An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors

RAMÓN D. ACOSTA, MEMBER, IEEE, JACOB KJELSTRUP, AND H. C. TORNG, SENIOR MEMBER, IEEE

Abstract—**Processors with multiple functional units, such as CRAY-1, Cyber 205, and FPS 164, have been used for high-end scientific computation tasks. Much effort has been put into increasing the throughput of such systems. One critical consideration in their design is the identification and implementation of a suitable instruction issuing scheme. Existing approaches do not issue enough instructions per machine cycle to fully utilize the functional units and realize the high-performance level achievable with these powerful execution resources.**

In this paper, the dispatch stack (DS), an innovative instruction issuing approach designed to overcome this limitation, is presented. The DS enhances performance in these systems by employing dynamic code scheduling to permit 1) one or more instructions to be issued per machine cycle and 2) instructions to be issued nonsequentially.

The effectiveness of the DS has been evaluated with extensive simulation using the Livermore Loops. The simulation results, which cover various Instruction/Execution unit configurations, are presented here. The statistics obtained establish that instruction issuing with DS results in speedups ranging from 1.71 to 2.79 over serial dispatching schemes.

Index Terms—**Dispatch stack, dynamic instruction scheduling, instruction issuing, instruction unit, multiple functional unit processors, multiple instruction dispatching, processor performance enhancement.**

I. Introduction

PROCESSORS with multiple functional units (FU's), such as CRAY-1, Cyber 205, and FPS 164, have been configured for high-end scientific computation workloads. Due to the advent of very large scale integrated (VLSI) circuit technologies, it is likely that computer architects will adopt multiple FU architectures for a wider range of applications. Much effort is being put into improving the throughput of such "supercomputing" systems.

The *throughput* of a computer system is defined in this investigation as the number of instructions processed per unit time. The primary ways of enhancing throughput include: reducing the machine cycle time, reducing the memory access time, and increasing the concurrent processing of instructions. This investigation concerns the last approach.

Manuscript received March 20, 1985; revised November 22, 1985. This work was supported in part by a grant from IBM.

R. D. Acosta was with the School of Electrical Engineering, Cornell University, Ithaca, NY. He is now with Microelectronics and Computer Technology Corporation, Austin, TX 78759.

J. Kjelstrup was with the School of Electrical Engineering, Cornell University, Ithaca, NY. He is now with Bell Communications Research, West Long Branch, NJ 07764.

H. C. Torng is with the School of Electrical Engineering, Cornell University, Ithaca, NY 14853.

IEEE Log Number 8610076.

To obtain reasonably high throughput in these execution structures, it is essential to keep their functional units as busy as possible. Let the number of functional units in a computer system be n and the machine cycle time be c. The maximum instruction execution rate of such a system is n/c. This implies that each functional unit receives and executes one instruction per cycle, a feat not easily realized.

The desire to increase the FU utilization demands that each instruction be issued as soon as it is free of data and path constraints. In a system such as the CRAY-1 [22], this is not possible due to its policy of *sequentially issuing at most one instruction per cycle*. In a study of the CRAY-1S architecture, Srini and Asenjo [23] point out that this is a performance bottleneck in an otherwise well-balanced configuration. Thus, even though this processor is capable of concurrent execution of instructions on its multiple function units, performance is bounded by its serial dispatching scheme.

In other existing [7], [24], [27] and proposed [11], [33] systems, a situation analogous to that of the CRAY-1 exists in that instructions are issued to real or virtual functional units sequentially, according to their positions in the instruction stream, and at most one can be issued per machine cycle. Consequently, in all of these instruction issuing schemes the instruction execution rate cannot be higher than $1/c$.

In this paper, we present the formulation and evaluation of the dispatch stack (DS), an instruction issuing approach that provides dynamic scheduling of serial instruction streams. The proposed mechanism, originally conceived by Torng [28], [29], [30], offers significant improvements over existing or previously reported approaches in two main aspects: 1) one or more instructions can be issued per cycle and 2) instructions can be issued nonsequentially.

Also presented are the results of extensive simulation work with the Livermore Loops. For various configurations, the speedups achieved over serial dispatching schemes range from 1.71 to 2.79.

In the following section, the general architecture of multiple functional unit systems is described. In Section III, the motivation behind dynamic code scheduling is discussed. The DS is formulated and described in Section IV; and an example is given. A comparison to previous approaches, including Tomasulo's reservation stations and common data bus used in the IBM 360/91 [27], appears in Section V. Sections VI–IX contain a description of the evaluation methodology and the DS simulator. This is followed by the simulation results and their analysis, in Sections X and XI. Hardware system integration aspects are discussed in Section XII. In Section XIII, other issues, including treatment of conditional branches, static scheduling via compilation, and code optimization, are qualitatively considered. Conclusions are presented in Section XIV.

II. General Architecture

A multiple functional unit processor can generally be partitioned into two parts: an *instruction unit* (IU) and an *execution unit* (EU), shown in Fig. 1. The IU fetches instructions from *main memory* into an *instruction window*, decodes these instructions, and fetches operands, if necessary. Decoded instructions, together with their requisite operands, are sent to the EU. The instruction window serves as a buffer for prefetching and decoding instructions.

The EU consists of the *functional units* and a bank of *registers*, connected by a suitable *interconnection network*. The FU's are responsible for performing arithmetic/logical operations and can operate in parallel, providing the primary degree of concurrency in the processor. An additional level of concurrency can be incorporated into the FU's by implementing them as pipelined processors for scalar and/or vector operations [13], [20].

The registers serve as buffers between the "fast" functional units and the "slow" main memory. These registers supply operands to the functional units and receive results from them. They also load from and write into the main memory. This constitutes a register–register architecture, with memory being accessed via load/store operations.

The design of the interconnection network is often critical in determining multiple FU system performance. The types of networks can range from nonblocking, fully connected crossbar switches to dedicated buses. Many tradeoffs arise in choosing an appropriate interconnection structure, including throughput, delay, connectivity, size, and cost [34].

Variations to the general organization of Fig. 1 can be found in practice. One possibility is to have an independent interconnection network in the EU between main memory and the register bank. This can serve to reduce some of the contention on data transmissions between registers and FU's. Another possibility is to use separate program and data memories, or caches. These can eliminate potential memory conflicts since they are accessed independently by the IU and EU.

III. Instruction Scheduling

As suggested earlier, one key issue in improving performance in systems with multiple functional units is to schedule the execution of programs effectively. Code scheduling involves arranging instructions in order to minimize register dependencies and execution resource conflicts. The scheduling of instructions can be categorized as being *static* or *dynamic*.

A. Static Versus Dynamic Code Scheduling

Static code scheduling is often done in software at compile time. It entails having a highly optimizing compiler produce an

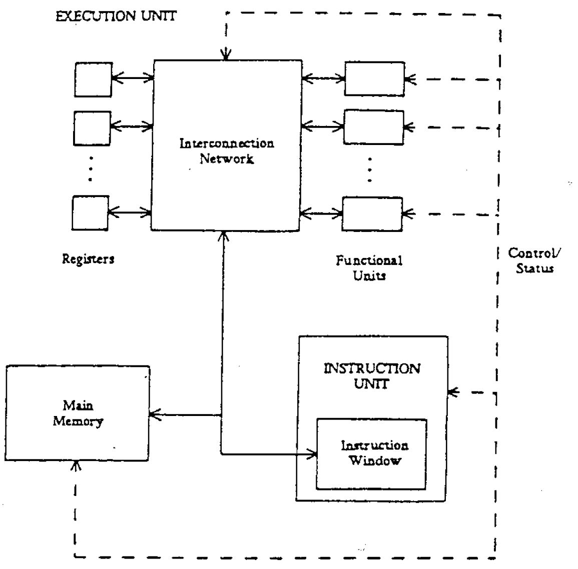


Fig. 1. Multiple functional unit processor—general architecture.

execution schedule that does not change while a program is running. Unfortunately, compilers capable of this optimization are slow and likely to be highly machine dependent. Furthermore, due to dynamic dependencies that cannot be resolved at the compiler level, it is usually not possible to statically generate optimal schedules.

Dynamic scheduling is usually a runtime activity performed in hardware on a window of instructions. As discussed in [33], dynamic scheduling carried out in hardware has the following advantages over static scheduling.

1) The machine performance is not dependent on the quality of the compiled code, relieving compilers and/or programmers of what is often a very difficult and burdensome task.

2) Dynamic dependencies, not available at compile time, can be uncovered. This has some of the characteristics of data flow scheduling approaches and is especially useful in branches and loops.

3) The execution schedule is dynamically generated with precise knowledge of the available execution resources, which is particularly important when the scheduling algorithm must adapt to variations in processor configurations due to component failures.

It should be noted that dynamic scheduling can also benefit from static scheduling techniques. For instance, a suitable allocation of registers might substantially improve the dynamic scheduling possibilities of a given instruction stream.

Yet there are disadvantages to dynamic code scheduling. It can lead to more complex hardware that is harder to design, debug, and maintain. It may also slow down a processor's instruction issuing phase. Despite these drawbacks, the advent of VLSI has resulted in low-cost implementations of very large and fast systems. In light of this, there has been a trend in recent years to move some software problems into hardware. Thus, dynamic code scheduling in instruction issuing mechanisms should be considered as a method for improving performance in future systems.

Within dynamic code scheduling, there is another division that can be made regarding instruction streams. Dynamic scheduling can be performed on the instructions as they appear in memory, constituting the *static instruction stream* [32]. This is the order of instructions as they are generated by a compiler. Alternately, dynamic scheduling can be conducted on the *dynamic instruction stream*, which is the sequential execution order of instructions in a compiled program. Both of these models have been proposed and implemented in concurrent dynamic scheduling systems.

The DS mechanism, described in Section IV, performs dynamic code scheduling on the dynamic instruction stream.

