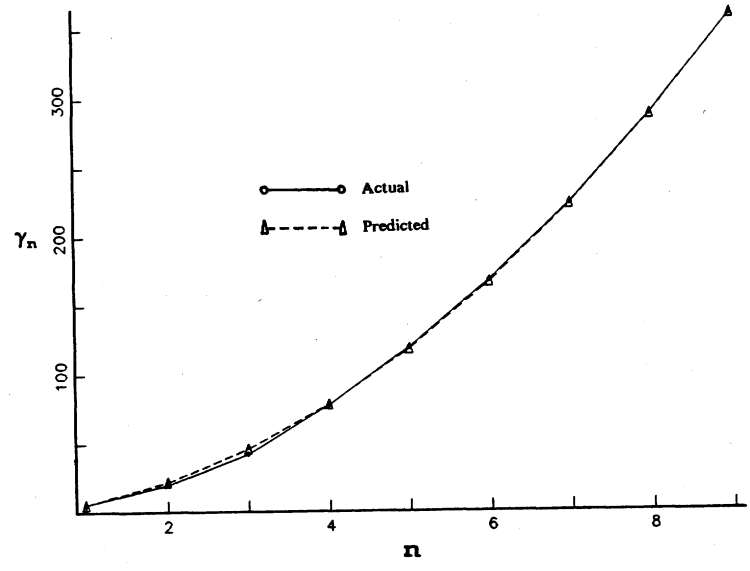

Fig. 11. Predicted and actual values of γ_n for $E/S = 1$.

difference $D_{E,S}(n) - nK$ is invariant for large n. For example, in the $E/S = 1$ pipeline subject to the eigenvalue benchmark, it was shown that $D_{E,S}(n + 1) = D_{E,S}(n) + K$ for $n \geq 6$. Therefore, for $n \geq 6$, the difference $D_{E,S}(n) - nK$ is equal to $D_{1,1}(6) - 6K$. From Table III, $D_{1,1}(6) = 148\ 379/54\ 693$, and from above, $K = 30\ 314/54\ 693$. Since $p_b = 4027/54\ 693$, the sequence γ_i is given by $\gamma_n = (68\ 682/17\ 161)n(n + 1)$, or equivalently, $\gamma_n = 4.00221n(n + 1)$ for the $E/S = 1$ pipeline subject to the eigenvalue benchmark.

Note that γ_n is an exact solution for $n \geq 6$. For example, $\gamma_6 = 168.093$ for this pipeline. The significance of this is that if it is known that the actual value of $\gamma = C/L$ is exactly 168, then the best $E/S = 1$ pipeline that can be built has six setup segments and six execute segments. However, if the actual value of γ is exactly 169, then the best pipeline that can be built has seven setup segments and seven execute segments. Fig. 11 is a plot of γ_i as computed above for $E/S = 1$. In Table V, the constants K and the associated sequences γ_n are listed for various E/S subject to the eigenvalue benchmark. These sequences are plotted in Fig. 12.

TABLE V
THE SEQUENCES γ_n FOR VARIOUS E/S

E/S	K	γ_n	EXACT
1	30314/54693	4.00221*n(n+1)	n≥6
2	62059/54693	12.79166*n(n+1)	n≥3
2/3	60628/54693	21.18437*n(n+1)	n≥3
3	94059/54693	25.31578*n(n+1)	n≥2
3/2	92373/54693	32.39790*n(n+1)	n≥3
4	126059/54693	41.96864*n(n+1)	n≥2
3/4	90942/54693	43.66587*n(n+1)	n≥2

VI. Machines with Complex Setup and Execution Units

Thus far, it has been assumed that the setup section of a machine consists of N_S pipeline segments each of which has unit service time, and that in the absence of taken branches, setup does not influence the performance of the machine. In many machines (particularly machines having complex architectures), the segments of setup are difficult to identify. In addition, the nature of the interactions between the components of setup and the memory system can be complex. However, setup was assumed to be simple in nature so that the branch penalty could be easily assessed. Thus, the only real consequence that complex setup sections have is that the branch penalty for such machines must be estimated. Fortunately, a fairly good approximation can be made rather easily.

