SAND93-4088
Unlimited Release
Printed December 1993

Distribution
Category UC-905

Varying Execution Discipline to Increase Performance

Philip L. Campbell
Dissertation Proposal
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico 87131
Advisor: Arthur B. Maccabe
Date of Defense: December 15, 1993
December 22, 1993

"Supercomputers define major frontiers in science and engineering.... Supercomputers also define a major frontier of computing technology. The familiar constraints of cost, physical size, and power consumption are put aside in favor of performance.... Future gains in speed will require new architectures, such as systolic arrays, hypercubes, data-flow machines, and other large networks of processing elements."--Peter J. Denning and George B. Adams III, "Research Questions for Performance Analysis of Supercomputers." ([11], p.405)

Abstract

This research investigates the relationship between execution discipline and performance. The hypothesis has two parts:

1. Different execution disciplines exhibit different performance for different computations, and
2. These differences can be effectively predicted by heuristics.

A machine model is developed that can vary its execution discipline. That is, the model can execute a given program using either the control-driven, data-driven or demand-driven execution discipline. This model is referred to as a "variable-execution-discipline" machine. The instruction set for the model is the Program Dependence Web (PDW). The first part of the hypothesis will be tested by simulating the execution of the machine model on a suite of computations, based on the Livermore Fortran Kernel (LFK) Test (a.k.a. the Livermore Loops), using all three execution disciplines.

Heuristics are developed to predict relative performance. These heuristics predict (a) the execution time under each discipline for one iteration of each loop and (b) the number of iterations taken by that loop; then the heuristics use those predictions to develop a prediction for the execution of the entire loop. Similar calculations are performed for branch statements. The second part of the hypothesis will be tested by comparing the results of the simulated execution with the predictions produced by the heuristics.

If the hypothesis is supported, then the door is open for the development of machines that can vary execution discipline to increase performance.

Note

This Report has also been published as a UNM Technical Report with the same title, No. CS93-12, University of New Mexico, Albuquerque, New Mexico, 87131.

Contents

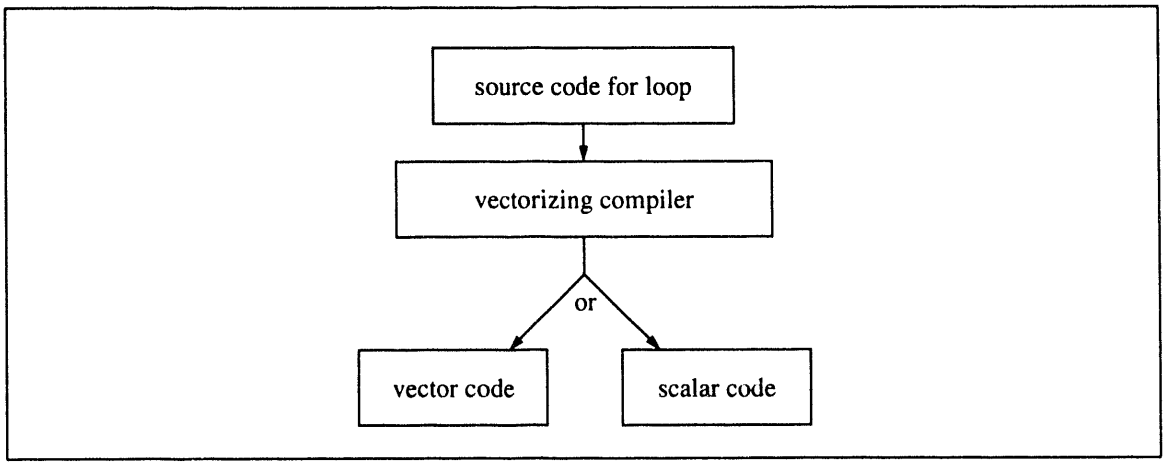
Figures

Tables

1.0 Introduction

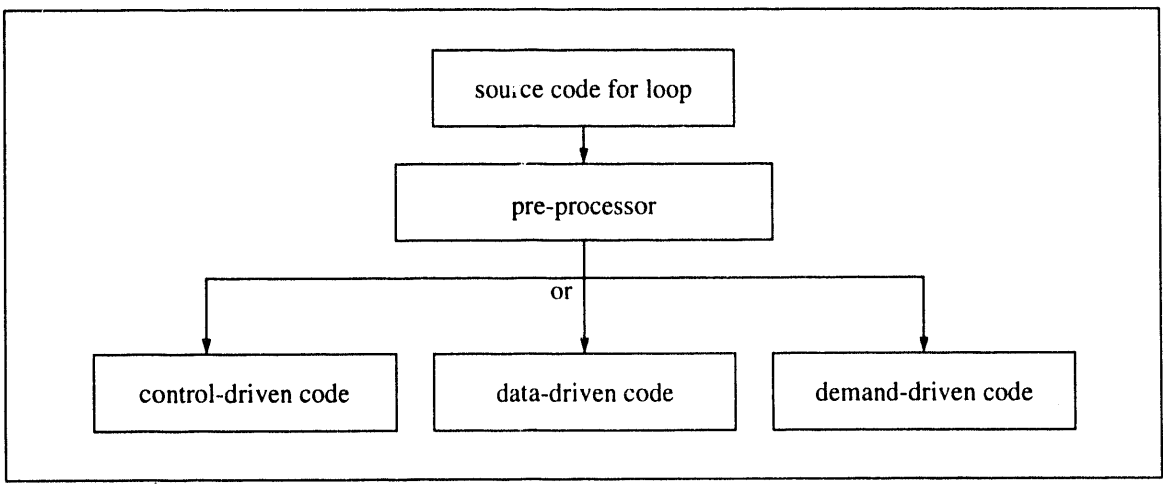
Vector computers such as the Crays have two sets of functional units – scalar and vector. For some loops the vector units execute faster than the scalar units. For other loops the scalar units execute faster than the vector units. Thus, varying the choice of hardware unit can increase performance. Determining the appropriate hardware unit is possible for many loops without actually executing the loop on both units and comparing execution times. It is the job of the compiler to make this determination, as shown in Figure 1.

Figure 1. A Vectorizing Compiler



Just as there are two sets of hardware units available to execute loops on vector computers, there are several possible ways to execute programs. These several ways are known as "execution disciplines." Most computers execute programs using the "control-driven" discipline that schedules operations using a Program Counter and follows the sequence specified explicitly by the programmer. Cray machines, for example, use the control-driven discipline for both the vector and scalar units. However, a few computers execute programs using the "data-driven" discipline. Still others use the "demand-driven" discipline. The hypothesis of this research is that varying the execution discipline can increase performance. The long-range goal of this research is the development of a pre-processor that would be similar to a vectorizing compiler in that it would partition a code to increase performance, but dissimilar in that the partitions would be targeted for different execution disciplines, as shown in Figure 2.

Figure 2. A Pre-Processor that Determines Execution-Discipline



The hypothesis of this research has two parts:

1. Different execution disciplines exhibit different performance for different computations, and
2. These differences can be effectively predicted by heuristics.

In order to test the first part of the above hypothesis, a way to execute a computation under each execution discipline is needed. Unfortunately there exists no one machine designed to provide each execution discipline. This leaves us with two choices: select three different machines, each designed with a different execution discipline in mind; or simulate a single machine that can provide each execution discipline. From the viewpoint of this research the fundamental problem with the first approach is that it compares implementations of the execution disciplines, not the disciplines themselves. This research will pursue the second approach. Accordingly, a machine model is developed that separates operation scheduling from operation execution and thus can accommodate each discipline. The Program Dependence Web (PDW) ([4], [9]) serves as the instruction set for that machine model. The PDW is an intermediate program representation. The unique quality of the PDW is that it can be interpreted via each of the disciplines. Simulated execution of the machine model on the Livermore Fortran Kernel (LFK) Test will provide the relative performance for the different disciplines for those different computations. This will test the first part of the hypothesis.

In order to test the second part of the hypothesis, heuristics are needed. Predictions are first developed for the execution time of one iteration of a loop. Then the number of iterations is predicted. The product of these two predictions, with a weighting factor for nested loops, is the predicted execution time of the loop. Similar calculations are performed for branch statements. The predictions derived by these heuristics will be compared with the relative performance from the simulated execution. This will test the second part of the hypothesis.

If the hypothesis is supported, then the model that should be used for the execution disciplines is of different tools, each with its own strength and area of application. Such a result opens the door to machines that can vary execution discipline based on the program to be executed, or even use different execution disciplines for different parts of the same program.

If the hypothesis is not supported, then the value of alternative execution disciplines, namely data- and demand-driven execution, is open to question.

The remainder of this proposal is organized as follows. Section 2.0 presents the three execution disciplines. Section 3.0 presents a machine model that will accommodate control-, data- and demand-driven execution. This is followed in Section 4.0 by a description and an example of the PDW that serves as the instruction set for the machine model presented in Section 3.0. Section 5.0 presents a summary of related research. Section 6.0 presents the research plan: the simulator and the heuristics depend on three ancillary pieces – a source-to-PDW translator, a suite of computations and an operation table that maps different operations on to execution time. This collection will enable testing both parts of the hypothesis. Section 7.0 presents conclusions and implications of this research. An Appendix presents the calculation of the heuristics on the first kernel of the LFK Test.

2.0 Execution Disciplines

This section describes the three execution disciplines. Table 1 presents a summary.

Table 1. Summary of Execution Disciplines

Question	Control-driven	Data-driven	Demand-driven
What characterizes an executable operation?	The operation's address is in the Program Counter at the beginning of the instruction cycle.	The operation has received enough of its inputs to be able to execute.	The operation has received enough of its inputs to be able execute *and* its output has been "demanded" (i.e., a request has been placed for the operation's output).
Is the discipline by nature sequential or parallel?	Sequential	Parallel	Parallel
What form of dependence is used to determine the relative ordering of operations?	Control Dependence	Data Dependence	Data Dependence

Control-driven execution can accommodate a limited amount of parallelism, such as instruction pre-fetching and instruction pipelining. Compilers can detect and exploit some parallelism as well. However, control-driven execution is constrained in the amount of parallelism it can accommodate. Data-driven execution, by comparison, reveals all of the (operation-level) parallelism.

All three execution disciplines rely on the notion of "dependence." There are two types of dependence: control and data. If X and Y are operations, then stating that Y is "control dependent" on X means that whether or not Y executes depends on the results of the execution of X. X must execute before Y, if Y is to execute at all.

Stating that Y is data dependent on X means that X and Y define and/or use the same variable. There are three types of data dependence:

flow (also known as "true"), anti- and output.[1]

If Y is flow dependent on X, then X defines (or sets a value to) a variable that Y uses. If Y is anti-dependent on X, then X uses a variable that Y defines. And if Y is output dependent on X, then both nodes define the same variable.

2.1 Control-Driven Execution

Under control-driven execution an operation is executable when its address is in the Program Counter (PC) at the beginning of the instruction cycle. Since execution of an operation can change the contents of the PC, scheduling the next operation for execution must wait until the current operation has completed execution. Various techniques, such as branch prediction and pipeline delay slots [20], allow some overlap in the scheduling of operations but they cannot

1. Flow, anti- and output dependences are also known as "def-use," "use-def," and "def-def" relationships respectively (where "def" is short for "definition"). There is a fourth possibility – use-use – but this is of little interest.

overcome the sequential nature of control-driven execution. Control-driven execution uses control-dependence to determine the relative ordering of operations.

Consider the program fragment shown below in Figure 3. The fragment, expressed in C-like syntax, computes the factorial function: `f(n)=f(n-1)*n, for n>0; f(1)=1.`

Figure 3. Factorial Fragment

```
/* assume input value for n is always > 0 */
1:   read  (n);
2:   write (n);
3:   fact = n;
4:   while (n > 1) {
5:        n = n - 1;
6:        fact = fact * n;
     }
7:   write (fact);
```

Control-driven execution of the code in Figure 3 begins with statement 1 that reads in the value for variable `n`. Assume that the value read in for `n` is 2. After statement 1 has completed, execution proceeds with statement 2 that writes out the value just read in. Execution continues with statement 3. The predicate for the `while` loop at statement 4 then executes. The predicate evaluates to *true*, since `n=2` and `2>1`, so execution proceeds with statements 5 and then 6. The predicate evaluates again, this time to *false*, so execution of the fragment terminates on the eighth step with statement 7. This sequence is shown in Table 2. Note that this is not the only sequence that would produce the same output. For example, statements 2 and 3 could execute in reverse order and still produce the same output.

Table 2. Control-Driven Execution of Figure 3

Step	Statements Executed
1	1: read (n); --n=2
2	2: write (n); --writes 2
3	3: fact = n; --fact=2
4	4: while (n > 1) --evaluates to *true*
5	5: n = n - 1; --n=1
6	6: fact = fact * n; --fact=2*1
7	4: while (n > 1) --evaluates to *false*
8	7: write (fact); --writes 2

2.2 Data-Driven Execution

Under data-driven execution an operation is executable when it has received enough of its inputs to execute. Which operation is chosen for execution – along with how many – is implementation dependent. Execution terminates when no operation can execute. Scheduling is implemented via a "Matching Unit" instead of a Program Counter. The Matching Unit notes the arrival of each input for each operation and schedules an operation for execution when it has

received enough of its inputs to do so. Data-driven execution can be thought of as "eager" evaluation: whenever an operation is able to execute, it may do so. Data-driven execution is by nature parallel. It reveals all possible parallelism. It uses data-dependence to determine the relative ordering of operations.