B. Instruction Data Dependencies

The criteria used to determine data dependencies for instruction scheduling purposes is now presented. The idea is to detect parallelism in a series of operations in an instruction window in the instruction unit of a processor. In general, the emphasis is on approaching optimality on a local level with respect to concurrently executable instructions. The approach discussed here is more thoroughly described by Keller in [11].

Throughout this work, the following instruction format is assumed:

$$i\colon OP,\ S1,\ S2,\ D \qquad (1)$$

where i is an instruction label, OP denotes a binary operation, $S1$ and $S2$ specify operand source registers, and D specifies the destination register. The semantics of such an instruction are such that its execution has the following effect.

$$i\colon D \leftarrow [S1]\ OP\ [S2]. \qquad (2)$$

The instruction format, specified in (1), is identical to that of the CRAY and Cyber machines. With registers D and $S1$ identical, the instruction takes on the same form as the instructions for the floating-point execution unit in the 360/91 system. It is also possible to think of $S1$, $S2$, and D as being vector registers, with OP specifying execution on a vector functional unit.

For instruction i, a domain and a range can be defined as follows:

$$\text{domain registers:}\quad \text{domain}(i) = \{S1,\ S2\}$$

$$\text{range registers:}\quad \text{range}(i) = \{D\}.$$

These registers can be thought of as *sources* and *sinks*, respectively, of instruction i. Register conflicts due to data dependencies can then be defined in the following way.

If i and j are two different operations, then conflict(i, j) = TRUE if, and only if, either

$$(1)\ \text{range}(i) \cap \text{domain}(j) \neq \emptyset$$

$$\text{or } (2)\ \text{range}(j) \cap \text{domain}(i) \neq \emptyset$$

$$\text{or } (3)\ \text{range}(i) \cap \text{range}(j) \neq \emptyset.$$

Otherwise, conflict(i, j) = FALSE.

The *conflict* relation can be used to partially order the operations in an instruction window according to their data dependencies. This, in turn, leads to the following local optimality criterion (for branch-free code).

Principle of Optimality: (Keller [11]) Whenever j is an operation corresponding to an instruction in the window, and there is no preceding operation i that is either being executed or is pending execution such that conflict(i, j) = TRUE, then j should be issued.

IV. The Dispatch Stack

An instruction window for the instruction unit of a conventional multiple functional unit processor is depicted in Fig. 2. The window is used as an instruction buffer for prefetching instructions from the dynamic instruction stream. It operates in a first-in-first-out (FIFO) sequential fashion, and each stack cell is implemented with a register. Hence, in a given cycle, the topmost instruction can be removed from the window and issued to the execution unit. As entries are vacated by instructions that have been issued, the other instructions in the buffer are "pushed" upward. Subsequently, new instructions are fetched from memory and appended to the bottom of the window.

As an illustration, the IU deposits the sequence of operations given below into an empty instruction window.

$$\begin{aligned} &I0\colon AD,\ R0,\ R1,\ R0\\ &I1\colon AD,\ R2,\ R3,\ R2\\ &I2\colon AD,\ R0,\ R2,\ R0\\ &I3\colon AD,\ R4,\ R5,\ R4\\ &I4\colon AD,\ R6,\ R7,\ R6\\ &I5\colon AD,\ R4,\ R6,\ R4\\ &I6\colon AD,\ R0,\ R4,\ R0. \end{aligned} \qquad (3)$$

In (3), $R0, R1, \cdots, R7$ denote registers, and AD stands for the "addition" operation. This program adds the contents of $R0$ through $R7$ and leaves the result in $R0$. After receiving this sequence, the window can be depicted with Fig. 3. Note that an "instruction tag" field is not necessary because of the strict sequential ordering maintained by the window.

The drawback of the instruction issuing mechanism employed in conventional instruction windows is that the IU examines only the instruction at the head of the dynamic stream; at best, only one instruction can be issued for every machine cycle. If the topmost instruction cannot be issued due to a violation of any one of the 3 conditions stated below.

Condition 1: Lack of a requisite functional unit.

Condition 2: Lack of requisite interconnection paths to transmit operands and/or results.

Condition 3: Data dependencies among instructions (in the sense of the conflict relation).

The instruction flow is stopped entirely. This situation exists in the CRAY-1 [22]. The proposed dispatch stack is aimed at relieving the bottleneck imposed by Condition 3.

The incorporation of reservation stations (i.e., virtual functional units) and the common data bus (CDB) scheme [27] reduces to a certain extent the constraints imposed by Conditions 1 and 3. Weiss and Smith have demonstrated that considerable speedup can be accomplished when this scheme

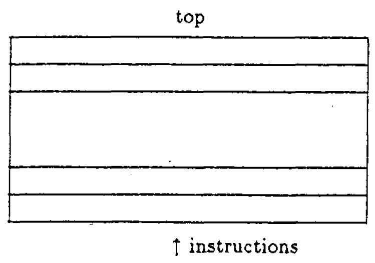


Fig. 2. Initial schematic diagram of an instruction window.

top

OP	S1	S2	D
AD	R0	R1	R0
AD	R2	R3	R2
AD	R0	R2	R0
AD	R4	R5	R4
AD	R6	R7	R6
AD	R4	R6	R4
AD	R0	R4	R0

Fig. 3. The sequence of instructions in (3) deposited into an instruction window.

is used in conjunction with the CRAY-1 scalar execution unit [33]. Nevertheless, at most one instruction is issued per machine cycle and these instructions are issued according to their positions in the instruction stream—sequentially.

The proposed DS strives to relieve multiple functional unit processors from this limitation. It accomplishes this by maintaining additional information for determining data dependencies among instructions.

A. α and β Fields

An instruction in an instruction window can be immediately processed with an available functional unit if it does not have data dependencies with any preceding instructions that have yet to be completed. Based on the conflict relation, the following three observations can be made to guide the instruction issuing process.

1) An instruction is data dependent upon a preceding, uncompleted instruction if one of its source registers appears as the destination register of the latter.

2) An instruction is data dependent upon a preceding, uncompleted instruction if its destination register appears as a source register of the latter.

3) An instruction is data dependent upon a preceding, uncompleted instruction if its destination register appears as the destination register of the latter.

For example, in (3), instruction $I2$ is data dependent upon $I0$. This is so because one of its source registers $R0$ is the destination register of $I0$. In other words $I2$ uses the result of $I0$ as an operand and, thus, must wait for the completion of $I0$ in order to begin executing. Instruction $I2$ is data dependent upon $I0$ in another sense: its destination register $R0$ is one of the source registers of $I0$. If $I2$ is issued and completed before $I0$ does, $I0$ may mistakenly use the result of $I2$ as one of its operands.

The above observations lead us to the formulation of the dispatch stack by enriching each entry in a conventional instruction window with data dependence information. The resultant fields are the following:

$$\text{Tag},\ OP,\ S1,\ \alpha(S1),\ S2,\ \alpha(S2),\ D,\ \alpha(D),\ \beta(D),\ \mathrm{I}^2. \tag{4}$$

The α and β dependence fields in each DS entry are defined as follows:

$\alpha(R)$ = the number of times that register R is designated as a destination register in preceding, uncompleted instructions.

$\beta(R)$ = the number of times that register R is designated as a source register in preceding, uncompleted instructions.

The I^2 field "summarizes" this dependence information, as will be explained in the following section.

The sequence of instructions in Fig. 3 is now represented in the DS of Fig. 4. In this illustration, instruction $I0$ is at the top of the stack and no instruction precedes it. Consequently,

$$\alpha(R0)=\alpha(R1)=\beta(R0)=0.$$

Instruction $I1$ is preceded by $I0$, but neither of its source or destination registers, $R2$ and $R3$, are used as a destination register by $I0$. Consequently,

$$\alpha(R2)=\alpha(R3)=0.$$

Furthermore, its destination register $R2$ is not employed as a source register by $I0$. Thus,

$$\beta(R2)=0.$$

For Instruction $I2$,

$$\alpha(R0)=\alpha(R2)=1$$

as $R0$ is the destination register for $I0$ and $R2$ is the destination register for $I1$. Also

$$\beta(R0)=1$$

since $R0$ is used as a source register in $I0$. Other α and β values can be similarly explained.

B. Instruction Issue Index

The issue index (I^2) for an instruction is defined as

$$\mathrm{I}^2=\alpha(S1)+\alpha(S2)+\alpha(D)+\beta(D) \tag{5}$$

Consequently, I^2 is the sum of all the dependencies for a given instruction entry in the DS. At each machine cycle, the DS is scanned. Any instruction with an I^2 value of 0 can be issued if an appropriate functional unit is available and if the processor is capable of doing so. It is important to note that in a given machine cycle, zero, one, or more than one instruction can be issued. Furthermore, instructions can be issued out of sequence. Various issuing modes based on this approach are discussed in Section VII-A.

Let us examine Fig. 4 again and, for the sake of simplicity, let us make the following assumptions.

Tag	OP	S1	α(S1)	S2	α(S2)	D	α(D)	β(D)	I^2
I0	AD	R0	0	R1	0	R0	0	0	0
I1	AD	R2	0	R3	0	R2	0	0	0
I2	AD	R0	1	R2	1	R0	1	1	4
I3	AD	R4	0	R5	0	R4	0	0	0
I4	AD	R6	0	R7	0	R6	0	0	0
I5	AD	R4	1	R6	1	R4	1	1	4
I6	AD	R0	2	R4	2	R0	2	2	8

Fig. 4. The sequence of instructions—with α, β, I^2 fields—in the DS.