If any suitable analytic model, as in [22]–[28], is applied just to the setup section of a machine, assuming no branching or data dependencies, then a raw bandwidth figure can be obtained, BW_S. The only information that is required to obtain BW_S is the topology of the machine along with the cycle times and effective access times of its associated components. If the memory system is hierarchical, miss ratios must also be assumed. Thus, the first estimate of setup bandwidth might require no workload information other than a rough estimate of cache miss ratio. Note that the inverse bandwidth in this case is calculated for an unflushed system, and it represents the average latency (i.e., the time between two consecutive instructions) in setup if there were no setup flushes due to taken branches.

Whenever there is a taken branch, there can be (at worst) a complete flush of setup. Although the estimate of BW_S gives the average instruction frequency in an unflushed pipeline, the full flush penalty requires knowledge of the average latency in a flushed pipeline (i.e., the average time at the end of setup between a taken branch and its target instruction). Since this flushed latency may be significantly more than the average inverse bandwidth during steady-state operation, an estimate of the average flushed latency L_F is required.

L_F is equal to the time for a branch target instruction to traverse the setup section once the prior branch instruction is resolved at the end of setup. For example, in a machine with no branch target prefetch and no overlap between memory accesses, let T_f be the average number of cycles to fetch the target instruction, T_D be the number of cycles to decode the instruction, T_m be the average access time in cycles to fetch a

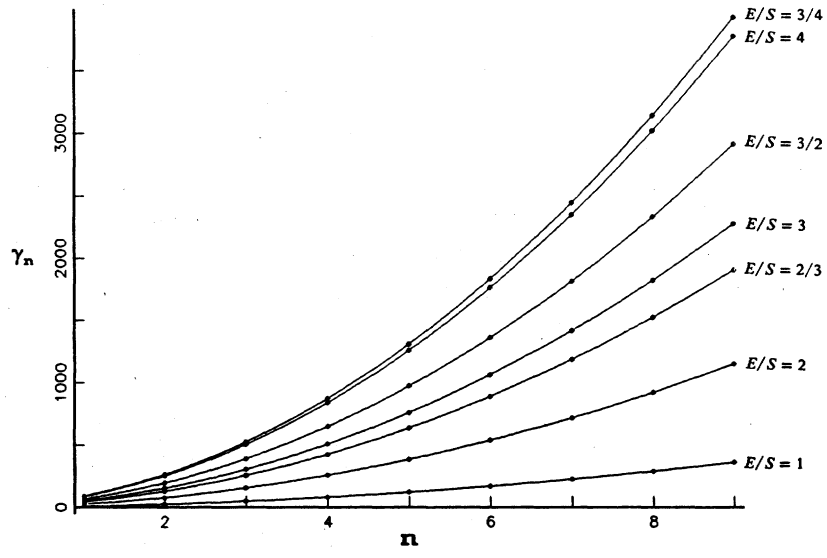

Fig. 12. The sequences γ_n for various E/S.

memory operand, T_R be the number of cycles to read a register, p_r be the average number of register operands per instruction, and p_m be the average number of memory operands per instruction. Then a good estimator for L_F is

$$L_F = T_f + T_D + p_r T_R + p_m T_m.$$

T_f and T_m would include estimates of hit ratios, access conflict cycles, communication delays, access times, and block-transfer effects for various levels of the memory hierarchy.

T_f would be 0 if branch target prefetching always occurs prior to the actual setup section (i.e., target prefetch is always completed before branch resolution). The $p_r T_R$ and $p_m T_m$ terms may be reduced if operand-access overlap features are included in the machine. A more accurate probabilistic model for estimating L_F would include an actual probability distribution for the various terms, rather than simply the averages (distribution means) of those distributions [27].

This estimate of the instruction-setup latency L_F can be used as the full flush penalty. In the case of short forward branches, a full flush may not be required. Note that, in the absence of taken branches, the average number of instructions in setup at any time is $N_S = L_F BW_S$. N_S thus represents the effective number of pipeline segments in setup. Thus, if the distance (in instructions) d from a forward branch to its target is less than N_S, then the flush penalty is only $(d - 1)/BW_S$. N_E can be evaluated similarly from the effective concurrency in the execution pipeline in the absence of data dependencies.