Revisiting Figure 3 above and assuming the same input for control-driven execution, data-driven execution begins with statement 1 since this is the only statement that has received enough of its inputs initially to execute. However, as soon as statement 1 completes, the value of `n` is available and thus statements 2, 3, 4 and 5 can all execute concurrently – the execution of these statements constitutes the second step. Assume that each of these statements receives a *copy* of the value of `n`, not the single address of `n`, so there is no race condition in their simultaneous execution. Statement 6 must wait until statement 5 has executed because it needs the decremented value of `n`. And the second execution of statement 4 must also wait for the completion of the execution of statement 5 for it needs that same value. So statements 6 and 4 must wait until step 3 to execute. In step 3 statement 4 evaluates to *false*. This terminates the loop and allows statement 7 to execute in step 4 – statement 7 is waiting on the final value of `fact`. Execution halts after step 4 completes because no operation can then execute.[2] This execution sequence is summarized in Table 3 below. Since data-driven execution is by nature parallel the imposition of "steps" is for explanatory purposes only.

Table 3. Data-Driven Execution of Figure 3

Step	Statements Executed
1	1: read (n); --n=2
2	2: write (n); --writes 2 3: fact = n; --fact=2 4: while (n > 1) --evaluates to *true* 5: n = n - 1; --n=1
3	6: fact = fact * n; --fact=2*1 4: while (n > 1) --evaluates to *false*
4	7: write (fact); --writes 2

2.3 Demand-Driven Execution

Under demand-driven execution an operation is executable when it has received enough of its inputs to execute *and* its output has been demanded (i.e., a request has been placed for the operation's output). Which operation is chosen – along with how many – is implementation dependent. Like data-driven execution, scheduling under demand-driven execution is implemented via a Matching Unit, except that the Matching Unit must also keep track of the propagation and fulfillment of demands. Also like data-driven execution, demand-driven execution is by nature parallel and it uses data-dependence to determine the relative ordering of operations. However, demand-driven execution is not eager: no operation executes until it is known that its execution contributes to the fulfillment of the original demands.

Demand-driven execution involves two phases – demand propagation and evaluation. If an operation has been demanded, then the operation first propagates that demand to the operations that generate its inputs. This is the demand propagation phase. When those demands have been fulfilled by the arrival of data, then the operation can

2. This is not completely true, given the discussion above. Since statement 5 provides its own input after receiving an initial value, its execution could continue indefinitely and the accumulation of resulting intermediate values would lead to deadlock [6]. However, execution on a real machine would throttle and terminate execution of this statement, perhaps through "static interpretation" or "loop unraveling" [7]. Note that in the PDW shown in Figure 9 below that the addition operator for statement 5 is tethered to the execution of the loop.

evaluate. This is the second phase of execution. The stage is set for execution by placing the demands from the world outside the program on the operations whose execution will fulfill those demands. When execution begins, these operations can enter their demand propagation phase. These original demands propagate, fanning out and forming a frontier that continues until each point on the frontier is either a constant or a read of data. Operations on the frontier can then begin their evaluation phase. At any moment in execution the operations can be partitioned as follows:

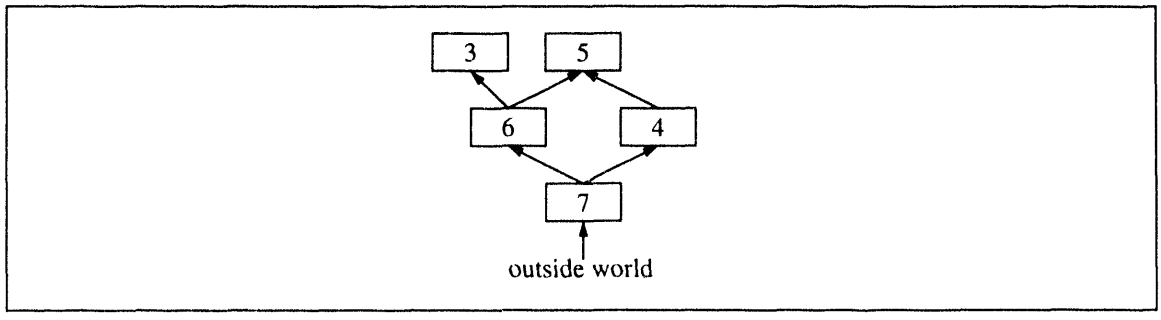
operations that are propagating a demand,
operations that are waiting for the fulfillment of the demands they have propagated,
operations that are evaluating, or that have completed evaluation and are fulfilling demands, and
operations that are not demanded.

The set of demanded operations can be described by a directed graph called a "demand graph:"

nodes with zero in-degree represent the operations demanded from the world outside the program;
the leaf nodes (i.e., nodes with zero out-degree) represent operations that will either propagate demands or can evaluate; and
the remaining nodes in the graph are operations that are waiting for the fulfillment of the demands they have propagated.

Any leaf node that has had all of its demands fulfilled is a candidate for execution. Note that the demand graph confines execution to just those operations that contribute to the fulfillment of the original demands. Figure 4, below, is an example of a demand graph. The "outside world" is demanding node 7, and node 7 is demanding nodes 6 and 4. Node 6 is demanding nodes 3 and 5; and node 4 is also demanding node 5. (Figure 4 is the demand graph at the beginning of step 3 in the execution traced below.)

Figure 4. A Demand Graph



Revisiting Figure 3 above and assuming the same input and that statement 7 alone is demanded, execution begins with statement 7 and proceeds in the steps shown below. Since demand-driven execution is by nature parallel the imposition of "steps" in the discussion below is for explanatory purposes only.

1. Statement 7 demands statement 2 in order to receive the "file handle" information necessary to generate output (assume that statements 2 and 7 write to the same file). Statement 2 demands statement 1 in order to receive both the file handle information as well as the value of n. Statement 7 also demands statement 4 that demands statement 1 that demands input from the world outside the program. Statement 7 demands statement 4 because statement 4 is the one that determines whether the value for `fact` in statement 7 is set in statement 3 or statement 6. Depending upon the input that statement 4 provides when it evaluates, statement 7 will either demand statement 3 or 6, as shown below. When the input that statement 1 has demanded has been received, then statement 1's demand on the outside world has been fulfilled and statement 1 can evaluate and can fulfill the demand placed on it by statements 2 and 4.

2. Statement 2 evaluates – now that its demand on statement 1 has been fulfilled – and it supplies statement 7 with the file handle information. And statement 4 also evaluates – now that its demand on statement 1 has been fulfilled – and it supplies statement 7 with the value *true* for the predicate, assuming once again that the input was the value 2.

3. Statement 7, based on the *true* input received from statement 4, proceeds to issue more demands. It demands the evaluation of the body of the loop by demanding statement 6. Statement 6 needs a value for `fact` as well as a value for `n`, so it demands statements 5 and 3. Both of these statements can evaluate without issuing a demand – the `read` in statement 1 provided these statements with input when it evaluated in step 1. Statement 7 also demands another evaluation of the predicate in order to determine if an additional evaluation of the body of the loop is needed. Statement 7 does this by demanding statement 4 for the second time. Statement 4 then, in turn, demands statement 5.

   When statements 3 and 5 evaluate, they fulfill statement 6's demand on them and, at the same time, statement 4's demand on 5. Note that statement 5 evaluates only once in this step even though it was demanded twice. A copy of the output of an evaluating statement is queued for each of the statements that is to receive its output, regardless of whether or not the receiving statements have posted a demand. A single evaluation of statement 5 thus fulfills all demands pending on it. This arrangement is possible only if it is assumed that the operations follow "single-assignment semantics" – imagine the output of an operation being copied and flowing to the operations that use it, as opposed to the output of an operation being stored in a single location in memory. The difference is that with the first view a subsequent evaluation of the operation provides a subsequent output value, whereas with the second view a subsequent evaluation overwrites the previous output.

4. Statements 6 and 4 evaluate. Statement 6 passes the value 2 to statement 7, and statement 4 passes a *false* to statement 7, fulfilling statement 7's second set of demands on statements 6 and 4.

5. Statement 7 evaluates, fulfilling the original demand and terminating execution of the fragment.

This execution sequence is summarized in Table 4 below.

Table 4. Demand-Driven Execution of Figure 3

Step	Demands Propagated	Statements Evaluated	Demands Fulfilled	Demands Outstanding
1	outside world demands 7 7 demands 2, 2 demands 1, 7 demands 4, 4 demands 1	1: read (n);	2's demand on 1, 4's demand on 1	outside world waiting on 7, 7 waiting on 2, 7 waiting on 4
2	Ø	2: write (n); 4: while (n>1)	7's demand on 2, 7's demand on 4	outside world waiting on 7
3	7 demands 6, 6 demands 5 and 3, 7 demands 4, 4 demands 5	3: fact = n; 5: n = n - 1;	6's demand on 5 and 3 4's demand on 5	outside world waiting on 7, 7 waiting on 6 and 4
4	Ø	6: fact=fact*n; 4: while (n>1)	7's demand on 6 and 4	outside world waiting on 7

Table 4. Demand-Driven Execution of Figure 3

Step	Demands Propagated	Statements Evaluated	Demands Fulfilled	Demands Outstanding
5	Ø	7: write (fact);	outside world's demand on 7	Ø

Suppose that instead of statement 7, only statement 2 were demanded. In this case, statement 2 would demand statement 1, and statement 1 would demand input from the world outside, and, when this completed, statement 2 would evaluate, fulfilling the original demand. None of the other statements in the fragment would evaluate. This points out how demand-driven execution can decrease the number of execution steps by evaluating only the necessary operations.

2.4 Relationships Between the Disciplines

This section presents (a) a notation that shows the relationship between the three disciplines, and (b) a brief comparison of the efficiency of each discipline.

Adopting the following denotations,

P denotes a program-counter phase
- in which the next operation is identified via the Program Counter,

S denotes a synchronization phase
- in which data synchronizations are performed and, as a result, executable operations are identified,

D denotes a demand propagation phase
- in which demands are propagated, and

E denotes an execution
- in which at least one operation is executed.

then

control-driven execution consists of a sequence {PE, PE,..., PE};
data-driven execution consists of a sequence {SE, SE,..., SE}; and
demand-driven execution consists of a sequence {DSE, DSE,..., DSE}.

Control-driven execution is the classic, sequential execution discipline. Control-driven execution requires at least as many execution steps as the other two disciplines because it cannot accommodate parallel execution, at least not without a parallelizing compiler or special annotations from the user. It is difficult to imagine a scheduling technique that requires less overhead than the use of a Program Counter. However, machine code that is designed for control-driven execution is surprisingly inefficient: operations that simply copy data from one place to another usually dominate computations ([24], pp. 416-8).

Data-driven execution imposes synchronization overhead on the transfer of each intermediate data value ([22], particularly section 5.1.4). Scheduling requires the functionality of associative memory. This is more expensive than updating a Program Counter. However, data-driven execution yields maximum parallel execution, thus possibly decreasing the number of execution steps. Whether or not data-driven execution completes in fewer steps than control-driven execution depends upon the available parallelism in the computation.

Demand-driven execution requires even more overhead than data-driven execution due to its use of demands. But demand-driven execution can possibly decrease the number of execution steps below even what data-driven execution requires by reducing the number of operations executed.

3.0 Variable-Execution-Discipline Machine Model

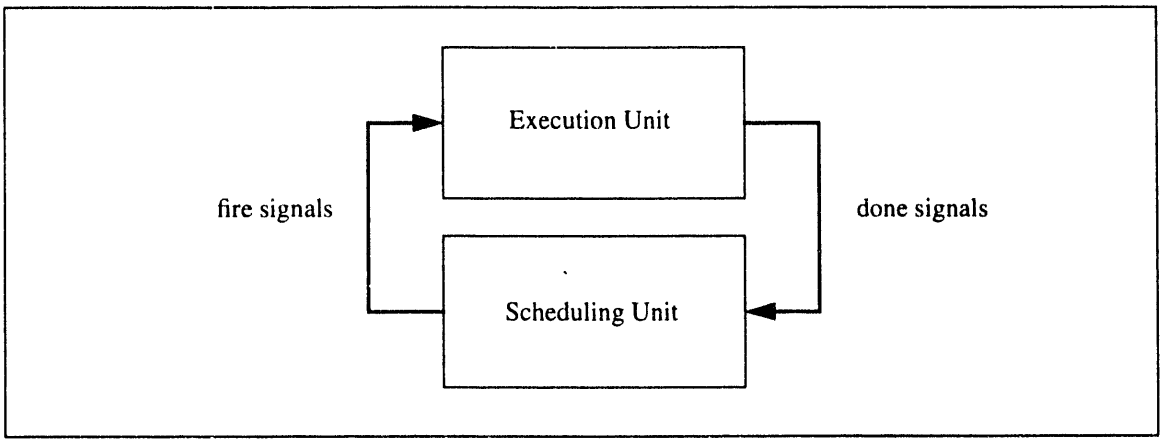
This section presents a machine model that can execute a program under each of the three disciplines. This machine provides an environment in which the performance of these disciplines can be compared, which is the purpose of this research.

Without exception there is only one available execution discipline for any given machine. This includes the "hybrid" machines reviewed in Section 5.3.4 on page 22 below. No current machine can provide the particular needs of this research, hence the necessity of developing a new model.

An execution discipline is a method of scheduling operations. The new model therefore needs to isolate all activities attributable to operation scheduling. The actual execution of the operations specified by the input program is independent of the scheduling of those operations and thus the execution should exhibit the same performance regardless of the discipline used. In the simulation of this new model, discussed in Section 6.2 on page 29 below, various activities of the machine are parameterized via an Operation Table. Other aspects of this model are not parameterized, differences such as instruction pipelining and vectorizing capabilities, compiler optimizations, differences due to machine instruction sets and architectural features tuned to a particular execution discipline.