Tag	OP	S1	α(S1)	S2	α(S2)	D	α(D)	β(D)	I^2
I1	AD	R2	0	R3	0	R2	0	0	0
I2	AD	R0	0	R2	1	R0	0	0	1
I3	AD	R4	0	R5	0	R4	0	0	0
I4	AD	R6	0	R7	0	R6	0	0	0
I5	AD	R4	1	R6	1	R4	1	1	4
I6	AD	R0	1	R4	2	R0	1	1	5

Fig. 5. The contents of the DS after decrementations and shifts, initiated by the completion of instruction *I*0.

Tag	OP	S1	α(S1)	S2	α(S2)	D	α(D)	β(D)	I^2
I2	AD	R0	0	R2	0	R0	0	0	0
I5	AD	R4	0	R6	0	R4	0	0	0
I6	AD	R0	1	R4	1	R0	1	1	4
Empty spaces ready for subsequent instructions									

Fig. 6. The contents of the DS after decrementations and shifts, initiated by the completion of instructions *I*0, *I*1, *I*3, and *I*4.

1) There are 4 "addition" functional units and these units are initially free.

2) It is possible to issue up to 4 instructions per cycle.

3) The operands are available at their designated registers.

4) Adequate data paths are available to transmit operands and results.

Instructions *I*0, *I*1, *I*3, and *I*4, all with an I^2-value of 0, are issued concurrently to the 4 free functional units.

Assuming that the register fields for each DS entry are *content addressable*, at the completion of an issued instruction, its destination register is used as a "key" to content address the *S*1, *S*2, and *D* fields of those instructions that follow it in the DS. Wherever there is a match, the appropriate α values are decremented by 1. Similarly, its source registers are used to content address the *D* fields of all subsequent instructions and decrement the appropriate β values. Note that for each completed instruction, "updating" is an $O(1)$ process because all of the entries in the DS are updated *concurrently*. Some aspects of the implementation of these content addressable fields are discussed in Section XII.

The DS update process is illustrated with Fig. 4. At the completion of Instruction *I*0, its destination register *R*0 is used to content address the *S*1, *S*2, and *D* fields of all instructions that follow *I*0 in the DS. The *S*1 and *D* fields of *I*2 and *I*6 match the *R*0 key and their corresponding $\alpha(S1)$ and $\alpha(D)$ fields are decremented by 1. Simultaneously, the source registers of *I*0, namely *R*0 and *R*1, are used to content address the *D* fields of all instructions that follow *I*0 in the DS. The *D* fields of *I*2 and *I*6 match the *R*0 key and their corresponding $\beta(D)$ fields are decremented by 1. Instruction *I*0 is removed from the DS to make room for subsequent instructions. The result of these reductions is presented in Fig. 5.

Similar (and perhaps concurrent) completions of *I*1, *I*3, and *I*4 reduce the contents of the DS to that shown in Fig. 6. Now instructions *I*2 and *I*5 can be issued and their completions reduce the I^2 value of *I*6 to 0.

With the assumptions given above, the issue and execution schedule of the sequence of instructions in (3) is

1st *I*0, *I*1, *I*3, *I*4

2nd *I*2, *I*5

3rd *I*6.

Note that this represents an optimal schedule. In fact, as long as there are enough functional units and the above assumptions hold, the dispatch stack issuing scheme is optimal in the sense of the principle of optimality of Section III-B.

V. Comparison to Previous Approaches

In Table I, a comparison of the DS with other dynamic scheduling approaches is presented. The CRAY-1 issuing scheme is also included, even though it does not really make use of dynamic scheduling techniques.

In this table, the dependence criteria column specifies the dependencies that are checked before issuing an instruction. RFU and VFU designate that an appropriate real or virtual functional unit, respectively, must be available to issue an instruction. Similarly, SRC and/or DST imply that in order for an instruction to be issued, it must be free of source and/or destination register data dependencies with uncompleted instructions.

Note that Tomasulo's reservation stations, together with the CDB [27], are actually a refinement of Thornton's scoreboard [24]. For example, since the CDC 6600 does not have reservations stations, the issuing of instructions is dependent on having the appropriate real FU's being free. In [11], Keller treats Tomasulo's algorithm in the more general and formal setting of lookahead processor architectures.

Major differences exist between Tomasulo's scheme for the 360/91 and the dispatch stack concurrent instruction issuing mechanism. In Tomasulo's algorithm, instructions are issued sequentially and at most one instruction is issued per cycle. However, there are no restrictions on issuing due to SRC or DST register dependencies since this information is kept in the reservation stations using tags. Instructions can be issued as long as there are reservation stations available. As can be seen, the dynamic code scheduling is accomplished by the CDB's forwarding of results at the execution level, and not at the instruction issuing level. Moreover, the data tags are used for maintaining the dependence information in a distributed fashion.

In the DS approach, the instruction dependence criterion is more restrictive, yet instructions can be issued in parallel and nonsequentially. Instruction dependencies are maintained by keeping usage counts for their source and destination registers in the instruction window. Thus, the data dependence information is centrally kept in the IU.

For an ideal, albeit unrealistic, processor, both the DS and

TABLE I
COMPARISON OF DYNAMIC SCHEDULING APPROACHES

Approach	Issue Order	Instruction Stream	Max Instructions Issued/Cycle	Dependence Criteria
Thornton's Scoreboard (CDC 6600)	sequential	dynamic	one	RFU DST
Tomasulo's Reservation Stations—Common Data Bus (IBM 360/91), Keller (1975)	sequential	dynamic	one	VFU
CRAY-1	sequential	dynamic	one	RFU SRC DST
Tjaden (1972) Wedig (1982) Uht (1985)	nonsequential	static	one or more	RFU SRC DST
Tjaden–Flynn (1970) Dispatch Stack	nonsequential	dynamic	one or more	RFU SRC DST

Tomasulo's algorithm conform to the Keller's principle of optimality. Consequently, the dynamic schedule generated by either scheme should be the same. When this is not the situation, the DS always guarantees issuing dependence-free instructions. On the other hand, the sequential nature of Tomasulo's algorithm makes it possible for data-dependent instructions to be using up execution resources while data independent instructions are held up in the instruction window.

This last point is illustrated if the schedule produced in the example of Section IV is compared to what would be obtained in a 360/91-like system. Since this scheme examines instructions sequentially, $I0$, $I1$, $I2$, and $I3$—being at the top of the instruction stream—are issued to the 4 free units (real or virtual). However, since at most one instruction can be issued per cycle, a delay of at least one cycle occurs between issuing each of them. In addition, $I2$, due to data dependencies, occupies a functional unit without actually being computed. That unit can be more advantageously employed to compute $I4$, as is done by the DS.

Tjaden [26], Wedig [32], and Uht [31] have explored the generation of dynamic schedules using static instruction streams. (The DS, as you might recall, uses dynamic instruction streams.) For ideal processors, their algorithms are also optimal. When used to schedule instructions for nonideal processors, these schemes attempt to use hardware bookkeeping mechanisms for consistent execution of the static streams. These can get complicated, especially when the entire static program does not fit into the instruction window.

Tjaden and Flynn [25] have suggested a look-ahead scheme for detecting independent instructions dynamically and executing them concurrently. Their approach relies on reducing dependencies by means of a register renaming algorithm. One drawback of their scheme is that $O(n^2)$ comparison hardware is required to simultaneously decode n instructions in a window.

It should be noted that since the DS issuing mechanism is independent of scheduling at the execution level, there is no reason why a more complex dynamic scheduling scheme at the execution level cannot be used. For example, it is possible for both the nonsequential, parallel issuing capabilities found in the DS and the freedom from register and FU dependencies offered by Tomasulo's reservation stations to coexist in a high-performance system.[1]

VI. Evaluation Methodology

The effectiveness of the DS is now evaluated by analyzing various systems that employ it to issue instructions. This is accomplished via simulation techniques. The simulator is written in *C* [12] and runs under the Unix™ operating system on a VAX®-11/780. It allows characterization of a system with a DS by providing programmable instruction and execution unit configurations. A more complete description of the simulator and the simulation results appears in [1].

A block diagram of the simulator appears in Fig. 7 (note the similarities with Fig. 1). The input to the system conists of an assembly language program and parameter settings to configure the instruction and execution units. The IU and EU are attached to a READ-only *Program Memory* and a READ/WRITE *Data Memory*, respectively. The simulator output includes statistics describing the number of instructions executed and cycles consumed, instruction issuing information, instruction execution information, and resource utilization.

The general approach used in obtaining statistics for the DS evaluation is to:

1) Choose an appropriate set of benchmark programs and hand-compile them into the simulator assembly language.
2) Run simulations to gather statistics for various IU and EU configurations.
3) Tabulate, plot, and analyze the results.

[1] Virtual FU's and the CDB can also be used in conjunction with other schemes in Table I in order to relax the FU/SRC/DST issuing restrictions. An analysis of such a modification for the CRAY-1 appears in [33].
™ UNIX is a trademark of Bell Laboratories.
® Vax is a registered trademark of Digital Equipment Corporation.

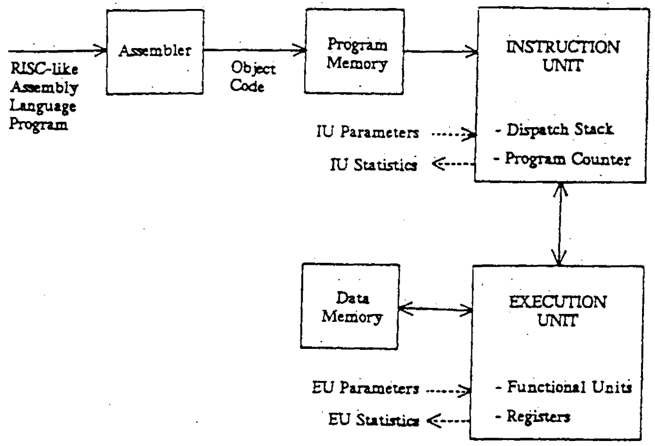


Fig. 7. Simulator block diagram.