In machines having multiple execution units of different classes, there can be instruction passing after setup. For example, consider a machine that has a three-segment pipelined adder, and a seven-segment pipelined multiplier. An add instruction can begin as much as three cycles after a multiply instruction begins, and still complete its execution before the multiply has completed. This effective passing is not reflected in the way that the trace reductions are performed on an execution trace, i.e., the add is not allowed to resolve any dependencies until at least one cycle after the multiply has completed.

The effect can be effectively handled by the trace reductions by modifying the definition of temporal dependency distance. Let each node i in the dependency graph be tagged with the execution time in cycles e_i associated with its instruction type. Then the generalization of effective dependency-distance must include the amount of time required by the resolving instruction to complete execution. Recall that the temporal distance associated with a dependency arc is the number of cycles between starting the resolving instruction in execution and starting the dependent instruction in execution. Assuming a steady-state instruction flow, the temporal distance associated with the scalar distance D_i can be estimated, in the absence of other overlapping dependencies, as

$$\delta_i = D_i \cdot BW_S^{-1} + (e_{i_0} - D_i \cdot BW_S^{-1})^+$$

where i_0 is the resolving instruction. If the same reductions are applied to any trace using this more general form of temporal distance, the resulting set of first-order statistics will characterize the trace for the corresponding machine architecture. When this assumption does not yield a good approximation, BW_S^{-1} can be replaced with an entire distribution rather than the mean. This will generate a higher instruction frequency due to the nonlinearity of the $(x)^+$ term in the δ_i formula [27].

VII. Conclusion

The nature of the interaction between data dependencies and branches in a program can be quite complex. If one tries to model the performance of a machine by examining an execution trace without performing reductions, no set of simple statistics is sufficient. Thus, the alternatives to date have been simulation, and simple models using first-order approximations.

In this paper, a set of reductions was presented that reduces a dependency graph to an equivalent simplified graph for which characterization by first-order statistics is sufficient and straightforward. These trace reductions were derived by assuming that a machine comprises a single pipeline having a setup section followed by an execution section. Several

estimation techniques were also presented for use with more complex machines. These techniques hinge on broadening the definition of "dependency distance." Except for the setup–execute assumption, the trace reductions are machine independent, and thus, any workload can be accurately characterized by a set of first-order statistics.

It was shown that data dependencies and branches cause a monotonic increase in cycles per instruction as the pipeline length is increased. However, lengthening the pipeline corresponds to reducing its cycle time if the architecture is fixed. These two opposing effects were used in conjunction to derive a general equation for the pipeline length that achieves maximum performance for executing a particular set of traces, given one technology parameter, namely the relative overhead for latching in the pipeline.

References

[1] L. A. Belady, "A study of replacement algorithms for a virtual storage computer," *IBM Syst. J.*, vol. 5, pp. 78–101, 1966.

[2] O. Babaoglu and D. Ferrari, "Two-level replacement decisions in paging stores," *IEEE Trans. Comput.*, vol. C-32, pp. 1151–1159, 1983.

[3] A. J. Smith, "Cache memories," *Comput. Surv.*, vol. 14, pp. 473–530, 1982.

[4] ——, "Sequentially and prefetching in database systems," *ACM Trans. Database Syst.*, vol. 3, pp. 223–247, 1978.

[5] ——, "Disk cache-miss ratio analysis and design considerations," *ACM Trans. Comput. Syst.*, vol. 3, pp. 161–203, Aug. 1985.

[6] ——, "Long term file migration: Development and evaluation of algorithms," *Commun. ACM*, vol. 24, pp. 521–532, 1981.

[7] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annu. Symp. Comput. Architecture*, vol. 9, 1981, pp. 135–148.

[8] J. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, pp. 6–22, Jan. 1984.

[9] D. E. Lang, T. K. Agerwala, and K. M. Chandy, "A modeling approach and design tool for pipelined central processors," in *Proc. 6th Annu. Symp. Comput. Architecture*, vol. 7, Apr. 1979, pp. 122–129.