A block diagram of the new model is shown in Figure 5 below. The diagram is a generalization of Gao's "argument-fetch" architecture [14].[3] Since the execution discipline used by this machine can vary, the model is named the "variable-execution-discipline" machine model.

Figure 5. Variable-Execution-Discipline Machine



The "Scheduling Unit" determines the operations that can execute and so informs the "Execution Unit." The Execution Unit executes the operations and informs the Scheduling Unit.

Operands and operators all remain within the Execution Unit. Like Gao's machine the Scheduling Unit of the variable-execution-discipline model contains a graphical form of the program to be executed and the Execution Unit contains the actual operations corresponding to that program, along with the input data and the intermediate results. However, in the variable-execution-discipline machine model the graphical form is a PDW, presented in the next section. The operations in the Execution Unit are the nodes of the PDW. It is assumed that both Units have sufficient

3. Evripidou and Gaudiot's "Decoupled Graph/Computation" model [12] has the same general architecture as this one: a "Data-Flow Graph Engine" – this is Gao's "Scheduling Unit" – communicates with a "Computation Engine" – this is Gao's "Execution Unit" – via a "Ready Queue" and an "Acknowledgement Queue" that hold <actor ID, context> pairs.

buffer space so that they have the capability of running independently of each other in time (i.e., asynchronously). It is also assumed that the Scheduling Unit and the Execution Unit have a sufficient number of processors available for simultaneous execution; however, the number of processors is set prior to execution.

The variable-execution-discipline machine model does not specify a scheduling discipline for the Scheduling Unit or an implementation for the Execution Unit. Gao, on the other hand, specifies both. The Scheduling Unit in Gao's machine is constrained to schedule operations via the data-driven discipline. And the Execution Unit is a RISC processor that achieves its parallelism via instruction pipelining. Like the HEP machine [39], operations from different parts of the program can simultaneously share the instruction pipeline.

The value of the variable-execution-discipline model is that it separates the scheduling discipline from the execution of a program. This allows experimentation with the scheduling discipline – exactly what is needed to test the hypothesis of this research. For example, if the Scheduling Unit operates schedules via a Program Counter, then, as explained above, control-driven execution is used. If the Scheduling Unit schedules via a dataflow graph, then data-driven execution is used. And if the Scheduling Unit schedules via a demand graph, then demand-driven execution is used.

This model is in effect a single "vanilla" machine that has no glaring architectural bias and executes in any of the three disciplines. Because the computation always executes on the same machine, variability due to anything but scheduling discipline is held constant. These constraints force control-driven execution to be serial – unless the user provides annotations via the "parallel" statement (see Section 6.1.1 on page 25) – since there is no compiler included that can produce parallel or vectorizing code. The constraints also force the data synchronization overhead of data-driven execution to be explicitly represented in performance, since there is in this vanilla architecture no matching unit per se or circular pipeline characteristic of data-flow machines. And they force the demand propagation overhead of demand-driven execution to be likewise explicitly represented in performance since there is no hardware assist for this activity.

Although beyond the scope of this research, this machine model also opens the door for exploration of a control/data/demand hybrid.

4.0 The Program Dependence Web (PDW)

Machine instructions for typical control-driven machines are different than machine instructions for typical data-driven machines. Demand-driven machines differ from both control- and data-driven machines. However, performance comparison requires a single set of machine instructions that can be interpreted under control-, data- or demand-driven disciplines. Returning to the machine model presented above, what is needed is an instruction set for that model. This is the nature of the PDW.

The PDW was developed from the Program Dependence Graph (PDG) ([13], [21], [36], [38], [5]). The PDG is based on the control dependence graph (CDG) and the data dependence graph (DDG). Control and data dependence were explained at the beginning of Section 2.0 above. As an example, the statements inside the `while` loop in Figure 3 (statements 5 and 6) are control dependent on the predicate of the loop (statement 4). And statement 2 is flow data dependent on statement 1: statement 1 defines the value of variable n; statement 2 uses that value.

The PDG represents the flow of both control and data. The PDG, however, is not interpretable. The PDW is different than the PDG in that it can not only be interpreted but that it can be interpreted under control-, data- and demand-driven disciplines. The only structure that is even similar to the PDW is the "demand graph"[4] used in the RC compiler for the DTN Dataflow Computer [41]. However, the demand graph is designed for data-driven execution only. (Also, the demand graph has no "control dependence" arcs per se or provision for unconstrained gotos.)

More formally, the PDW = (V,A) is a directed multigraph where the vertices represent operations and the arcs represent either control or data dependences. The set of vertices is

$$V = \{\text{operators, predicates, read, write, Switch}^T, \text{Switch}^F, \eta^F, \gamma, \beta, \text{region, Entry, Exit}\},$$

and the set of arcs is

$$A = \{\text{control dependence}^{\text{true}}, \text{control dependence}^{\text{false}}, \text{data}^{\text{flow}}, \text{data}^{\text{anti}}, \text{data}^{\text{output}}\}.$$

The vertices of the PDW fall into two groups that will be given *ad hoc* names of "data-manipulation" nodes and "execution-control" nodes. The first group, data-manipulation nodes, contains operator nodes (such as add, subtract, etc.), predicate nodes and read/write nodes. This group transforms or uses the data of the program.

Included in the set of operator nodes are two nodes called "array fetch" and "array store". These nodes make explicit the additional addressing requirements attendant with the use of arrays. These nodes do not "manipulate" data, of course, so it is a stretch to put them in this group, but they do not fit anywhere else and making a third group just for them would unnecessarily complicate matters.

The second group, execution-control nodes, contains all of the other nodes of the PDW. This group in concert provides the operational semantics for each of the execution disciplines. These nodes

copy or consume data (the two Switch nodes and the η^F node);
regulate initial and iterating loop values (the β node);
generate demands for the evaluation of loop bodies (the η^F node);
properly sequence the output of the two branches of conditional statements (the γ node); and
mark the beginning and completion of execution (the Entry and Exit nodes)
represent a set of control conditions (regions).

These nodes behave or are used differently under each execution discipline. It is this differential in their activity and use that realizes the different execution disciplines. Note that the region nodes partition the graph [4].

4 Not to be confused with the "demand graph" used in Section 2.3 on page 5 above to explain demand-driven execution.

The arcs of the PDW also fall into two groups: control dependence arcs (used for control-driven execution only) and data dependence arcs. The $data^{flow}$ arcs are used for data- and demand-driven execution only. $Data^{flow}$ arcs represent flow dependences; $data^{anti}$ represent anti-dependences and $data^{output}$ represent output dependences. The $data^{anti}$ and $data^{output}$ arcs are remnants from the PDG. Assuming that all operations in the PDW observe single-assignment semantics, then neither of these latter sets of arcs are necessary for execution under any discipline. As noted above, single-assignment semantics stipulate that each variable is assigned a value at most once. (The source code is not under this constraint.) For example, Figure 6 shows two statements using elsewhere-defined functions `f` and `g`, that have a true dependence on the variable `y`, and the associated PDW fragment, with only $data^{flow}$ arcs shown. (Consistent with single-assignment semantics assume that there are no definitions of the variable `y` between statements 1 and 2.) Variable names are shown adjacent to each arc. Note that statement 2 is not able to execute until statement 1 executes because statement 2 requires the value of `y` computed in statement 1. This is indicated by the $data^{flow}$ arc connecting the output of the top box with the input of the lower box.

Figure 6. A Flow Dependence

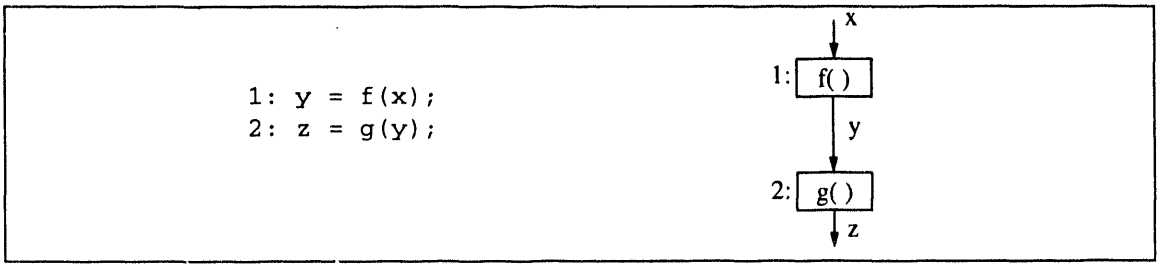


Figure 7 shows an anti-dependence. The arc connecting the two boxes in the associated PDW fragment is a $data^{anti}$ arc, shown as a dotted line to distinguish it from the $data^{flow}$ arcs, shown as solid lines. The output of the `g ( )` box is labelled x' to distinguish it from the input to the `f ( )` box. (Recall that the source code is not confined to single-assignment semantics.)

Figure 7. An Anti-Dependence

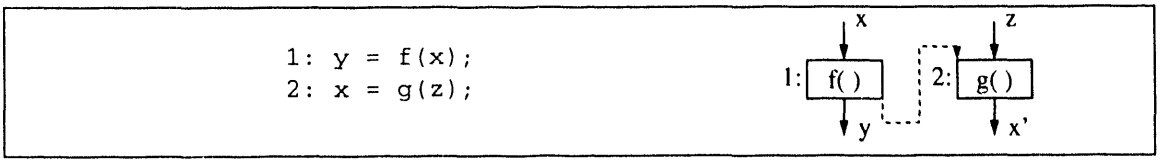
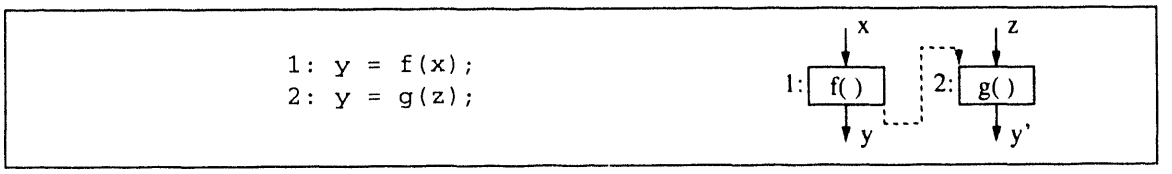


Figure 8 shows an output dependence, shown again as a dotted line. Note again that there is no $data^{flow}$ arc connecting the two boxes in the associated PDW fragment. The output of the `g ( )` box is labelled y' to distinguish it from the output of the `f ( )` box. (If the only effect of `y = f(x)` were to assign a value to `y` (i.e., no side-effects from the computation of `f(x)`), then an optimizer could eliminate the first statement entirely.)

Figure 8. An Output Dependence



Arrays in loops present the largest challenge to single-assignment semantics in a parallel world. There are two approaches to this challenge. The classic approach, based on languages such as Fortran, assumes sequential execution and allows parallel access to an array only if there is a guarantee that race conditions do not exist. The alternate

approach, based on functional languages, assumes parallel execution and prevents copying of an array only if there is a guarantee that race conditions do not exist. "I-structures" in the Id language [3] provide a blend of these two approaches. I-structures are arrays with elements than can be written to only once but read from many times (readers who issue their request before the write has occurred are automatically queued). I-structures thus assume parallel execution but minimize the copying of arrays. That is, if there is no guarantee of race conditions, the array is copied ([31]). These approaches introduce a variable that, from the viewpoint of this research, needs to be constrained. Accordingly, this research will treat arrays as monoliths: dependence between array elements is not considered ([43], [45]). There are drawbacks to this position, of course, but its value is that it preserves a tractable problem. Meanwhile, parallelism can be manually exploited via the "parallel" statement (see Section 6.1.1 on page 25).

A detailed description of each of these nodes, exactly how they operate under each execution discipline and exactly how they are arranged to provide the operational semantics of loops, conditionals, etc., can be found in the references ([4], [9]).

4.1 An Example PDW

A PDW for the program fragment of Figure 3, "Factorial Fragment," on page 4 above is shown in Figure 9 below.

The boxes in Figure 9 denote different PDW nodes, such as "write," "β," "T" (representing Switch^T nodes), etc.

The circle to the left of each box in the Figure denotes the control dependence region of the associated node. For example, the "read" node at the top of the Figure is in control dependence region "1t," and the two "T" nodes are in control dependence region "2t." The circle to the right of a box denotes the root of a control dependence region. For example, the predicate node (">") is the root of control dependence region 2.

The arrows indicate data^{flow} arcs that carry data from a source node to one or more target nodes. There is a queue for each arc entering each node. The arcs are "piped," meaning that they can hold an unbounded sequence of values but the target node can only access the first value in that sequence. The arc junctions, such as the one directly below the right β node in the Figure, denote "split" functions: a value flowing on that arc is replicated to each output of the junction. These arc junctions could be replaced by new nodes called "split," say, but this would unnecessarily clutter the graphs.