VII. Instruction Unit Simulator

The main parameters of the IU are the size of the dispatch stack, the number of banks in the program memory, and the issue mode. The system can fetch as many instructions as there are memory banks within one cycle. These are placed in the DS and subsequently sent to the EU according to the issue mode. Note that while instructions are executing, they also remain in the DS to insure that dependencies are preserved. When instructions finish executing, they are returned from the EU to the IU and removed from the DS window.

A. Instruction Issue Modes

Using the simulator, statistics for studying and comparing various instruction issuing mechanisms are obtained. There are four basic modes in which the IU issues instructions.

Uniprocessor: Sequentially issues at most one instruction per cycle. The IU must wait until an instruction completes execution to issue the next instruction.

CRAY-1: Operates as a conventional instruction window: it sequentially issues a new instruction every cycle as long as it is not dependent on instructions already executing and the required execution resources are available.

n-Parallel: Up to n instructions can be dispatched per cycle as long as they are not dependent on each other or on instructions already issued and executing, and the required execution resources are available. These instructions do not have to appear sequentially in the DS.

Fully Parallel: This is a generalization of the n-parallel scheme in which as many instructions as possible (up to the size of the DS) can be issued in one cycle.

The n-parallel and the fully parallel modes implement the dispatch stack parallel instruction issuing mechanism. These are also the only ones that perform dynamic code scheduling since instructions are not forced to be issued sequentially. Both the DS n-parallel and fully parallel issue modes can benefit from static scheduling specified by the compiler and/or programmer. Static code scheduling is the only scheduling method that can be used to increase performance in the CRAY-1 issuing mode.

B. Instruction Set

The instruction set chosen for the simulator is modeled after the Berkeley reduced instruction set computer (RISC) [18]. This provides instructions in the form of quadruples such as

$$OP,\ S1,\ S2,\ D \tag{6}$$

(see Section II). One addition to these fields is a bit in each instruction that indicates whether the condition codes should be set by it. The instruction set is appropriate for a register–register architecture in which memory can only be assessed via load/store operations and data can only be operated on while it is in registers. This conforms with the structure depicted in Fig. 2.

It should be pointed out that branch instructions are never sent to the EU. Whenever a conditional branch is encountered in the instruction stream, the IU stops fetching new instructions. It waits until all instructions in the DS that set the condition codes are executed. Then the branch outcome is determined by the IU based on the condition codes. Subsequently, the program counter is loaded and new instructions are fetched from the appropriate stream.

VIII. Execution Unit Simulator

The EU can be configured in a variety of ways by means of the appropriate parameters. These include the number of registers and FU's, memory and bus delays, number of cycles per operation, and FU types.

A general block diagram of the multiple FU execution unit organization being simulated appears in Fig. 8. Note that four types of pipelined functional units are available. The interconnection network is modeled as a set of buses between registers and FU's. The global data memory, which looks like another FU, is also connected to the registers by the interconnection network.

This very general EU model was chosen over modeling a particular EU, such as the CRAY-1 because it provides a more generic framework for evaluating instruction issuing schemes. Since the issuing mechanism used by the IU is independent of the EU, the system performance can be examined with respect to various EU configurations. For this evaluation, the following EU's are used (see Table II for a summary of specific parameter values).

Paracomputer (PARA): This is an ideal machine with potentially an infinite number of FU's (i.e., as many as necessary in any given cycle) capable of performing any operation in one cycle [10].

Crossbar Machine (XBAR): This machine provides a fixed number of nonpipelined FU's. The registers are connected globally to the FU's by means of an unlimited number of buses that function as a nonblocking crossbar interconnection network. There is a delay associated with accessing memory.

Eight-Bus Machine (EBUS): Like XBAR, this EU contains a fixed number of nonpipelined FU's. However, the FU's are connected to the registers by a fixed number of buses. The use of these buses results in transmission delays.

Four-Pipe Machine (FPIPE): For this machine, there is

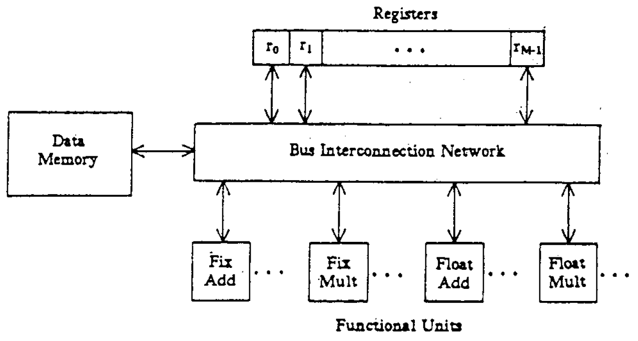


Fig. 8. Architecture of the execution unit simulator.

TABLE II
EXECUTION UNIT PARAMETERS USED IN SIMULATIONS

Parameter	PARA	XBAR	EBUS	FPIPE
Fixed Add FUs	-	2	2	1
- Stages	1	1	1	2
- Delay/Stage	1	1	1	1
Fixed Mult FUs	-	2	2	1
- Stages	1	1	1	2
- Delay/Stage	1	1	1	1
Float Add FUs	-	2	2	1
- Stages	1	1	1	3
- Delay/Stage	1	2	2	1
Float Mult FUs	-	2	2	1
- Stages	1	1	1	4
- Delay/Stage	1	3	3	1
Registers	64	64	64	64
Number of Buses	-	-	8	-
- Delay/Bus	0	0	1	1
Memory Delay	0	1	1	1

Notes:
1) A "-" means that an unlimited (i.e., as many as are needed in a given cycle) number of resources of the given type, are available.
2) Delays are given in number of cycles.

only one FU of each type. Each of them operates in a pipelined fashion. The interconnection network is treated as another pipeline stage.

One of the basic assumptions in the EU is that the memory access delay is fixed, i.e., there are never any memory bank conflicts. This assumption was made to simplify the simulator. Note, however, that a separate data memory is being used. This data memory may have many banks and can be accessed in parallel with the program memory. Consequently, there would probably not be many memory conflicts anyway and the assumption seems justified.

The FU's being modeled are all scalar. Hence, the results are for performance with respect to scalar operations. Studying scalar performance is important since it can often limit vector performance in supercomputers. Nevertheless, because of the IU/EU independence alluded to earlier, the DS can easily be extended to operate with vector registers and instructions (in either pipelined processors or processor arrays).

For example, if a system such as the CRAY-1 is equipped with a DS, data dependencies among *nonsequential* vector instructions can be determined. This enables those that are free of dependencies to be activated simultaneously. Thus, the FU utilization is increased because the instruction flow to its pipelined vector functional units is not halted by sequential dependencies.

IX. Benchmarks

The benchmark programs measured are the computation intensive Lawrence Livermore Loops [17], [21]. They represent typical inner loops or small routines found in numerical scientific programs. In Table III, the benchmarks are described and their static and dynamic instruction counts are given.

The input to the simulator is obtained by hand-compiling the programs' Fortran source code into the RISC-like Assembly language used in the system. In doing this, the register allocation is crucial since the DS uses register dependencies among instructions as the criterion for issuing them. Register allocation and code scheduling are intimately related issues for increasing the performance in high-performance parallel computers. Thus, assigning registers for such things as loads/stores, calculating array indexes, and performing arithmetic computations must be accomplished in such a way as to exploit the inherent parallelism of the program. Techniques analogous to those of tree height reduction [5], [14], [15], used for extracting parallelism from arithmetic expressions and other program constructs, can be employed within a compiler for register allocation.

In hand-compiling the benchmarks, a "dumb" compiler has been assumed. Such a compiler, although being simple, fast, and reliable, is not capable of improving the code it generates using complex optimizations [2], [3], [9]. For instance, redundant subexpressions and redundant loads are not removed. Besides performing the register allocation, the only other optimization being used is to put loop indexes in registers. In typical Fortran progamming environments, compilers can easily do this optimization.

X. Measurements and Results

The simulation measurements using the Livermore Loops are now presented. These results are analyzed in Section XI. The key parameters are the following:

1) instruction issuing mode,
2) dispatch stack size,
3) fetch count (number of program memory banks), and
4) execution unit configuration.

To reduce the volume of data, the statistics reflect totals or means for all of the benchmarks.

The following notation is introduced for describing multiple FU processor and DS configurations.

$P[c,\ \mathrm{DS}[s{:}f]]$ = multiple FU processor using EU configuration c and an IU dispatch stack with a window of size s and a fetch count of f.

When the DS size or the fetch count is unlimited in magnitude, "inf" is used to designate their potentially infinite size. The following abbreviations are used for designating the issuing modes.

U = uniprocessor

C = CRAY-1

$1P$ = 1-parallel

TABLE III
LIVERMORE BENCHMARKS USED IN SIMULATIONS

Name	Description	Static Count	Dynamic Count
loop1	Hydro Excerpt	19	6802
loop2	Inner Product (5 terms at a time)	37	6604
loop3	Inner Product	13	9004
loop4	Banded Linear Equations	30	7646
loop5	Tri-diagonal Elimination, Below Diagonal	31	9659
loop6	Tri-diagonal Elimination, Above Diagonal	47	14987
loop7	Equation Of State Excerpt	45	5162
loop8	PDE Integration	235	9014
loop9	Integrate Predictors	75	7302
loop10	Difference Predictors	130	12802
loop11	First Sum	13	6999
loop12	First Difference	10	7994
loop13	2-D Particle Pusher	134	16898
loop14	1-D Particle Pusher	63	9152
total		882	130025

$2P = 2$-parallel

$4P = 4$-parallel

$FP =$ fully parallel.