[10] B. Kumar and E. S. Davidson, "Performance evaluation of highly concurrent computers by deterministic simulation," *Commun. ACM*, vol. 21, pp. 904–913, Nov. 1978.

[11] ——, "Computer system design using a hierarchical approach to performance evaluation," *Commun. ACM*, vol. 23, pp. 511–521, Sept. 1980.

[12] B. L. Peuto and L. J. Shustek, "An instruction timing model of CPU performance," in *Proc. 4th Annu. Symp. Computer Architecture*, 1977, pp. 165–178.

[13] D. W. Clark and H. M. Levy, "Measurement and analysis of instruction use in the VAX 11/780," in *Proc. 9th Int. Symp. Computer Architecture*, Apr. 1982, pp. 9–17.

[14] R. P. Blake, "Exploring a stack architecture," *Computer*, vol. 10, pp. 30–41, May 1977.

[15] M. Kobayashi, "Dynamic profile of instruction sequences for the IBM, System/370," *IEEE Trans. Comput.*, vol. C-32, pp. 859–861, Sept. 1983.

[16] ——, "Dynamic characteristics of loops," *IEEE Trans. Comput.*, vol. C-33, pp. 125–132, Feb. 1984.

[17] M. H. MacDougall, "Program behavior and processor design," in *The Measurement of Computer Software Performance*, Los Almos National Lab., 1983.

[18] R. W. Doran, "The Amdahl 470 V/8 and the IBM 3033: A comparison of processor designs," *Computer*, vol. 15, pp. 27–36, Apr. 1982.

[19] S. R. Kunkel and J. E. Smith, "Optimal pipelining in supercomputers," in *Proc. 13th Annu. Symp. Comput. Architecture*, 1986, pp. 404–411.

[20] A. G. Larson and E. S. Davidson, "Cost-effective design of special-purpose processors: A fast Fourier transform case study," in *Proc. 11th Annu. Allerton Conf. Circuit Syst. Theory*, Oct. 1973, pp. 547–557.

[21] S. H. Unger and C. J. Tan, "Clocking schemes for high-speed digital systems," *IEEE Trans. Comput.*, vol. C-35, pp. 880–895, Oct. 1986.

[22] T. C. Chen, "Parallelism, pipelining, and computer efficiency," *Comput. Design*, pp. 69–74, Jan. 1971.

[23] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: Hemisphere, 1981.

[24] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[25] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948–960, Sept. 1972.

[26] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *Comput. Surv.*, vol. 9, pp. 59–102, Mar. 1977.

[27] P. G. Emma and E. S. Davidson, "A residual-time model for pipeline performance," submitted for publication.

[28] ——, "A class of performance estimators for pipelines with embedded buffers," submitted for publication.

Philip G. Emma received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana.

He joined the IBM Research Division, Yorktown Heights, NY where he works on high-end processor organization.

Edward S. Davidson was born in Boston, MA in 1939. He received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1961, the M.S. degree in communication science from the University of Michigan, Ann Arbor, in 1962, and the Ph.D. degree in electrical engineering from the University of Illinois in 1968.

He worked in logic design of the H-200 family at Honeywell from 1962 to 1965 and was an Assistant Professor of Electrical Engineering at Stanford University from 1968 to 1973. He then returned to the University of Illinois where he now holds a joint appointment as Professor of Electrical and Computer Engineering, the Coordinated Science Laboratory, and Computer Science. He is involved in the Cedar parallel supercomputer implementation as Associate Director for Hardware at the Center for Supercomputer Research and Development. He has performed research in computer architecture, parallel and pipeline processing, performance modeling, VLSI systems, computer-aided design, and fault tolerance. He has supervised 33 Ph.D. and 36 M.S. degree students and authored 50 technical publications. He has served as Consultant to Digital Equipment Corporation, General Electric, Hewlett-Packard, Honeywell, Fort Monmouth, Sperry, Defense Nuclear Agency, and others.

Dr. Davidson is a former Chairman of ACM-SIGARCH.