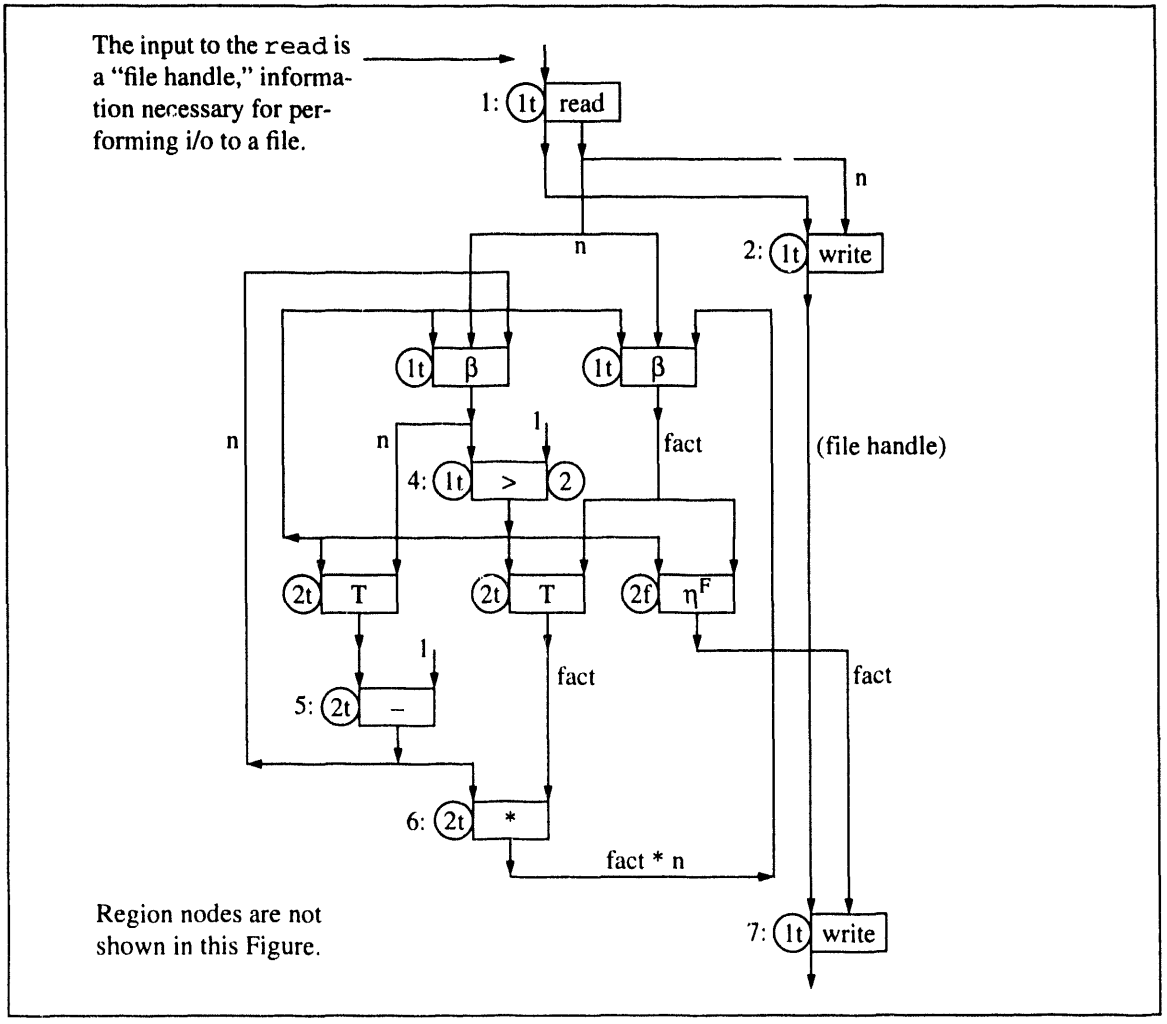
Constants are shown by a short arc with an adjacent integer, such as the right input to the predicate (">") node in the middle of the Figure. The adjacent "1" indicates that the integer value 1 is always available to the predicate node as input.

For ease of reference the statement numbers from Figure 3 are shown to the left of each corresponding PDW node. Note that there is no node in the PDW corresponding to statement 3: `fact = n`. The effect of this statement is taken care of by the split function implicit in the arc junction between the read node and the β nodes. Note also that the β, T, and η^F nodes have no associated statement numbers. These nodes provide the necessary operational semantics to drive the `while` loop. Also, as a reference aide, the variable names and expressions from Figure 3 are shown adjacent to various arcs in Figure 9. Execution of the Figure is explained in the next section.

4.2 The PDW and The Execution Disciplines

The PDW provides different execution disciplines by using the arcs and nodes in different ways. Under control-driven execution the queue associated with each arc has a capacity of one; putting a second data item on the arc overwrites an item already in the queue. Under data- and demand-driven execution the queues have unbounded capacity. The behavior of the nodes under each of the execution disciplines is consistent with each discipline. The details are available via the references [9].

Figure 9. PDW for Figure 3, "Factorial Fragment," on page 4



Control-driven execution is implemented by letting a single control thread flow through control dependence regions, with nested regions executing as soon as they are entered. Within regions execution is constrained by data dependences that are represented by data arcs. Any execution sequence that does not violate these constraints is acceptable. (Note that control-driven machines usually follow the total ordering explicitly specified by the source code. The PDW reveals other orderings for control-driven execution that will provide the same output. See Section 2.1 above.) Data-driven execution is implemented by letting each node execute as soon as it has received enough of its inputs to do so. Demand-driven execution is implemented by letting a node execute if it has been demanded and either (a) it has not propagated demands, or (b) it has received enough of its inputs to execute. In the former case the node first propagates demands, then waits for the inputs that are the fulfillment of those demands; in the latter case the node actually executes, fulfilling the demand placed on it. Execution under each of the disciplines of the PDW shown above in Figure 9 is explained below.

4.2.1 Control-Driven Execution

Under control-driven execution the nodes in control dependence region 1t execute first. They execute in an order that is consistent with the data dependences indicated by the data^{flow} arcs. The `read` node executes first, followed by the `write` associated with statement 2 of Figure 3 and the two β nodes. Sometime after the left β node has completed

execution, the predicate (">") executes. Assuming that the value read in is 2, then the predicate evaluates to *true*. This activates control dependence region 2t. The two T nodes in that region execute, followed by the subtract, followed by the multiply, followed by the two β nodes again. The predicate this time evaluates to *false*. This activates control dependence region 2f. The η^F node in that region now executes for the first time, outputting the second input from the right β node to the `write` associated with statement 7 of Figure 3, completing execution of the fragment. (The first input from the right β node, in the queue to the η^F node, was overwritten when the second input from the right β node was generated.)

4.2.2 Data-Driven Execution

Under data-driven execution the `read` node must execute first because it is the only node with all of its inputs. The `write` associated with statement 2 of Figure 3 and the two β nodes can then execute. They can do so sequentially, concurrently, or in any combination of those two. When the left β node has completed execution the predicate can execute, evaluating to *true*. The left T node can then execute. If the right β node has completed execution, then the right T and the η^F node can also execute. Because the predicate evaluated to *true*, the T nodes produce output and the η^F node does not. When the subtract completes execution the left β node can execute again, and when the multiply completes execution the right β node can execute again. Then, when the left β node completes execution the predicate can execute again, this time evaluating to *false*. When this *false* value reaches the β nodes they execute a third time, resetting themselves in anticipation of another execution of the loop. The T nodes do not produce output this time but the η^F node does. As a consequence neither the subtract nor the multiply execute. But the `write` associated with statement 7 of Figure 3 subsequently does execute, completing execution of the fragment.

4.2.3 Demand-Driven Execution

Under demand-driven execution, and assuming that the `write` associated with statement 7 of Figure 3 alone is demanded, then...

...that second `write` demands
the first `write`, that demands
the `read`, that demands
the "file handle" input (generated by a node outside the Figure),
and the η^F node, that demands
the predicate, that demands
the left β node, that demands
the `read` (already demanded),
and the "1" constant (always able to immediately fulfill a demand),
and the right β node that demands
the `read` (already demanded).

When the `read` receives the file handle input (from a node outside the fragment), that `read` reads from the file specified by that file handle. The `read` can then fulfill the demands placed on it by the first `write` node and the two β nodes. The first `write` can perform a write and fulfill the demand placed on it by the second `write`. Meanwhile, the two β nodes can execute, fulfilling the demands placed on them by the predicate and the η^F node. The predicate can then evaluate, and because the input read in is assumed to be the value 2, the predicate evaluates to *true*. The predicate fulfills the demand placed on it by the η^F node. At this point there is only one demand outstanding – the demand placed by the second `write` on the η^F node. The η^F node executes. Since the input that the η^F node received from the predicate was *true*, the η^F node generates a second set of demands that has the effect of executing the body of the loop. That is,...

...the η^F node demands
the predicate, that demands
the left β node, that demands
the predicate (already demanded)
and the subtract, that demands
the left T node, that demands
the predicate (already demanded)
and the left β node (already demanded)
and the constant "1,"
and the constant "1,"
and the right β node, that demands
the predicate, (already demanded)
and the multiply, that demands
the subtract (already demanded)
and the right T node, that demands
the predicate (already demanded)
and the right β node (already demanded).

Each of these chains of demands can execute without propagating further demands because the T nodes have enough inputs already waiting on their input arcs to execute. After these demands are fulfilled – and thus after the body of the loop has executed – the predicate evaluates to *false* and there is again only one demand outstanding – the same one as before. And again the η^F node executes. This time, since the output of the predicate was *false*, the η^F node fulfills the demand placed on it by providing output to the `write` associated with statement 7 of Figure 3. The `write` statement can then execute and thus fulfill the original demand, bringing execution of the fragment to a close.

Note that under demand-driven execution, like control-driven execution but unlike data-driven execution, values may be left waiting on various nodes after execution of the fragment has completed. The operational semantics of the nodes under demand-driven execution accounts for this by first clearing out these stale values when a node is again demanded – but not before. Doing so beforehand would violate the "lazy" nature of demand-driven execution. Under control-driven execution these stale values will be overwritten prior to the time when the receiving nodes execute again, so here too these stale values present no problem.

4.3 The PDW and the Variable-Execution-Discipline Machine Model

In the variable-execution-discipline machine model, a PDW resides in the Scheduling Unit (SU). The Execution Unit (EU) contains a corresponding list of tuples. Each tuple contains at least the following information: a PDW node type, memory addresses of input values to be fetched and memory addresses where output values are to be stored. Assume, before execution begins, that the desired execution discipline is already set in the SU (the EU operates the same under each discipline). As execution begins, the SU traverses the PDW and sends to the EU fire signals that the EU interprets as addresses of operations that can execute. When the EU completes execution of an operation, it sends the operation's address back to the SU in a done signal. If the operation is a predicate, then the EU includes a condition code that indicates whether the predicate evaluated to *true* or *false*.

5.0 Related Research

This section presents research on performance prediction, the algorithm-to-architecture mapping problem (defined below) and on control-, data- and demand-driven machines. There are applicable results in the first area; the other two areas are of interest but provide no results for this research. For example, there has been no research in the area of relative performance prediction for control-, data- and demand-driven disciplines.

5.1 Performance Prediction

The area of performance prediction that is relevant to this research is the development of heuristics that will predict the relative performance of a given computation under control-, data- and demand-driven execution. There are three studies that directly apply.

In the first study, Gonzalez & Ramamoorthy [16] show the value of a simple heuristic based on loops. Gonzalez & Ramamoorthy set out to determine whether or not a given Fortran program is "suitable" for parallel execution. By suitability they mean "either many parallel paths exist or, if the parallel paths are few, they must be long." ([16], p.647) They first describe a "recognizer" program that identifies parallel tasks. The recognizer is expensive: it requires $O(n^2)$ time *and* space. So the authors look for a heuristic. They perform a simple, arithmetic calculation that uses the size and number of loops and the relative frequency of other statement types in the program under consideration. This calculation provides a single number, Sf, that represents the "suitability" of a program for parallel processing. Based on four test programs they conclude that their cheap heuristic is valuable. The importance of this study for the purposes of the research in this proposal is the simplicity of the heuristic and its emphasis on loops.

In the second study, Gurd et al. present a way to predict speedup on their dataflow computer [19]. They measure "Sinf" – the number of execution steps required to execute a program given an infinite number of processors available. As a testbed they use a software simulator that makes several simplifying assumptions, namely, that "each instruction executes in the same time (execution therefore proceeds in discrete equal time steps)," and "that an unlimited number of function units can be used during any one time step." ([19], p.43) Sinf is described as the "longest cycle of dependent instructions" ([19], p.43) and as "the length of the shortest path." ([19], p.44) Gurd et al. test a number of other parameters, but they found that their best predictor was average parallelism ("avePara"), defined as S1/Sinf, where "S1" is the number of nodes in the graph representing the program.

> However, the major pattern to emerge from this study is the importance of the parameter avePara in determining the shape of the speedup curve for various programs. Programs with similar values of avePara exhibit virtually identical speedup curves. The higher the value, the closer the curve is to the 100 percent utilization rate. This seems to indicate that this crude approximation to the overall average parallelism of a code is all that is necessary for an accurate prediction of its speedup curve. This applies regardless of factors such as time variance of parallelism, the source language used, the proportion of one-input instructions executed, etc. It is surprising that such a simple measure should give such a constant indication of the pattern of use of processing resources, but it does help to answer the question of what nature a program has to have if it is to be suitable for execution on this dataflow system. A program is suitable if it has a value of avePara in the region of 40 or more. ([19], p.49)

Subsequent studies confirm that "average parallelism is a robust characterization of parallel processing task systems and only marginal benefits are achieved by considering the higher order moments of the parallelism of the task graph." ([15], p.617) The heuristics developed below attempt to predict characterizations similar to Sinf without having to execute the program.