A. Throughput

Throughput is the primary measure of performance used here. If

$$\left\{\begin{array}{l} \text{Idynamic}(t) = \text{dynamic number of instructions of program } t \\ \text{Texecution}(t : A) = \text{execution time, in cycles, of program } t \text{ on machine } A \end{array}\right.$$

then the throughput of executing t on machine A is calculated as follows:

$$\text{throughput}(t : A) = \frac{\text{Idynamic}(t)}{\text{Texecution}(t : A)}. \tag{7}$$

As described in Section I, in a real machine, performance is intimately related to the actual cycle time. Yet cycle time is often associated more with technology than machine architecture. In this vein, the simulator simply treats the cycle time as one time step.

Plots of throughput versus DS size appear in Figs. 9–12 for the four EU's described in Section VIII. To isolate the window size effect on performance, the fetch count is unlimited, although it is actually bounded by the DS size.

In these results, the throughputs are 1.0 when the U, C, and $1P$ issuing modes are used on PARA. This is to be expected since these modes can only issue one instruction at a time and PARA executes all instructions in one cycle. Variations in these modes can be seen in the nonideal machines. This illustrates the reason for measuring the $1P$ mode: although it can only issue at most one instruction per cycle, it is capable of doing this nonsequentially and, consequently, benefits from having a larger DS size. The throughputs for the U and C modes, on the other hand, remain constant regardless of window size.

Except for the FP mode, the performance of the parallel issuing modes levels off at about a DS size = 16. On EBUS and FPIPE, both the $2P$ and $4P$ modes do almost as good as

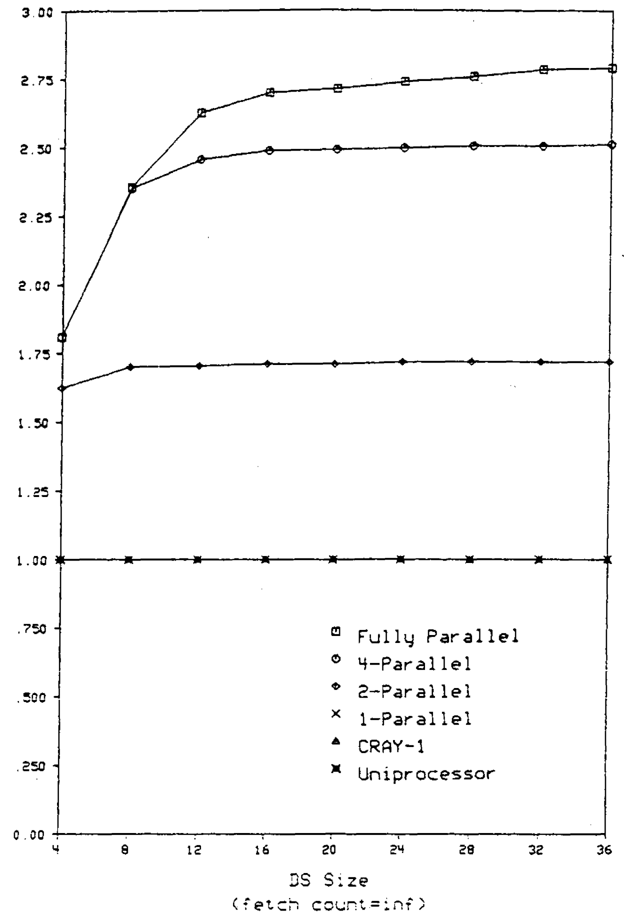


Fig. 9. Throughput versus DS Size—PARA.

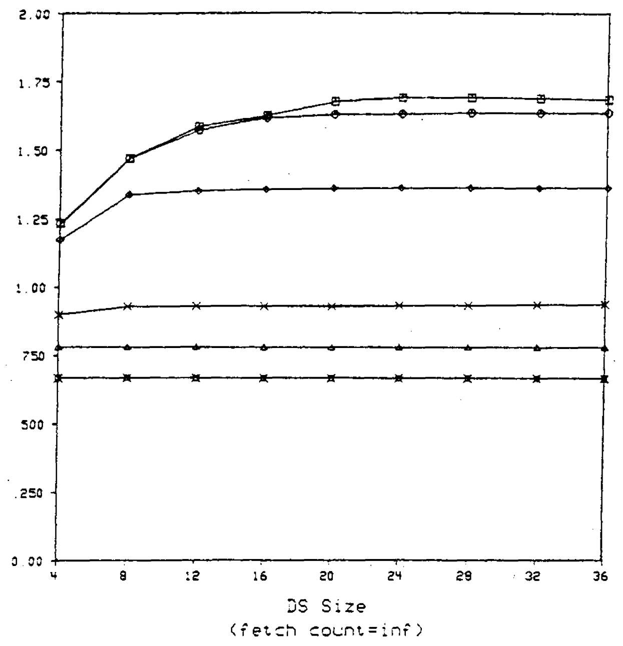


Fig. 10. Throughput versus DS Size—XBAR.

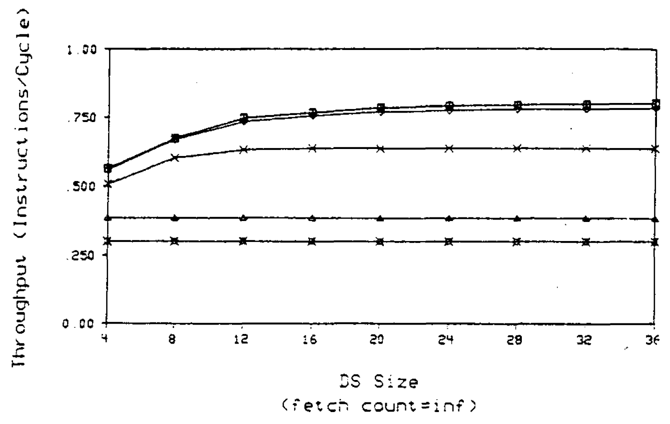


Fig. 11. Throughput versus DS Size—EBUS.

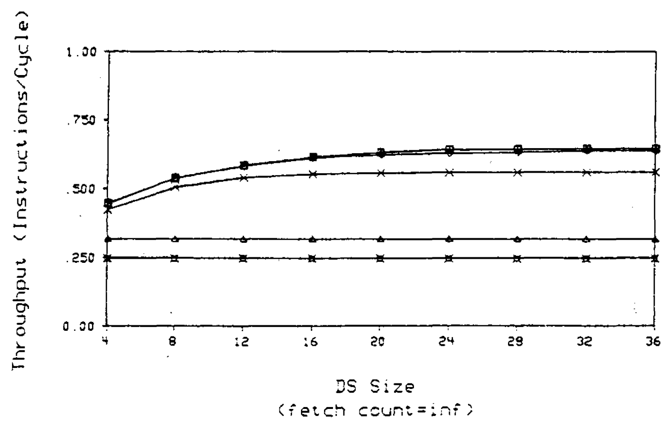


Fig. 12. Throughput versus DS Size—FPIPE.

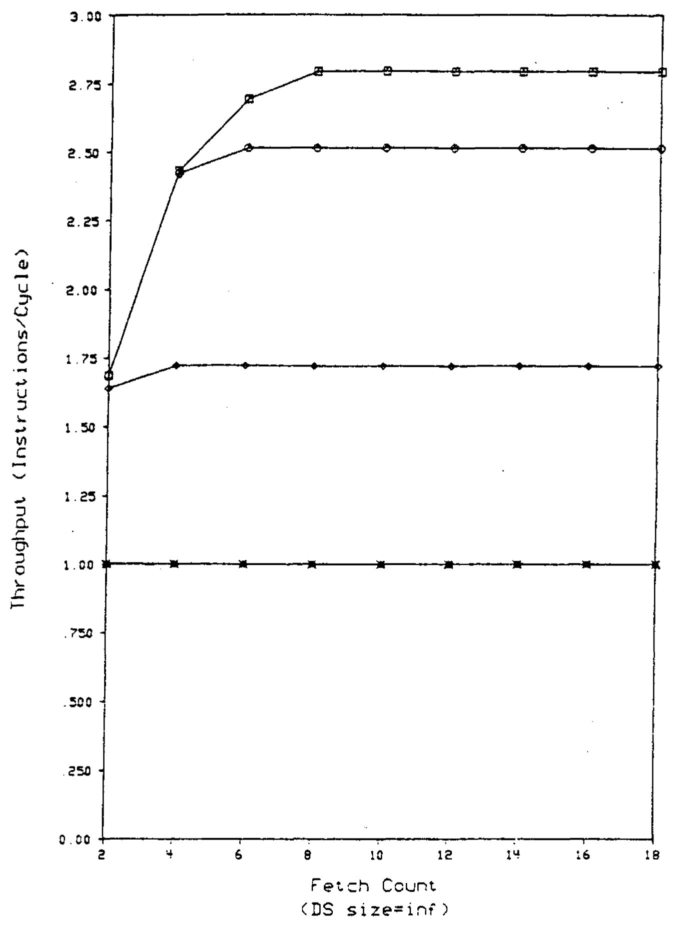


Fig. 13. Throughput versus fetch count—PARA.

the *FP* mode. In general, as the EU gets "less ideal," both the effect of DS size on performance and the advantage of having a lot of parallelism in the issuing mechanism decrease.

To examine the effects of the number of instructions fetched per cycle, plots for throughput vs. fetch count are presented in Figs. 13 through 16. Here, the DS window is unlimited in size.

The fetch count does not affect performance for the *U*, *C*, and 1*P* modes on any of the machines. On PARA, the throughputs level off at fetch counts of 4, 6, and 8 for the 2*P*, 4*P*, and *FP* modes. The fetch count effect on throughput decreases as the EU gets less ideal; on EBUS and FPIPE, throughput variations are rather minimal.

Based on the above graphs, three DS configurations are used to illustrate performance results throughout the rest of these measurements: DS[16:4], DS[32:8], and DS[inf:inf]. In Table IV, these are used in conjunction with the various EU types to obtain mean throughput statistics.

The effects of DS size and fetch count are only noteworthy for the 4*P* and *FP* modes on PARA. For EBUS and FPIPE, the 2*P* issuing mode is almost as good as the 4*P* and *FP* modes, these last two being equal in performance. Also, on these machines, the 2*P*, 4*P*, and *FP* modes perform about twice as good as the *C* issuing mode. The trend seems to be that the relative improvements of the 1*P* and 2*P* modes increase as the machine gets less ideal.

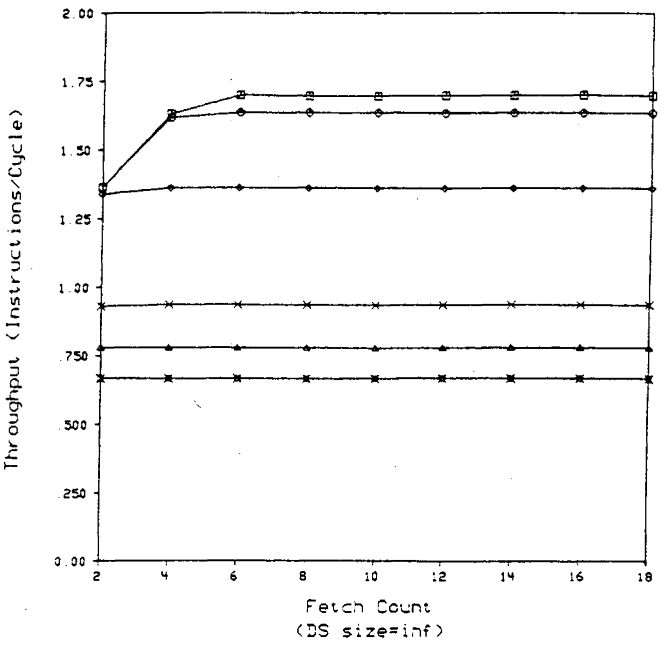


Fig. 14. Throughput versus fetch count—XBAR.

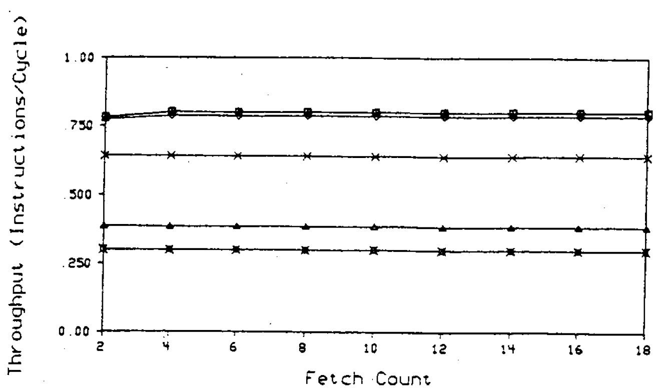


Fig. 15. Throughput versus fetch count—EBUS.

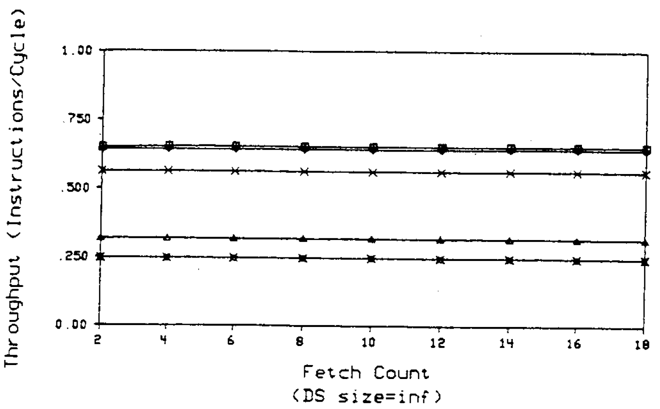


Fig. 16. Throughput versus fetch count—FPIPE.

TABLE IV
THROUGHPUT (INSTRUCTIONS/CYCLE)

Configuration	U	C	1P	2P	4P	FP
P[PARA, DS[16:4]]	1.00	1.00	1.00	1.71	2.40	2.40
P[PARA, DS[32:8]]	1.00	1.00	1.00	1.72	2.51	2.78
P[PARA, DS[inf:inf]]	1.00	1.00	1.00	1.72	2.51	2.79
P[XBAR, DS[16:4]]	0.67	0.78	0.93	1.36	1.60	1.62
P[XBAR, DS[32:8]]	0.67	0.78	0.94	1.36	1.64	1.68
P[XBAR, DS[inf:inf]]	0.67	0.78	0.94	1.36	1.64	1.70
P[EBUS, DS[16:4]]	0.30	0.39	0.64	0.76	0.77	0.77
P[EBUS, DS[32:8]]	0.30	0.39	0.64	0.78	0.80	0.80
P[EBUS, DS[inf:inf]]	0.30	0.39	0.64	0.79	0.80	0.80
P[FPIPE, DS[16:4]]	0.25	0.32	0.55	0.61	0.62	0.62
P[FPIPE, DS[32:8]]	0.25	0.32	0.56	0.64	0.65	0.65
P[FPIPE, DS[inf:inf]]	0.25	0.32	0.56	0.64	0.65	0.65

B. Speedup

The speedup is a relative measure of performance between two computers. Specifically, the speedup of machine A compared to machine B for executing a program t is

$$\text{speedup}(t:A:B)=\frac{\text{Texecution}(t:B)}{\text{Texecution}(t:A)} \tag{8}$$

Using (7), if Idynamic(t) is assumed constant for both machines A and B, the speedup can be calculated from throughput values as follows:

$$\text{speedup}(t:A:B)=\frac{\text{throughput}(t:A)}{\text{throughput}(t:B)} \tag{9}$$

In Table V, speedups relative to the Uniprocessor issuing mode corresponding to Table IV are shown. The greatest speedup observed is on PARA using DS[inf:inf] and the *FP* issuing mode. On EBUS and FPIPE, the speedups for all the parallel modes are greater than 2.0. This is considerably better than the *C* mode. Across all machines, only small variations in speedups for the 4*P* and *FP* issuing schemes are observed. This is not the case of the 1*P* and 2*P* modes. As in the throughput statistics, their improvement, relative to the 4*P* and *FP* modes, increases as the ideality of the EU decreases.

TABLE V
SPEEDUP (COMPARED TO UNIPROCESSOR)

Configuration	U	C	1P	2P	4P	FP
P[PARA, DS[16:4]]	1.00	1.00	1.00	1.71	2.40	2.40
P[PARA, DS[32:8]]	1.00	1.00	1.00	1.72	2.51	2.78
P[PARA, DS[inf:inf]]	1.00	1.00	1.00	1.72	2.51	2.79
P[XBAR, DS[16:4]]	1.00	1.16	1.39	2.03	2.39	2.42
P[XBAR, DS[32:8]]	1.00	1.16	1.40	2.03	2.45	2.51
P[XBAR, DS[inf:inf]]	1.00	1.16	1.40	2.03	2.45	2.54
P[EBUS, DS[16:4]]	1.00	1.30	2.13	2.53	2.57	2.57
P[EBUS, DS[32:8]]	1.00	1.30	2.13	2.60	2.67	2.67
P[EBUS, DS[inf:inf]]	1.00	1.30	2.13	2.63	2.67	2.67
P[FPIPE, DS[16:4]]	1.00	1.28	2.20	2.44	2.48	2.48
P[FPIPE, DS[32:8]]	1.00	1.28	2.24	2.56	2.60	2.60
P[FPIPE, DS[inf:inf]]	1.00	1.28	2.24	2.56	2.60	2.60

C. Instructions Issued/Cycle, Efficiency

The number of instructions issued/cycle can be used to examine the efficiency of an instruction issuing mechanism. This statistic is also dependent on the inherent parallelism of the benchmark programs. Its value is the total number of instructions issued for execution divided by the total number of system cycles. The results given below are for average, rather than "peak," instruction issuing activity.

Table VI gives instructions issued/cycle measurements. Since branches (and other miscellaneous instructions, such as "halt") are never dispatched to the EU, these measurements are slightly lower than the corresponding throughputs. One of the main characteristics of these results for the 1*P*, 2*P*, 4*P*, and *FP* modes is that as the power of the instruction issuing mechanism increases, its capabilities for issuing instructions in parallel are used less, on the average.

The efficiency of a given instruction issuing mechanism is the instructions issued/cycle divided by the maximum number of instructions the mechanism is capable of issuing. Efficiency values are tabulated in Table VII. Measurements for the *FP* mode using DS[inf:inf] are omitted since this mode's issuing capabilities vary with DS size.

As observed above, in the parallel issuing modes, the issuing efficiency decreases as the concurrent issuing capabilities of the issuing mode increase. Note that even though the 4*P* and *FP* modes achieve much higher throughputs than the *C* mode, they are considerably less efficient. In general, the most efficient of all schemes is the 1*P* mode.

D. FU and Bus Utilization

Average execution resource utilization is now considered. This is simply the percentage of cycles in which a particular resource is occupied. Utilizations give an indication of how well a processor's IU and EU are configured for executing specific workloads.