In the third study, Knuth empirically examined a suite of Fortran programs and concluded that "less than 4 per cent of a program generally accounts for more than half of its running time." ([25], p.512) Knuth's "program profiles," along with estimated execution times per operation type, provide a way to estimate program performance. The study implies that the execution of loops dominates execution and thus it is the relative execution characteristics of a computation's loops that can predict relative execution of the entire computation. The same idea appears in Aho, Sethi & Ullman's compiler book: "There is a popular saying that most programs spend ninety per cent of their execution in ten per cent of the code. While the actual percentage may vary, it is often the case that a small fraction of a program accounts for most of the running time." ([2], p.585 (see also [20]))

5.2 Algorithm-to-Architecture Mapping Problem

Which architectures provides the best performance for a given algorithm? This is the algorithm-to-architecture mapping problem. Jamieson [23] is interested in the relationship between these two within the control-driven world:

> Ideally, a set of orthogonal characteristics would describe parallel algorithms and a corresponding set of orthogonal characteristics would describe parallel architectures, with a unique bijection performing the mapping from one to another. Experience shows that the relationship between parallel algorithms and parallel architectures is clearly too complex to conform to such a desirable model.

The algorithm characteristics that Jamieson considers include the nature and degree of the parallelism, the uniformity of operations (similar to grain size), synchronization requirements, static/dynamic nature of the algorithm, data dependencies, the fundamental operations of the algorithm (e.g., integer, floating point), data types and precision, memory requirements, data structures used, and I/O requirements ([23], p. 32). Jamieson is interested in matching algorithms and machines at an abstract level.

The research of this proposal is at a more concrete level: source codes – a subset of algorithms – are matched to a single machine model. At the same time the subject of this proposal is the algorithm-to-architecture mapping problem but with a wider domain. The research on this problem, such as Jamieson's presented immediately above, has focused on the control-driven, von Neumann world; the subject of this proposal considers a larger domain that includes data- and demand-driven machines.

Agrawal, et al. [1], for example, develop a way to choose "the appropriate architecture for a class of applications." Unfortunately, they only consider various network arrangements for classical control-driven von Neumann processors.

Denning & Adams discuss the "domain-architecture matching problem," described as "...which architectures are suitable for a given problem domain?" ([11] p.415) This question does not match algorithms with architectures but rather "problem domains" – a superset of algorithms – with architectures. Denning & Adams note that when there was only one choice, i.e., the "sequential-process architecture," this problem "never came up." Although results in this area could be applicable to the research in this proposal, Denning & Adams have none to report: their paper presents research questions.

5.3 Machines

For the purposes of this research there are three categories of machines: von Neumann, dataflow and reduction. These machines use control-, data- and demand-driven execution respectively. Von Neumann and dataflow machines have a typical hardware structure; reduction machines do not. Machines that use a melding of execution disciplines are referred to as "hybrids." Hybrids do not constitute a fourth category because, as is pointed out below, the execution

discipline they exploit requires a compiler, and compilers are beyond the scope of this research. Like the reduction machines hybrid machines do not have a typical hardware structure.

5.3.1 von Neumann

The typical von Neumann machine consists of two main parts: a control unit and a memory unit. The memory unit contains both operations and data and has no processing ability beyond presenting or changing the value of a memory location. The control unit executes the following cycle:

use the address in the Program Counter (PC) to fetch an instruction code from memory,
decode the instruction,
fetch the operands from memory,
perform the operation specified by the instruction,
store the results in memory, then
increment the PC and start the cycle again.

Real systems are more complex, of course, and include other levels of memory (registers, cache, disk, tape, etc.) and I/O systems but they execute using this same cycle. Virtually all of the computers in existence today are von Neumann machines.

5.3.2 Dataflow

The typical dataflow machine is built around a "circular pipeline" that, like the simplified von Neumann machine above, consists of two main parts: a "Matching" Unit and a Processing Unit. The output of the Matching Unit goes to the Processing Unit, and the output of the Processing Unit goes to the Matching Unit, hence the circular pipeline. "Tokens" flow between the units; each token consists of

a token identifier,
an operation,
slot (or slots) for the input data,
a slot for the output data, and
references to the tokens that are to receive a copy of the results of the operation.

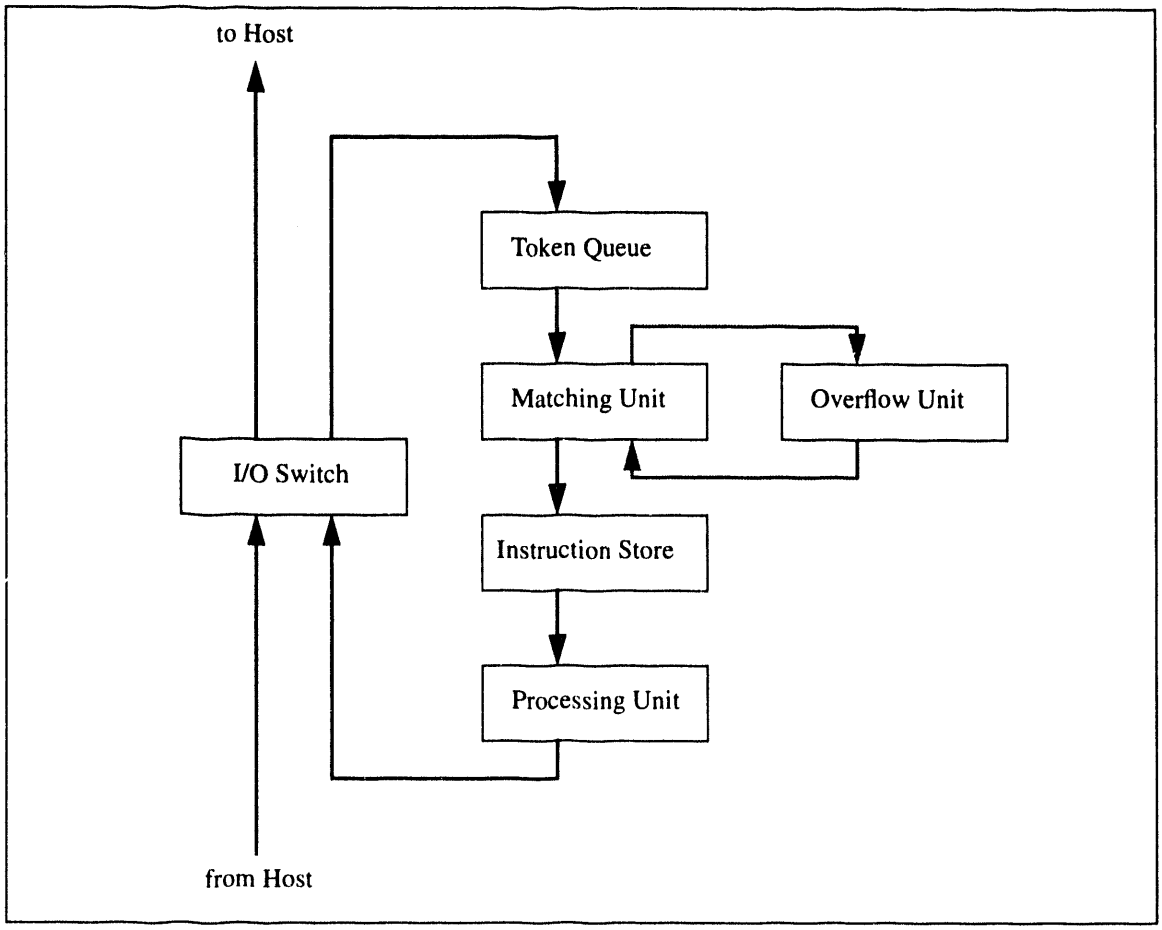
Since the data "flows" around the circular pipeline, instead of sitting statically in memory, these machines are referred to as *dataflow* machines. The Processing Unit transforms the data by executing the operation specified by the token in hand, clearing the input slots and filling the output slot, and sending the token on to the Matching Unit. The Matching Unit copies the output data from a token whose operation has completed to the input slots of the appropriate tokens. If any of these output tokens are not present, the Matching Unit retains the input so that it can copy it into the tokens when they arrive. Meanwhile, if any tokens have enough inputs to execute, then the Matching Unit sends them on to the Processing Unit. The Processing Unit has a number of independent functional units; parallelism is realized by simultaneous execution of these functional units.

The variable-execution-discipline machine model presented in this proposal has the same structure as the simplified dataflow model presented above: two units – one a processing unit – connected in a circular pipeline. The difference between the two models is what flows through the pipeline. In the dataflow model tokens contain application-level data flow; in the variable-execution-discipline model signals contain no application-level data.

Real dataflow systems need more than just the two units discussed here. The system structure for the Manchester Dataflow Prototype Computer [19] is shown below in Figure 10. The Token Queue buffers the tokens between the Processing Unit and the Matching Unit. The Overflow Unit provides extra storage for un-matched tokens. The

Instruction Store contains instructions; this lightens the requirements of the information that is carried in the tokens, at the expense of lengthening the pipe. The I/O Switch provides communication with the outside world.

Figure 10. Manchester Dataflow System Structure



5.3.3 Reduction

There is no one typical reduction machine. Some, such as the GMD Reduction Machine [40], are based on stacks: the program is scanned for expressions that can be reduced and then become grist for more scanning. Others, such as the North Carolina Cellular Tree Machine [28], are based on a tree structure: the program, residing in the leaves of the tree, is scanned by the processors – that constitute the rest of the tree – for expressions that again can be reduced and become grist for more scanning. Still others, such as Turner's S-K reduction machine [40], are "implementation techniques" for functional languages or are built on "conventional microprocessors," such as the SKIM reduction machine [40], or are simply abstract machines implemented on a von Neumann machine, such as Koopman & Lee's TIGRE machine [26].

5.3.4 Hybrid

Hybrid machines combine two execution disciplines in an attempt to glean the best of both. Usually it is control- and data-driven disciplines that are hybridized. The approach has been to schedule groups of operations, known as "threads," instead of just one operation at a time. When the data required for the first operation in the thread is avail-

able, then the entire thread can execute in sequence. The threads themselves are scheduled via the data-driven discipline; the operations *within* the thread are scheduled via the control-driven discipline. The hope is to increase performance by avoiding the extra overhead for data-driven scheduling when it is not needed, such as within threads, but paying for data-driven scheduling between threads where parallelism can be exploited. Some of these control/data hybrids start with a von Neumann machine and add dataflow features ([8], [22], [31], [30]); others start with a dataflow machine and add von Neumann features ([33], [14], [17] & [18]). Hybrids have also been developed between data-driven and demand-driven execution ([35], [42]).

Note that each of these hybrid machines depends upon a compiler that works prior to the execution of the program. In the case of control/data hybrids the compiler creates threads. In the case of data/demand hybrids the compiler performs "program transformations" or creates an "extended dataflow graph" for use at run-time. Since the focus of this research is in the relationship between performance and execution disciplines and not <execution discipline, compiler> pairs, the variability due to compilers is held constant by avoiding them, and thus the approach in this research is independent of the hybrid machines.

6.0 Research Plan

The purpose of this research is to test the hypothesis that varying execution discipline can increase performance and that this relationship can be heuristically predicted. Testing the hypothesis requires two tools, each of which maps a computation and an execution discipline to a performance:

1. A simulator for the variable-execution-discipline machine model that will simulate execution of a PDW in control-, data- and demand-driven fashion.
2. Heuristics that will map a computation and an execution discipline to a value representing the predicted execution time of that PDW under that discipline.

Three ancillary pieces are needed in order to test the hypothesis using the two tools above:

A source-to-PDW translator,

A suite of computations, and

An operation table that maps an operation (e.g. add) to a value representing the execution time of that operation.

The two tools and the three ancillary pieces will enable the following:

1. Evaluation of the proposition that different computations exhibit different performance under different execution disciplines. This evaluation can be done by comparing the performance of individual computations under those different disciplines.
2. Evaluation of the proposition that the performance differences hypothesized above can be effectively predicted by heuristics. This evaluation can be done by comparing the predictions produced by the heuristics with the performance produced by simulated execution.

This will enable testing of both parts of the hypothesis. This section describes the tools and the ancillary pieces and how they will be used to test the hypothesis.

6.1 Ancillary Pieces

Three pieces are needed for the operation of the simulator and the heuristics. The first piece is a translator that will produce a PDW when given a program source code. The second piece is a suite of computations to run through the translator. And the third piece is an operation table that maps an operation to a value representing the execution time of that operation. Each of these pieces is presented below.

6.1.1 Source-to-PDW Translator

The source-to-PDW translator is being developed in Objective-C as part of another project in the Computer Science Department at UNM [27]. The source accepted by the translator is a restricted, simpler form of C. For example, pointers and structures are not in the restricted language.[5] (Arrays are *in* the restricted language.) The extensions added to the language include a parallel statement and provisions for trip and branch predictions. For example, the code in Figure 11 is an adaptation of the code from Figure 3 and shows all three of these extensions.

The branch prediction in line 2, `[5%]`, specifies that the user predicts that 5% of the time the *true* branch of the conditional will be taken. The `parallel` statement starting on line 5 indicates that statements 6 and 8 can execute in

5. The absence of pointers and structures in the restricted language is a constraint imposed by those developing that restricted language. It is not a constraint imposed by this research.

Figure 11. Code Exemplifying Extensions (See Figure 3)

```
 1:  read  (n);
 2:  if [5%] (n < 0)                    <--branch predictor
 3:       write ("bad input\n");
 4:  else {
 5:       parallel                      <--parallel statement
 6:            write (n);
 7:       and
 8:            fact = n;
 9:       while [10] (n > 1) {          <--trip predictor
10:            n = n - 1;
11:            fact = fact * n;
               }
12:       write (fact);
     }
```

parallel. The trip prediction for the `while` loop on line 9, `[10]`, indicates that the user predicts that the body of the `while` loop will execute 10 times, on the average, each time the loop executes. The grammar is not powerful enough to indicate that the first execution of statement 9 for an execution of the loop can execute in parallel with statements 6 and 8, and that each subsequent execution of statement 9 for an execution of the loop can execute in parallel with statement 11. (The PDW, however, is able to exploit that parallelism.)

6.1.2 Computation Suite

This section defines "computation" and describes the computation suite used as input to the simulator.

For the purposes of this research the term "computation" is defined to be a 3-tuple,

<program_source_code, input_data, demanded_output>

where

"program_source_code" is the source code of the program to be executed,
"input_data" is an instance of the input values required to simulate execution (at least)
(i.e., values used to control loop iteration and conditional branching), and
"demanded_output" is an instance of the operations whose results are demanded.

By convention, if demanded_output is not provided, then all of the statements are demanded. Demanded_output is ignored under control- and data-driven execution.

The gauge for this research is performance, so programs developed to gauge performance are needed. Perhaps the most well-known set of such programs is the LFK Test (a.k.a. the Livermore Loops) [29]. The LFK Test contains a "spectrum of CPU-limited computational structures" in the form of 24 loops or "kernels" extracted from programs constituting the core of the workloads at Lawrence Livermore National Laboratory. These kernels are "small, key pieces from real programs." [20] Each kernel in the LFK Test will be augmented with input_data and with demanded_output to form computations that will serve as grist for the simulator and the heuristics presented below.[6] A review of the kernels themselves indicates that in almost all of them the loops are controlled by an induction variable. Almost two-thirds of the kernels are singly-nested loops. Only one kernel has a triply-nested loop. Only five have branch statements (see Table 5).

6. The LFK Test is written in Fortran. The C version of the loops, obtained from Livermore, will be used in this research.

Table 5. Characteristics of the LFK Test Kernels

Characteristic		Number of Applicable Kernels
Maximum Nesting-Level	Single	15
	Double	7
	Triple	1
Loop Iterations Controlled by Induction Variable		23
Branch Statements		5
gotos		1

6.1.3 Operation Table

An operation table, shown below as Table 6, maps an operation to a value representing the execution time of that operation. This table is used by the simulator and the heuristics. The purpose of the table is to isolate operations that depend only on the execution discipline and to parameterize the performance of the other operations of the machine model, providing a more realistic environment.

There are four types of operations:

1. Arithmetic Logic Unit (ALU) operation:
   Arithmetic operations: addition, subtraction, multiplication and division for integers and floating point numbers;
   Comparison operations (PDW predicate nodes): <, <=, =, !=, >, and >=.
2. Memory-cpu transfer operations:
   Fetching and storing of scalars or array elements.
   (Array fetch and array store appear as PDW nodes labelled "=[ ]" and "[ ]=" respectively. See Figure 13 in the Appendix below for an example.)
3. PDW intrinsic operations:
   PDW nodes Entry, Exit, $Switch^T$, $Switch^F$, η^F, γ, and β.
4. Scheduling operations:
   For control-driven execution: updating the Program Counter;
   For data-driven execution: matching;
   For demand-driven execution: generating demands, and matching.

There are four PDW nodes not explicitly listed above: operator, read, write and region. The "operator" nodes include the arithmetic operations as well as the array fetch and array store nodes, so these nodes span two of the operator type categories listed above. The read and write nodes do not appear at all in the table because the suite of computations that will be used contains no I/O. The region nodes represent common control conditions and are unnecessary even for control-driven execution.

The purpose of the array fetch and array store operations is to make explicit in the PDW the addressing overhead due to arrays. The array fetch consists of an integer add – array base address plus offset – and a scalar fetch – to get the array element. Using Table 6 this is i_add + s_fetch = 2 + 3 = 5. This operation appears in a PDW as a node labelled "=[ ]." The array store consists of an integer add – array base address plus offset – and a scalar fetch – to determine

the effective storage address. Using Table 6 this is i_add + s_fetch = 2 + 3 = 5. This operation is appears in a PDW as a node labelled "=[ ]."

Table 6. Operation Table

Type Description	Subtype Description	Operation	Sub-operation	Parameter	Execution Time[a]
ALU Operations	Arithmetic Operations	Addition	Integer	i_add	2
			Float	f_add	3
		Subtraction	Integer	i_sub	2
			Float	f_sub	3
		Multiplication	Integer	i_mult	6
			Float	f_mult	8
		Divide	Integer	i_div	8
			Float	f_div	10
	Comparison Operations[b]	<, <=, =, !=, >, >=		compare	2
Memory-CPU Transfer Operations		Fetch	Scalar	s_fetch	3
			Array	a_fetch	5
		Store	Scalar	s_store	3
			Array	a_store	5
PDW Intrinsic Operations		Entry, Exit, Switch^T, Switch^F, η^F, γ, β		pdw	1
Scheduling Operations		Update Program Counter		pc	0
		Match 1 Input		match	0.5
		Propagate 1 Demand		demand	0.5

a. The numbers in this column are inspired by those used in Knuth's study on FORTRAN programs [25].

b. These are PDW "predicate" nodes.

It is assumed for all three disciplines that instructions always have to be fetched, so instruction fetch does not appear explicitly in the table but is part of each operation. Similarly, the machine model being used does not require the duplication of intermediate results so this activity also does not appear in the table.

The performance of ALU operations are the same for each discipline. These operations are confined to the ALU and are independent of any other activity, including the execution discipline in force.

Similarly, the performance of Memory-CPU Transfer operations can be assumed to be the same for each discipline. Pure data-driven machines do not fetch or store – the operands and operators all flow with the instructions in tokens. However, there is a performance cost associated with that flow, just as there is a cost associated with fetching and storing. (Included in the cost of "flowing" is the cost of the duplication of intermediate results.) For the purposes of this research the performance costs for fetching and storing are assumed to be equivalent to the cost of flowing (and duplicating). This simplification is possible because of the machine model being used. Note that this does not rule out increases in performance due to registers or cache. The activity of registers and cache will be reflected in a parameter that is the cache hit rate presented below.

The performance of PDW Intrinsic operations can be considered to be the same for each discipline. These nodes provide different functionality and act differently under each discipline but the effect of these differences on performance is small and can be ignored.

The performance of Scheduling operations, unlike the other three, are completely dependent on the discipline. For control-driven execution, scheduling is confined to updating the Program Counter. Matching, and propagating demands do not apply. For data-driven execution, scheduling involves matching intermediate results – this is what the Matching Unit does in the Manchester Dataflow System, for example. In this case updating the Program Counter and propagating demands do not apply. And for demand-driven execution, scheduling involves the generation of demands as well as matching. As with data-driven execution, updating the Program Counter does not apply.

Two additional parameters are needed. The first, referred to above, is the hit rate for the cache (and the registers). It is represented as a number between zero and one. It indicates the percentage of the time that the target of a fetch is found in the cache (or registers). If hit_rate is zero, then the machine being simulated has neither registers nor cache; and if it is one, then, like the Cray machines, cache is the entire memory. Hits are assumed to reduce fetch time to zero. So, if hit_rate is 0.8 and a fetch takes 3 units of time, then the effective fetch rate is

$$3 * (1-0.8) = 3 * 0.2 = 0.6.$$

The second parameter is number-processors, the number of processors available to the Scheduling Unit (SU) and Execution Unit (EU).

Some experimentation with the setting of hit_rate is anticipated. More extensive experimentation is anticipated with number-processors. Hit_rate has not been applied to the values for fetch and store in the operation table above. Both parameters are shown in Table 7 below.

Table 7. Cache and Multiprocessor Parameters

Type	Parameter	Setting
Cache (and register) hit rate	hit_rate	0.8
Number of Processors in SU and EU	number-processors	–[a]

a. The heuristics assume sufficient processors are available and thus make no use of this parameter. There is no reason to provide a setting for this parameter here.

6.2 Simulator

The variable-execution-discipline machine model developed above is not an actual machine (it is not a "real system" in modelling parlance [44]) so the machine must be simulated. The parameter of importance is elapsed execution

time. This is the "experimental frame." The "base model" is the set of actual subcomponents of the real system that, unfortunately, does not exist, as noted above. The "lumped model" is a simplified version of the base model that will facilitate simulation.

The results of each run of the simulator will be a number that will indicate performance. For the purposes of this research "performance" will be defined as elapsed execution time and it will be measured by simulated clock cycles.

The simulator must be able to do the following:

1. Simulate simultaneous execution of the SU and EU.
2. Reflect the time required to schedule nodes in the SU and the time required for different operations in the EU (e.g., addition should take less time than division).
3. Simulate multiprocessing capabilities for the SU and EU (i.e., simulate multiple processors in both SU and EU).
4. Provide the simulated time for the simulation run when execution completes.

6.2.1 Design

The simulator consists of the following pieces:

1. Two processing Units,
   the SU and EU, corresponding to the Scheduling Unit and Execution Unit respectively, based on the variable-execution-discipline machine model developed above;
   (these processing Units may each contain many subsidiary processors);
   (the SU holds two special flags:
   "discipline" that indicates the execution discipline to be used, and
   "demand_only" that indicates that demand-driven execution has not yet commenced).
2. Two lists,
   the fire list that communicates fire signals from the SU to the EU, and
   the done list that communicates done signals from the EU to the SU.
3. A controller that initializes the lists and flags, initiates the processing Units and passes control between the two Units as execution proceeds, then provides the output when the simulation run is complete.

The controller initializes lists, flags and data as summarized in Table 8 below. Under control-driven execution the Entry node provides the SU with all that it needs to begin execution. The Entry node does not "execute" per se, so this node is loaded into the done list and execution begins with the SU. Under data-driven execution the PDW nodes that can execute initially are loaded into the fire list. These nodes are ready to execute, so execution begins with the EU, not the SU. And under demand-driven execution the PDW nodes initially demanded are loaded into the done list, and like control-driven execution, execution begins with the SU. The flag "demand-only" is set, directing the SU to bypass its matching phase for these nodes and proceed to generating demands. When the SU has completed these demands, it resets the demand_only flag so that the next time the SU executes it performs matching as well as demand propagation.

Execution proceeds by the controller passing control back and forth between the SU and EU. When the fire list is empty after the SU has completed a turn at execution, execution of the PDW is complete.[7]

7. The EU always produces exactly one node in the done list for each node in the fire list so termination of execution cannot hinge on the contents of the done list.

Table 8. Initialization of the Simulator

	Control-driven	Data-driven	Demand-driven
Contents of Fire list		(Nodes initially ready to execute)	
Contents of Done list	Entry node		(Nodes initially demanded)
Flag settings	discipline = control	discipline = data	discipline = demand demand_only = *true*
Data Initialization	(As needed)		
Unit that Executes First	Scheduling Unit	Execution Unit	Scheduling Unit

Simulation of simultaneous execution of the SU and EU requires the use of a "time-stamp" in each signal. Each time a signal passes through the SU or EU, the simulated execution of the work associated with that signal is added to the time-stamp in the resulting signal.

Simulation of variable execution times of different PDW nodes is provided by the use of operation table presented above (see Table 6).

Simulation of multiprocessing capabilities for the SU and EU requires maintaining a min-priority queue (or its equivalent) that represents the processors for each Unit. The time value of the node at the head of the queue is always the smallest and thus always represents the next available processor. When the EU executes an operation, as directed by an entry in the fire list, it sets the current simulated time to that of the processor at the head of the processor queue. The new completion time of this processor will be the old completion time plus the time required to execute the given operation. This can be calculated at the time of the assignment. This new completion time is included in the done signal that is sent to the SU. The queue is then adjusted so that the node at the head of the queue once again has the smallest time value. Multiprocessing for the SU works in a similar fashion. For both the SU and EU, all of the processors have an initial completion time of zero. The execution time for the PDW is the largest completion time of any of the processors when execution is complete.

It is assumed that the SU and EU allocate processors on a per operation or node basis. That is, when the EU allocates the execution of operation X to processor Y, processor Y alone fetches the inputs, executes the operation and stores the output, even if other processors are idle. The processor to which the SU allocates the scheduling of a node performs the updating of the Program Counter or the matching and/or demand propagation, as appropriate for the execution discipline in force. The effect of this is that multiple matches on the same node are not overlapped: the time for two matches is twice the time for one match. The same is true of multiple demand propagations. The justification for this approach is that allocation on a sub-operation basis is excessively fine-grain for the EU, and dataflow machines typically have only one matching unit.[8]

6.2.2 Implementation

The simulator will be implemented using the object-oriented language Objective-C. This will enable building on top of the project developing the source-to-PDW translator mentioned above. Several new classes are required: one for

8. A dataflow machine with multiple matching unit has been proposed but it is uncertain whether or not it has ever been implemented [34]. Multiple matching units introduce problems of their own that limit performance [15].

the Scheduling Unit, another for the Execution Unit, a third for the fire and done lists, a fourth for the min-priority queue that implements the processor list, and a fifth that implements a function for the operation table. A controller is also needed. The structure of the new classes and the controller are already in place. The contents of the fire and done lists will be pointers to nodes of the PDW to be executed. When the fire list is empty after the SU has completed a turn at execution, the simulated execution can halt. The execution time of the simulation run is the largest value of completion time of the simulated processors.

6.3 Heuristics

The heuristics developed below predict the relative performance of a computation under each of the disciplines without actually having to execute that computation. That is, the heuristics map a computation to at least three values:

predicted elapsed execution time for control-driven execution,
predicted elapsed execution time for data-driven execution, and
predicted elapsed execution time for demand-driven execution.