Functional unit utilizations are shown in Table VIII. Statistics for PARA are not given since it has an unlimited

TABLE VI
INSTRUCTIONS ISSUED/CYCLE

Configuration	U	C	1P	2P	4P	FP
P[PARA, DS[16:4]]	0.95	0.95	0.95	1.64	2.31	2.31
P[PARA, DS[32:8]]	0.95	0.95	0.95	1.65	2.41	2.69
P[PARA, DS[inf:inf]]	0.95	0.95	0.95	1.65	2.42	2.70
P[XBAR, DS[16:4]]	0.64	0.75	0.89	1.30	1.55	1.56
P[XBAR, DS[32:8]]	0.64	0.75	0.89	1.31	1.58	1.63
P[XBAR, DS[inf:inf]]	0.64	0.75	0.89	1.31	1.58	1.64
P[EBUS, DS[16:4]]	0.29	0.37	0.61	0.73	0.74	0.74
P[EBUS, DS[32:8]]	0.29	0.37	0.61	0.75	0.77	0.77
P[EBUS, DS[inf:inf]]	0.29	0.37	0.62	0.76	0.78	0.78
P[FPIPE, DS[16:4]]	0.24	0.30	0.53	0.59	0.60	0.60
P[FPIPE, DS[32:8]]	0.24	0.30	0.54	0.62	0.63	0.63
P[FPIPE, DS[inf:inf]]	0.24	0.30	0.54	0.62	0.63	0.63

TABLE VII
INSTRUCTION ISSUING EFFICIENCY

Configuration	U	C	1P	2P	4P	FP
P[PARA, DS[16:4]]	0.95	0.95	0.95	0.82	0.58	0.14
P[PARA, DS[32:8]]	0.95	0.95	0.95	0.83	0.60	0.08
P[PARA, DS[inf:inf]]	0.95	0.95	0.95	0.83	0.61	-
P[XBAR, DS[16:4]]	0.64	0.75	0.89	0.65	0.39	0.10
P[XBAR, DS[32:8]]	0.64	0.75	0.89	0.66	0.40	0.02
P[XBAR, DS[inf:inf]]	0.64	0.75	0.89	0.66	0.40	-
P[EBUS, DS[16:4]]	0.29	0.37	0.61	0.37	0.18	0.05
P[EBUS, DS[32:8]]	0.29	0.37	0.61	0.38	0.19	0.02
P[EBUS, DS[inf:inf]]	0.29	0.37	0.62	0.38	0.20	-
P[FPIPE, DS[16:4]]	0.24	0.30	0.53	0.30	0.15	0.04
P[FPIPE, DS[32:8]]	0.24	0.30	0.54	0.31	0.16	0.02
P[FPIPE, DS[inf:inf]]	0.24	0.30	0.54	0.31	0.16	-

TABLE VIII
FU UTILIZATION (IN PERCENT)

Configuration	U	C	1P	2P	4P	FP
P[XBAR, DS[16:4]]	8.24	9.72	11.72	16.91	20.31	20.48
P[XBAR, DS[32:8]]	8.24	9.72	11.78	16.98	20.70	21.32
P[XBAR, DS[inf:inf]]	8.24	9.72	11.80	16.98	20.70	21.49
P[EBUS, DS[16:4]]	3.71	4.80	7.99	9.61	9.74	9.74
P[EBUS, DS[32:8]]	3.71	4.80	8.00	9.91	10.13	10.14
P[EBUS, DS[inf:inf]]	3.71	4.80	8.02	9.99	10.20	10.20
P[FPIPE, DS[16:4]]	3.86	4.99	8.77	9.79	9.89	9.89
P[FPIPE, DS[32:8]]	3.86	4.99	8.86	10.20	10.36	10.36
P[FPIPE, DS[inf:inf]]	3.86	4.99	8.86	10.23	10.43	10.43

TABLE IX
BUS UTILIZATION (IN PERCENT)

Configuration	U	C	1P	2P	4P	FP
P[EBUS, DS[16:4]]	11.77	15.15	25.10	29.99	30.42	30.42
P[EBUS, DS[32:8]]	11.77	15.15	25.15	30.87	31.33	31.34
P[EBUS, DS[inf:inf]]	11.77	15.15	25.23	31.17	31.34	31.34

number of FU's. The highest utilizations are observed in XBAR, probably because the ideal interconnection network reduces instruction dispatching delays. Although utilizations for FPIPE are better than for EBUS, they are not as high as would be expected when the fact that FPIPE has only four pipelined FU's is considered. In all machines, percentages for the $2P$, $4P$, and FP schemes are respectably better than those of the C issuing mode.

Bus utilizations are tabulated in Table IX. Only those for EBUS appear as this is the only machine with a limited number of buses. For all of the parallel issuing modes, the bus is much more heavily utilized than in the U and C modes. This is a direct consequence of their higher instruction issuing rate.

XI. ANALYSIS

As noted, the dispatch stack dynamic scheduling mechanism improves over other issuing approaches in two main aspects: 1) one or more instructions can be issued per cycle and 2) instructions can be issued nonsequentially. This allows the IU to supply more instructions per cycle to the EU and attack the problem of under-utilization of the execution resources. The simulation results indicate that computers using multiple functional units, such as the CRAY-1, would benefit substantially from the DS concurrent instruction issuing mechanism.

The results confirm that the C issuing mode is obviously better than the U mode. The C mode keeps the EU busier by issuing more instructions per cycle. This results in using the EU parallelism more effectively, something that is not done at all with the U mode. Consequently, the throughput is increased.

All the measurements indicate that the issuing modes that use the DS parallel instruction issuing mechanism perform better than the C issuing mode. They are capable of issuing more instructions per cycle, resulting in higher utilizations for the multiple functional units of the EU. Due to this, the number of cycles required to execute a given program is reduced, and the throughputs and relative speedups increase. For different IU/EU configurations, these speedups range from a low of 1.71 for the $2P$ mode to a high of 2.79 for the FP mode, relative to the U mode.

The $1P$ issuing mode deserves some attention. Even though it can issue at most one instruction per cycle, it outperforms the C mode because of its ability to issue instructions nonsequentially. This relatively simple enhancement to the CRAY-1 issuing mechanism seems to pay off very well. Speedups for the $1P$ mode range from 1.00 to 2.24, as compared to 1.00 to 1.30 for the C mode. The $1P$ mode is also the most efficient of the issue schemes studied.

Among the parallel issuing modes, the results for the $4P$ and FP modes are practically the same for XBAR, EBUS, and FPIPE. The performance of the $1P$ and $2P$ modes, relative to the more powerful issuing modes, improves with decreasing ideality in the execution unit. This improvement appears in all the measurements. The parallel issuing modes benefit from having a larger DS and more program memory banks. Yet the differences are not as striking as might be expected. In general, the efficiency decreases as the parallel issuing capabilities of the issue mode increase.

One interesting observation is that the $2P$ system performs much better than the C mode, and respectably when compared to the issuing modes with greater parallelism. Thus, even a simple system with a 16-instruction dispatch stack and 4 program memory banks using the $2P$ issuing mode would yield great increases in performance, while not being overly ambitious in terms of the amount and cost of additional hardware.

XII. SYSTEM INTEGRATION: HARDWARE CONSIDERATIONS

Having already shown that considerable speedups are attainable by issuing instructions according to the DS algorithm, some hardware implementation issues are now considered. The reader is referred to [1] for a more thorough treatment of this topic.

The instruction unit of a processor, which contains the DS window, can be conceived as residing on one VLSI chip, with the multiple functional units of the EU occupying additional chips. Hence, the IU control chip dispatches instructions to the FU chips. The DS design should be compact and regular enough to be amenable for VLSI implementation.

Three operational control phases exist in a DS cycle: *fetch*, *issue*, and *update*. It is interesting to note that the fetch and update phases can be carried out concurrently since they operate on different portions of the DS. For instance, instructions might be fetched into a DS buffer while the updating is occurring in the DS itself. Subsequently, a simple merging phase can combine their contents.

The fetch phase is simplified by using a *precedence count memory* (PCM) [28] for maintaining global α and β counts for the entire DS. The PCM is used for assigning α and β values to newly fetched instructions and is updated along with the DS when instructions complete execution. This eliminates the need to recalculate data dependence information every time an instruction is appended to the DS.

An area of potential concern in the DS operation is the shifting of instruction entries required by the update phase. This arises because of the sequentiality that is maintained in the way instructions are placed in the window. However, the following observation can be made to alleviate this concern: once the dependence information has been incorporated into a DS entry, the spatial ordering in the DS need not be preserved. A proof of this can be found in [1]. There are three important benefits derived from this:

1) instructions do not have to be appended to the bottom of the DS,
2) no shifting of DS entries is required, and
3) explicit tags are not necessary because they will implicitly correspond to the DS slots occupied by the instructions.

Another observation that simplifies DS implementation is that it is not necessary for the issue index to be implemented as a sum of the α and β values, as in (5). The logical OR of these fields suffices to summarize the data dependencies of a DS entry as follows:

$$I^2_{\text{OR}} = \alpha(S1)|\alpha(S2)|\alpha(D)|\beta(D). \tag{10}$$

This value should be available as a DS output for controlling the issue phase.

Probably the most important implementation aspect of the DS mechanism is its content addressability for updating the dependence counters. In considering the DS entry fields proposed in Section IV, we see that only the *OP*, *S*1, *S*2, and *D* fields need to be read/writable. Of these, the three register fields also have to be content addressable.

The α and β, which can be implemented with down counters, should be externally writable by the fetch phase using the PCM. Upon encountering a suitable associative match in the update phase, they are internally updated. If these counters are physically located close to the appropriate content addressable cells, this is a relatively fast operation since no external control is required.

XIII. Other Considerations

The usefulness of the proposed DS scheme in enhancing system performance in multiple functional unit processors has been clearly established. In addition to studying implementational aspects in relative detail, Acosta [1] has examined enhancements and other topics related to the mechanism. These are briefly reviewed below.

In terms of conditional branches, various possibilities exist. One is assigning a higher priority to instructions on which the branch is dependent. Another possibility is to incorporate a *delayed branch* into the system [18], [19]. This consists of defining the architecture so that a given number of instructions which sequentially follow the branch are always executed. This requires the use of an optimizing compiler to rearrange the code appropriately. Prefetching instructions along the preferred branch path can also alleviate the conditional branch bottleneck. In general, the emphasis is on maintaining more instructions in the instruction window in order to uncover greater concurrency among them.