For each execution discipline the heuristics presented below provide four predictions (or only two if there are no branches in the source code). The heuristics assume that the number of processors in the SU and EU is unbounded. This delivers the best possible prediction for data- and demand-driven execution and does not effect the prediction for control-driven execution since it is assumed to be sequential. At this point in the research there is no compelling reason to factor the number of processors in to the heuristics. However, as the research proceeds this may change.

6.3.1 Design

The design of the heuristics will be explained by focusing on individual nodes, then expanding the focus to conditionals and loops.

The predicted time required for an individual PDW node to execute depends on
the type of the node and
the scheduling discipline being used.
All nodes require two inputs requiring two fetches. (The β node requires only one input on the first iteration of a loop but this exception is infrequent and can be ignored.) Only a single output of any node needs to be stored, regardless of the number of targets of a node's output in the PDW. This is because the intermediate data in the machine model is stored in a memory and not "flowed" through a circular pipeline. So, for each discipline, the following common activities are required by the EU:
fetching of the two inputs,
execution of the node itself, and
storing of the one output.
Control-driven execution requires in addition
updating the Program Counter.
Data-driven execution requires in addition
matching two inputs.
And demand-driven execution requires in addition
matching two inputs and
propagating two demands.
Note that the matching and the propagating of demands are assumed to proceed sequentially.

The predicted time required to execute the body of a loop one time, referred to as "iteration time," depends on
the nodes in that loop, and
the length of the longest path through the loop
(if the scheduling discipline is data- or demand-driven).

For control-driven execution the iteration time is the cumulative time for each of the nodes in the loop. For data-driven execution the iteration time is the longest path through the loop. This is the same as "Sinf" that is so effective in predicting performance of the Manchester Dataflow System (see "Performance Prediction" on page 19 above). For demand-driven execution the iteration time is the longest path through the demanded portion of the loop. The time required to execute the arm of a branch statement is determined in an analogous fashion.

The predicted time required to execute a loop depends on
the loop's iteration time (discussed above) and
the number of times the loop body is predicted to execute,
referred to as "trips."
The heuristics make two predictions. One prediction is based on the trip predictions provided by the user in the source code (see the example in Section 6.1.1 on page 25 above). This is referred to as "trip predicted." The second prediction ignores those user-supplied predictions and sets the trips to T_L. This is referred to as "trip unpredicted." The predicted execution time for the loop is the product of the iteration time, presented above, and the predicted number of trips. For example, let the iteration time of a loop be $T_{iteration}$, and the predicted number of trips be *trips*. Then the predicted execution time for the loop is

$$T_{iteration} * trips$$

The trip prediction for nested loops are multiplied by T_N. T_L is initially set to 5 and T_N to 10, as shown in Table 9 below.

Table 9. "Trip Unpredicted" Loop Parameters

Description	Parameter	Setting
Trips for Outermost Loops	T_L	5
Factor for Trips for Nested Loops	T_N	10

The predicted time required to execute a branch statement depends on
the body in each arm and
the percentage of time that the *true* arm is executed.
As with the loop, the heuristics make two predictions for the arms. One prediction is based on the branch predictions provided by the user in the source code (see the example in Section 6.1.1 on page 25 above). This is referred to as "branch predicted." The second prediction ignores those user-supplied predictions and assumes that the branch predictions are all 50%, indicating that the probability of the *true* arm executing is the same as the probability of the *false* arm executing. This could be referred to as "branch unpredicted" – this would make it symmetric with "trip unpredicted" – but it can be more accurately referred to as "branch average." The predicted execution time for a branch statement is the average predicted time for the two arms. For example, let the prediction execution times of the *true* and *false* arms of a branch be T_t and T_f respectively, and let P be the branch prediction. Then the predicted execution time for the branch is

$$(T_t*P + T_f*(1-P)) / 2$$

The elements involved with the calculation of predicted execution time are summarized in Table 10 below.

Since there are two ways to calculate the trip prediction and two ways to predict the branch prediction, there are four heuristics:
trip predicted & branch predicted;
trip predicted & branch average;
trip unpredicted & branch predicted; and
trip unpredicted & branch average.

Table 10. Calculation of Predicted Execution Time

Focus			Control-driven	Data-driven	Demand-driven
Individual node	Unit that Does the Work	EU	fetch the two inputs, execution of the node itself, store the one output		
		SU	update the Program Counter	match two inputs	match two inputs, propagate two demands
Loop Body ("iteration time") (or Branch Arm)			cumulative time for each node in the loop	longest path through the loop	longest path to the demanded node
Loop			iteration time * the number of trips the loops is predicted to execute		
Branch statement			(the time to execute the *true* arm * (branch prediction) + the time to execute the *false* arm * (1 - branch prediction)) / 2		

For convenience each has been given a name that is simply the first letter of the words that describe the heuristic, as shown in Table 11 below.

Table 11. Heuristics

Heuristic name	Trip function	Branch function
TpBp	Trip predicted	Branch predicted
TpBa		Branch average
TuBp	Trip unpredicted	Branch predicted
TuBa		Branch average

The calculation of the heuristics on a computation containing the first kernel of the LFK Test is shown in Section 8.0 on page 37 below.

6.3.2 Implementation

The heuristic algorithms require implementation of functions that calculate

the predicted execution time of an individual node,
the cumulative predicted execution time for all nodes in a loop body or conditional arm,
the longest path through a loop body or conditional arm,
the longest path to a demanded node of a loop body or conditional arm, and
trip and branch predictions.

These functions can be implemented by simple, O(n) algorithms.

7.0 Conclusions

For some loops the vector units in supercomputers provide better performance than the scalar units. For other loops the scalar units provide better performance than the vector units. Overall performance can increase if the appropriate unit is chosen for each loop. That is, varying the choice of hardware unit can increase performance. The research discussed in this proposal hypothesizes that a similar relationship exists for different execution disciplines. That is, the hypothesis is that varying the execution discipline can increase performance. The "cash value" of this research is the exploration of an avenue that could provide better performance. The intellectual value is the exploration of an hypothesis that would subsume three approaches that are currently disjoint.

The hypothesis has two parts:

1. Different execution disciplines exhibit different performance for different computations, and
2. These differences can be effectively predicted by heuristics.

Testing of the hypothesis uses the following major pieces:

1. A "variable-execution-discipline" machine model based on Gao's "argument fetch" architecture;
2. The Program Dependence Web (PDW), an intermediate program representation that can be interpreted in control-, data- or demand-driven fashion;
3. The LFK Test (a.k.a. the Livermore Loops) as sample program source code;
4. A simulator that provides performance figures based on simulated execution of the machine model under each of the execution disciplines on the LFK Test; and
5. Heuristics that generate predicted performance figures for the machine model.

The first part of the hypothesis will be tested by simulating the execution of the machine model on a suite of computations, based on the LFK Test, using all three execution disciplines. The second part of the hypothesis will be tested by comparing the results of the simulated execution with the predictions produced by the heuristics.

If the hypothesis is supported, then the model that should be used for the execution disciplines is not that of competitors but that of different tools, each with its own strength and area of application. Such a result opens the door to machines that can vary execution discipline based on the program to be executed, a control/data/demand hybrid machine – mentioned in Section 3.0 above– that could use different execution disciplines for different parts of the same program.

If the hypothesis is not supported, then the value of alternative execution disciplines, namely data- and demand-driven execution, is open to question. That is, if the only criterion is performance, then further research on data- and demand-driven execution is at a tangent to research on control-driven execution.

8.0 Appendix: Example Calculation of Heuristics

This section applies the heuristics presented above to an example computation in order to clarify the nature of the heuristics. A computation is shown below in Figure 12. The program_source_code is the first kernel from the LFK Test. The loop prediction on the first line is an addition, of course. The input_data is `n=6`, indicating that the loop will actually iterate 6 times. The initial values for x, q, y, r, z and t are unimportant for this example because they are not used to control loop iteration (and there are no branches in the code). The demanded_output is `y[k]*(r*z[10] + t*z[11])`, indicating that only this intermediate value is of interest.

Figure 12. Example Computation

```
program_source_code:
/* Kernel 1 Hydro Fragment */
for ( k=0 ; k<n ; k++ ) [8]
     x[k] = q + y[k] * ( r*z[10] + t*z[11] );

input_data:
n=6

demanded_output:
y[k] * ( r*z[10] + t*z[11] )
```

The PDW for the source code shown above is shown below in Figure 13. Due to space constraints the β nodes for variables r, t, y, q and x are not shown. Likewise, a number of the true data dependence arcs are not shown. The primary purpose in including the PDW is to give credence to the calculation of the longest path discussed below.

Figure 13. PDW for Example Computation

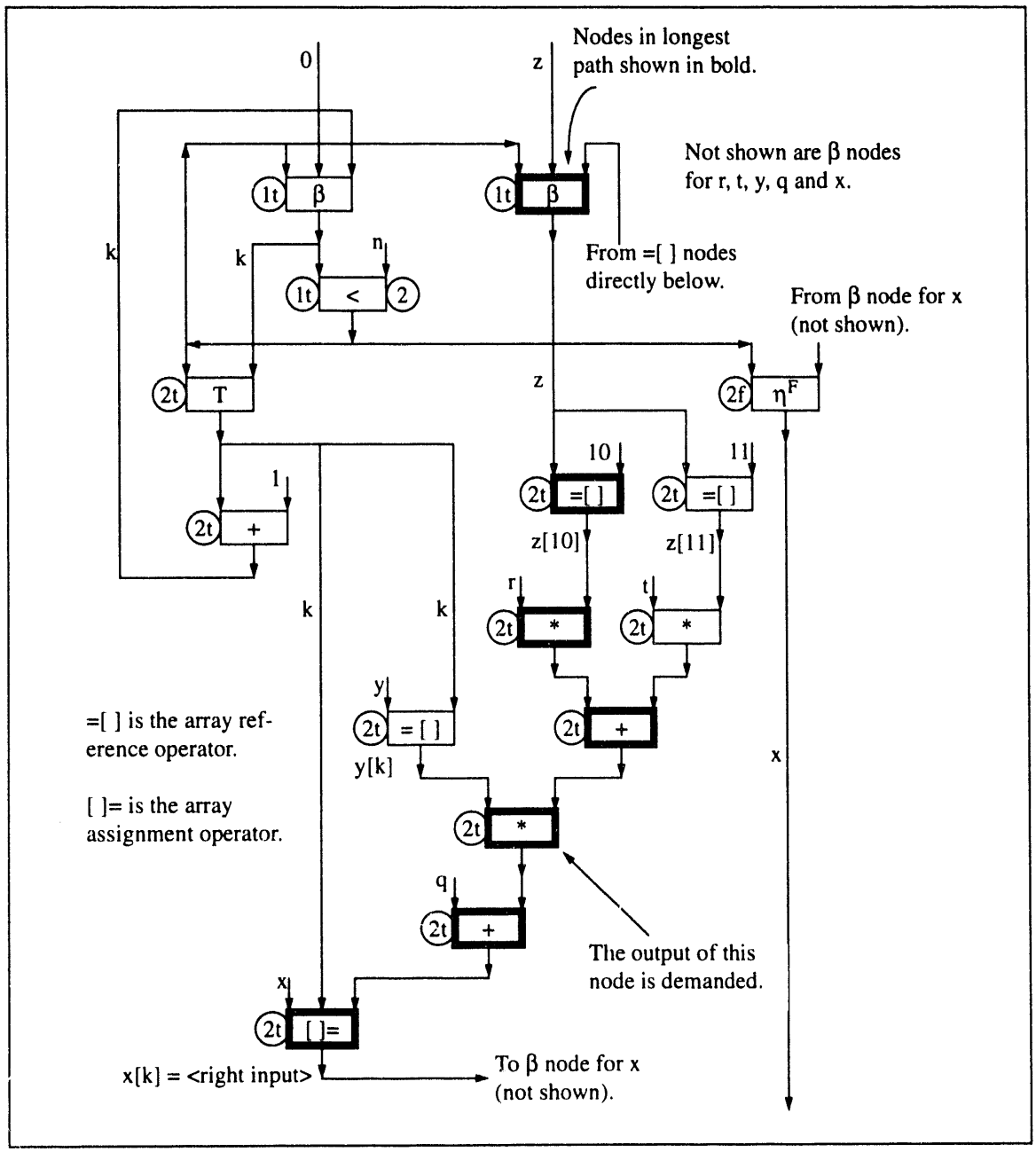


Table 13 below shows the performance (i.e., predicted execution time) of the nodes of the PDW shown above. To simplify the Tables, the common factors for the execution of any PDW node under each of the three execution disciplines is isolated and shown in Table 12 as "base" times.

Table 12. Base Time

	Control-driven	**Data-driven**	**Demand-driven**
Formula	[(2 * s_fetch) + s_store] * (1 - hit_rate) + pc	[(2 * s_fetch)+ s_store] * (1 - hit_rate) + (2*match)	[(2 * s_fetch)+ s_store] * (1 - hit_rate) + (2*match) + (2*demand)
Using values from Table 6	[(2*3) + 3] * (1-0.8) + 0	[(2*3) + 3] * (1-0.8) + (2*0.5)	[(2*3) + 3] * (1-0.8) + (2*0.5) + (2*0.5)
Base time	1.8 = $base_{control}$	2.8 = $base_{data}$	3.8 = $base_{demand}$

Table 13. PDW Node Execution Time

PDW Node	**Execution Time**		
	Control-driven	**Data-driven**	**Demand-driven**
β, T, η^F	pdw + $base_{control}$ = 1 + 1.8 = 2.8	pdw + $base_{data}$ = 1 + 2.8 = 3.8	pdw + $base_{demand}$ = 1 + 3.8 = 4.8
<	compare + $base_{control}$ = 2 + 1.8 = 3.8	compare + $base_{data}$ = 2 + 2.8 = 4.8	compare + $base_{demand}$ = 2 + 3.8 = 5.8
+ (Integer)	i_add + $base_{control}$ = 2 + 1.8 = 3.8	i_add + $base_{data}$ = 2 + 2.8 = 4.8	i_add + $base_{demand}$ = 2 + 3.8 = 5.8
+ (Float)	f_add + $base_{control}$ = 3 + 1.8 = 4.8	f_add + $base_{data}$ = 3 + 2.8 = 5.8	f_add + $base_{demand}$ = 3 + 3.8 = 6.8
=[ ]	a_fetch + $base_{control}$ = 5 + 1.8 = 6.8	a_fetch + $base_{data}$ = 5 + 2.8 = 7.8	a_fetch + $base_{demand}$ = 5 + 3.8 = 8.8
* (Float)	f_mult + $base_{control}$ = 8 + 1.8 = 9.8	f_mult+ $base_{data}$ = 8 + 2.8 = 10.8	f_mult + $base_{demand}$ = 8 + 3.8 = 11.8
[ ]=	a_store + $base_{control}$ = 5 + 1.8 = 6.8	a_store + $base_{data}$ = 5 + 2.8 = 7.8	a_store + $base_{demand}$ = 5 + 3.8 = 8.8

Predicted iteration time can now be calculated. The calculation is shown in Table 14 below.

Table 14. Iteration Time

	Control-driven	**Data-driven**	**Demand-driven**
Rule (see Table 10)	cumulative time for each node in the loop	longest path through the loop	longest path to the demanded node
(Listed at the right are the Nodes, and their number, that constitute iteration time)	7*β + + ("<") + add_{int} + T + 2*("=[ ]") + 3*$mult_{float}$ + 2*add_{float} + "[ ]="	β + + "=[ ]" + 2*$mult_{float}$ + 2*add_{float} + "[ ]="	β + + "=[ ]" + 2*$mult_{float}$ + add_{float}
Using Table 13 above	7*(2.8) + 3.8 + 3.8 + 2.8 + 2*(6.8) + 3*(9.8) + 2*(4.8) + 6.8	3.8 + 7.8 + 2*(10.8) + 2*(5.8) + 7.8	4.8 + 8.8 + 2*(11.8) + 6.8
Iteration time	= 89.4	= 52.6	= 44

And finally the heuristics can be completed. The value for "trip prediction" is provided in the source code of the computation (see Figure 12). Table 15 shows that calculation. (Since there are no branches in the source code of the computation, the four heuristics reduce to two.) The calculation predicts that the performance of demand-driven execution will be slightly (17%) better than data-driven and significantly (51%) better than control-driven execution.

Table 15. Heuristic Predictions

Heuristic	**Predicted Execution Time**		
	Control-driven	**Data-driven**	**Demand-driven**
Trip Predicted (TpBp or TpBa)	Iteration time * trip prediction = 89.4 * 8 = 715.2	Iteration time * trip prediction = 52.6 * 8 = 420.8	Iteration time * trip prediction = 44 * 8 = 352
Trip Unpredicted (TuBp or TuBa)	Iteration time * T_L = 89.4 * 5 = 447	Iteration time * T_L = 52.6 * 5 = 263	Iteration time * T_L = 44 * 5 = 220

9.0 Bibliography

[1] D.P. Agrawal, et al., "Evaluating the Performance of Multicomputer Configurations." Computer, May 1986, pp. 23-37.

[2] A.V. Aho, R. Sethi and J.D. Ullman, "Compilers: Principles, Techniques, and Tools." Addison-Wesley, Reading Massachusetts. 1986.

[3] Arvind, R.S. Nikhil, K.K. Pingali, "I-Structures: Data Structures for Parallel Computing." Computation Structures Group Memo, February 1987. Massachusetts Institute of Technology (MIT). Cambridge, Massachusetts.

[4] R.A. Ballance, A.B. Maccabe and K.J. Ottenstein, "The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages." In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pp. 257-271, June 1990.

[5] R.A. Ballance, A.B. Maccabe and K.J. Ottenstein, "Program Dependence Graphs for the Rest of Us." UNM Technical Report No. CS92-10, October 1992, University of New Mexico, Albuquerque, New Mexico.

[6] M. Beck, K.K. Pingali, "Static Scheduling for Dynamic Dataflow Machines." Journal of Parallel and Distributed Computing, 10, 279-288 (1990).

[7] L. Bic, "A process-oriented model for efficient execution of dataflow programs." Proceedings of the 7th International Conference on Distributed Computing. Berlin, West Germany. September 1987.

[8] R. Buehrer and K. Ekanadham, "Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution." IEEE Transactions on Computers, Vol. C-36, No. 12. December 1987, pp. 1515-1522.

[9] P.L. Campbell, K. Krishna and R.A. Ballance, "Refining and Defining the Program Dependence Web." UNM Technical Report No. CS93-6, March 31, 1993, University of New Mexico, Albuquerque, New Mexico. (Also appears as Sandia Report, SAND93-0961 • UC-905, May 1993. Sandia National Laboratories, Albuquerque, New Mexico.)

[10] D.C. Cann, "Compilation Techniques for High Performance Applicative Computation." Technical Report CS-89-108 (Ph.D. Dissertation), Colorado State University.

[11] P.J. Denning and G.B. Adams III, "Research Questions for Performance Analysis of Supercomputers." In Performance Evaluation of Supercomputers, J.L. Martin (editor), Elsevier Science Publishers B.V., (North-Holland), 1988. pp. 403-419.

[12] P. Evripidou, J-L. Gaudiot, "The USC decoupled multilevel data-flow execution model." Chapter 13 (pp. 347-379) of Advanced Topics in Data-Flow Computing. Prentice Hall, New Jersey. 1991.

[13] J. Ferrante, K. Ottenstein and J. Warren, "The Program Dependence Graph and its Use in Optimization." ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July 1987, pp. 319-349.

[14] G.R. Gao, "A flexible architecture model for hybrid data-flow and control-flow evaluation." Chapter 12 (pp. 327-346) of Advanced Topics in Data-Flow Computing. Prentice Hall, New Jersey. 1991.

[15] Ghosal, Dipak and L.N. Bhuyan, "Performance Evaluation of a Dataflow Architecture." IEEE Transactions on Computers, Vol, 39, No. 5, May 1990., pp. 615-627.

[16] M.J. Gonzalez, C.V. Ramamoorthy, "Program Suitability for Parallel Processing." IEEE Transactions on Computers, Vol. C-20, No. 6, June 1971, pp. 647-654.

[17] V.G. Grafe and J.E. Hoch, "The Epsilon-2 Multiprocessor System." Journal of Parallel and Distributed Computing, 10, 309-318 (1990).

[18] V.G. Grafe, J.E. Hoch and G.S. Davidson, "Eps'88: Combining the Best Features of von Neumann and Dataflow Computing." Sandia Report, SAND88-3128*UC-32, January 1989. Sandia National Laboratories, Albuquerque, New Mexico.

[19] J.R. Gurd, C.C. Kirkham and I. Watson, "The Manchester Prototype Dataflow Computer." Communications of the ACM, January 1985, pp. 34-52.

[20] J.L. Hennessy and D.A. Patterson, Computer Architecture: A Quantitive Approach. Morgan Kaufmann Publishers, Inc. San Mateo, CA. 1990.

[21] S. Horwitz, J. Prins and T. Reps, "On the Adequacy of Program Dependence Graphs for Representing Programs." In Proc. of the 15th ACM Symposium on Principles of Programming Languages, 1988, pp. 146-157.

[22] R.A. Ianucci, "A Dataflow/von Neumann Hybrid Architecture." MIT/LCS/TR-418, May 12, 1988. (Ph.D. dissertation.) Massachusetts Institute of Technology (MIT). Cambridge, Massachusetts.

[23] L.H. Jamieson, "Using Algorithm Characteristics to Evaluate Parallel Architectures." In Performance Evaluation of Supercomputers, J.L. Martin (editor), Elsevier Science Publishers B.V., (North-Holland), 1988. pp. 21-49.

[24] W. Kluge, "The Organization of Reduction, Data Flow, and Control Flow Systems." The MIT Press, Cambridge, Massachusetts. 1992. ISBN 0-262-61081-7.

[25] D.E. Knuth, "An Empirical Study of FORTRAN Programs." Software-Practice and Experience, 1, pp. 105-133.

[26] P.J. Koopman, P. Lee, "A Fresh Look at Combinator Graph Reduction (Or, Having a TIGRE by the Tail)." SIGPLAN Notices, vol/edit 24, no. 7, July 1989, pp. 110-119.

[27] K. Krishna, "Program Specialization via Partial Evaluation." Dissertation Proposal. University of New Mexico, Albuquerque, New Mexico. July 5, 1992.

[28] G.A. Mago, "A Network of Microprocessors to Execute Reduction Languages." International Journal of Computer and Information Sciences, Vol. 8, 1979. Part I is in No. 5, pp. 349-385; and Part II is in No. 6, pp. 435-471.

[29] F.H. McMahon, "The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range." Lawrence Livermore National Laboratory, Livermore, California, UCRL-53745, December 1986.

[30] R.S. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?" ACM SIGARCH Computer Architecture News (The 16th Annual International Symposium on Computer Architecture), Vol. 17, No. 3, May-June 1989, pp. 262-272.

[31] R.S. Nikhil, G.M. Papadapoulos and Arvind, "*T: A Multithreaded Massively Parallel Architecture." Computation Structures Group Memo 325-1, November, 15, 1991. MIT, Cambridge, MA.

[32] R.S. Nikhil and Arvind, "Programming in *Id*: A Parallel Programming Language." Unpublished manuscript. Copyright 1989, 1990, 1991, 1992 by The Massachusetts Institute of Technology.

[33] G.M. Papadopoulos, "Implementation of a General Purpose Dataflow Multiprocessor." Ph.D. Dissertation. MIT, December 1988. Massachusetts Institute of Technology (MIT). Cambridge, Massachusetts.

[34] L.M. Patnaik, R. Govindarajan, and N. S. Ramdoss, "Design and performance evaluation of EXMAN: An extended Manchester dataflow computer, IEEE Transactions on Computers, vol, C-35, no. 3, pp. 229-243, Mar. 1986.

[35] K. Pingali and Arvind, "Efficient Demand-Driven Evaluation. Part 1." ACM Transactions on Programming Languages and Systems, Vol. 7, No. 2, April 1985, pp. 311-333. and "Part 2" in Vol. 8, No. 1, January 1986, pp. 109-139.

[36] G. Ramalingam and T. Reps, "Semantics of Program Representation Graphs." Computer Sciences Technical Report #900, December 1989, University of Wisconsin-Madison.

[37] R.H. Saavedra-Barrera, A.J. Smith, E. and Miya, "Machine Characterization Based on an Abstract High-Level Language Machine." IEEE Transactions on Computers, Vol. 38, No. 12, December 1989, pp. 1659-1679.

[38] R.P. Selke, "A Rewriting Semantics for Program Dependence Graphs." In Proc. 16th ACM Symposium on Principles of Programming Languages, 1989, pp. 12-24.

[39] B.J. Smith, "A Pipelined, Shared Resource MIMD Computer." In Proceedings of the 1978 International Conference on Parallel Processing, pp. 6-8. 1978.

[40] Treleaven, Brownbridge, and Hopkins, "Data-Driven and Demand-Driven Computer Architecture." Computing Surveys, Vol. 14, No. 1, pp. 93- 143, March 1982.

[41] A.H. Veen and R. Born, "The RC Compiler for the DTN Dataflow Computer." Journal of Parallel Computing and Distributed Computing, Vol. 10, pp. 319-332. 1990.

[42] H.M. Vin, F. Berman and J.S. Mattson, "Efficient Data-Driven Evaluation: Theory and Implementation." Journal of Parallel and Distributed Computing, 10, 367-385 (1990).

[43] M. Wolfe, "Optimizing Supercompilers for Supercomputers." The MIT Press, Cambridge, Massachusetts. 1990. ISBN 0953-7767.

[44] B.P. Ziegler, "Theory of Modelling and Simulation." John Wiley & Sons. New York. 1976.

[45] H. Zima (with B. Chapman), "Supercompilers for Parallel and Vector Computers." ACM Press, New York, New York. 1991. ISBN 0-201-17560-6.

DISTRIBUTION:

1 R. E. Hollenbach
Charlotte Way
Livermore, CA 94550

1 Los Alamos National Laboratory
Attn: W. F. Hemsing, M-7
PO Box 1663
Los Alamos, NM 87545

3 University of New Mexico
Attn: R. A. Ballance
K. Krishna
A. B. Maccabe
Department of Computer Science
Albuquerque, NM 87131

1 8523-2 Central Technical Files
5 7141 Technical Library
1 7151 Technical Publications
10 7613-2 Document Processing
for DOE/OSTI
1 1900 D. L. Crawford
1 1901 R. E. Palmer
1 1902 N. R. Morse
1 1903 D. C. Jones
1 1904 A. R. Iacoletti
1 1905 G. Gutierrez
1 1906 R. C. Dougherty
1 1932 C. D. Brown
1 1952 P. W. Dean
1 1951-1 D. H. Ching
1 1952 R. E. Cline, Jr.
1 1954 M. O. Vahle
1 1955 D. M. Darsey
1 1955-1 J. P. Sena
1 1956 R. J. Pryor
10 1956 P. L. Campbell
1 1957 W. D. Swartz

END

DATE
FILMED
3/29/94