The identification of a suitable register allocation scheme, involving static scheduling by an optimizing compiler [2], [9], can be crucial when dynamic scheduling based on instruction dependencies is employed in an instruction unit. A simple and effective approach is to "spread" the execution of a program over as many registers as possible by means of *single-assignment* allocation methods [6]. Other compilation techniques, such as the reduction of expression tree height and conventional optimizations, should be pursued in improving performance in multiple FU processors.

Frequent external interrupts (e.g., context switches) can severely degrade the performance in a system employing a DS. Unless the instruction window is allowed to empty when an interrupt occurs, saving enough state information to resume execution is a difficult and costly operation. In the case of internal interrupts (e.g., division by zero), due to the nonsequential scheduling of instructions, the concept of imprecise interrupts used in the IBM 360/91 must be employed [4].

The DS bookkeeping mechanism maintains dependence information that allows dynamic transformations for improving instruction streams. In fact, the dispatch stack can perform as a dynamic peephole optimizer on an instruction sequence being executed [8], [9], [16]. The optimizations that can be recognized include eliminating useless assignments, redundant computations, and redundant loads.

Additional scenarios for using the dispatch stack include operating on multiple instruction streams (each with its own program counter, condition code bits, and register set) and scheduling for multiprocessor systems.

XIV. Conclusions

The dispatch stack instruction issuing mechanism provides an effective approach to enhancing performance in multiple functional unit processors. Through its ability to dynamically schedule the nonsequential execution of multiple instructions per cycle, the DS can effectively decrease the number of cycles taken to execute a given program. This enhances operational concurrency and, consequently, results in increases in both throughput and FU utilization. When viewed in an ideal setting, the DS conforms to Keller's principle of optimality.

Since the DS mechanism appears as an instruction window in the instruction unit, the dynamic code scheduling it performs is independent of the multiple functional unit configuration in the execution unit. Thus, improvements in performance due to using a DS are possible in widely differing EU configurations. In comparison to serial dispatching schemes, the results in this paper have shown speedups between 1.71 and 2.79 are attainable using various DS parallel issue modes.

Although the DS instruction issuing mechanism requires more complex and costly hardware, the resulting increases in performance indicate that its dynamic code scheduling capabilities are worthwhile for high-performance systems. In addition, dynamic scheduling in hardware can free compilers of burdensome static scheduling decisions. Advances in VLSI technology are allowing more complex systems to be built at lesser costs. Hence, the DS offers both an effective and a practical approach to increasing performance in computers.

XV. Acknowledgment

The authors express their appreciation to C. K. Chow, H. Ling, W. Nohilly, and D. Tjon for their assistance and encouragement.

References

[1] R. D. Acosta, "Evaluation, implementation, and enhancement of the dispatch stack instruction issuing mechanism," Ph.D. dissertation, School Elec. Eng., Cornell Univ., Ithaca, NY, Aug. 1985.

[2] A. V. Aho and J. D. Ullman, *Principles of Compiler Design.* Reading, MA: Addison-Wesley, 1977.

[3] F. E. Allen and J. Cocke, "A catalogue of optimizing transformations," in *Design and Optimization of Compilers*, Courant Computer Science Symposium 5, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[4] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM system/360 model 91: Machine philosophy and instruction handling," *IBM J.*, vol. 11, pp. 8-24, Jan. 1967.

[5] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. Ass. Comput. Mach.*, vol. 21, pp. 201-206, Apr. 1974.

[6] D. D. Chamberlain, "The single-assignment approach to parallel processing," in *Proc. AFIPS Fall Joint Comput. Conf.*, 1971, pp. 263-269.

[7] A. E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS 164 family," *Computer*, vol. 14, pp. 18-27, Sept. 1981.

[8] J. W. Davidson and C. W. Fraser, "Eliminating redundant object code," in *Proc. 9th Annu. ACM Symp. Principles Programming Lang.*, Jan. 1982, pp. 128-132.

[9] G. Goos and W. M. Waite, *Compiler Construction.* New York: Springer-Verlag, 1984.

[10] R. W. Hockney and C. W. Jesshope, *Parallel Computers.* Bristol, Great Britain: Hilger, 1981.

[11] R. M. Keller, "Look-ahead processors," *Comput. Surv.*, vol. 7, pp. 63-72, Dec. 1975.

[12] B. W. Kernighan and D. E. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[13] P. M. Kogge, *The Architecture of Pipelined Computers.* New York: McGraw-Hill, 1981.

[14] D. J. Kuck, Y. Muraoka, and S.-C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Trans. Comput.*, vol. C-21, pp. 1293-1310, Dec. 1972.

[15] D. J. Kuck, "A survey of parallel machine organization and programming," *Comput. Surv.*, vol. 9, pp. 29-59, Mar. 1977.

[16] W. M. McKeeman, "Peephole optimization," *Commun. Ass. Comput. Mach.*, vol. 3, pp. 443-444, July 1965.

[17] F. H. McMahon, "Fortran CPU performance analysis," Lawrence Livermore Lab., 1972.

[18] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *Computer*, vol. 15, pp. 8-21, Sept. 1982.

[19] G. Radin, "The 801 minicomputer," in *Proc. Symp. Architect. Support Programming Lang. Operat. Syst.*, pp. 39-47, Mar. 1982.

[20] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *Comput. Surv.*, vol. 9, pp. 61-102, Mar. 1977.

[21] J. P. Riganati and P. B. Schneck, "Supercomputing," *Computer*, vol. 17, pp. 97-113, Oct. 1984.

[22] R. M. Russell, "The CRAY-1 computer system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 63-72, Jan. 1978.

[23] V. P. Srini and J. F. Asenjo, "Analysis of the Cray-1S architecture," in *Proc. 10th Annu. Symp. Comput. Architect.*, pp. 194-206, June 1983.

[24] J. E. Thornton, *Design of a Computer: The Control Data 6600.* Glenview, IL: Scott, Foresman and Co., 1970.

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

[26] G. S. Tjaden, "Representation and detection of concurrency using ordering matrices," Ph.D. dissertation, Johns Hopkins Univ., Baltimore, MD, 1972.

[27] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J.*, vol. 11, pp. 25-33, Jan. 1967.

[28] H. C. Torng, "Arithmetic engines in the VLSI environment," School of Elec. Eng., Cornell Univ., Itahca, NY, Tech. Reps. EE-CEG-82-7 and EE-CEG-83-3, Oct. 1982 and July 1983.

[29] ——, "Instruction issuing mechanism for a processor having multiple functional units," U.S. Patent Application, Oct. 7, 1983.

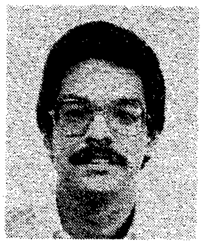
[30] ——, "An instruction issuing mechanism for performance enhancement," School of Elec. Eng., Cornell Univ., Ithaca, NY, Tech. Rep. EE-CEG-84-1, Feb. 1984.

[31] A. K. Uht, "Exploitation of low-level concurrency: An implementation and architecture," Dep. Elec. Comput. Eng., Carnegie-Mellon Univ., Pittsburgh, PA, Res. Rep. CMUCAD-85-52, May 1985.

[32] R. G. Wedig, "Detection of concurrency in directly executed language instruction streams," Ph.D. dissertation, Stanford Univ., Stanford, CA, June 1982.

[33] S. Weiss and J.E. Smith, "Instruction issue logic for pipelined supercomputers," in *Proc. 11th Annu. Symp. Comput. Architect.*, June 1984, pp. 110-118.

[34] C. L. Wu, "Interconnection networks," *Computer*, vol. 14, Dec. 1981.



Ramón D. Acosta (S'80-M'81-S'81-M'85) received the B.S. and M.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1981 and 1982, respectively, and the Ph.D. degree in electrical engineering from Cornell University, Ithaca, NY. His doctoral work was carried out under the auspices of a Cooperative Research Fellowship from AT&T Bell Laboratories.

His main research interests include hardware/software tradeoffs in computer architecture, parallel processing, and VLSI systems. At present he is a Member of the Technical Staff at the Microelectronics and Computer Technology Corporation (MCC), Austin, TX, working on applications of artificial intelligence in digital system synthesis.

Dr. Acosta is a member of the Association for Computing Machinery, American Association for Artificial Intelligence, Tau Beta Pi, and Eta Kappa Nu.

Jacob Kjelstrup was born in Oslo, Norway, on April 3, 1960. He received the B.S. in electrical engineering in 1983 and the M.E.E. degree in 1984, all from Cornell University, Ithaca, NY.

Since July 1984, he has been a Member of Technical Staff of Bell Communications Research, Red Bank, NJ. His current interests include software and hardware developments in telecommunications and computer systems.

H. C. Torng (M'61-SM'68) received the B.S. degree in electrical engineering from National Taiwan University, Taiwan, China in 1955, and the M.S. and Ph.D. degrees from Cornell University, Ithaca, NY, in 1958 and 1960, respectively.

In 1960, he joined the faculty of Cornell University where he is a Professor of Electrical Engineering. He worked at Bell Laboratories in 1966-1967 and in 1980-1981; he is a consultant to several industrial organizations. His present interests are in processor organizations, digital systems, and telecommunications. He is the author of two books on the subject of logic design and over 40 articles.

Dr. Torng is an Editor of the IEEE TRANSACTIONS ON COMPUTERS. He has served as an IEEE Computer Society Distinguished Visitor since 1983 and is a member of IEEE Award Candidates Search Committee. He is a member of the Association for Computing Machinery, Sigma Xi, Phi Kappa Phi, and Eta Kappa Nu. In 1985, he received the Spira Excellence in Teaching Award.