Single Instruction Stream Parallelism Is Greater than Two

Michael Butler, Tse-Yu Yeh, and Yale Patt
Department of Electrical Engineering
and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

Mitch Alsup, Hunter Scales, and Michael Shebanow
Motorola Incorporated
Microprocessor and Memory Technology Group
6501 William Cannon Drive West
Austin, Texas 78735

Abstract

Recent studies have concluded that little parallelism (less than two operations per cycle) is available in single instruction streams. Since the amount of available parallelism should influence the design of the processor, it is important to verify how much parallelism really exists. In this study we model the execution of the SPEC benchmarks under differing resource constraints. We repeat the work of the previous researchers, and show that under the hardware resource constraints they imposed, we get similar results. On the other hand, when all constraints are removed except those required by the semantics of the program, we have found degrees of parallelism in excess of 17 instructions per cycle. Finally, and perhaps most important for exploiting single instruction stream parallelism now, we show that if the hardware is properly balanced, one can sustain from 2.0 to 5.8 instructions per cycle on a processor that is reasonable to design today.

1 Introduction

The increasing density of VLSI circuits has motivated research into ways to utilize large numbers of logic elements to improve computational performance. One way to use these elements is to replicate multiple functional units on a single chip. This technique is effective only if the amount of instruction–level parallelism present in real applications warrants it. Early studies[3, 5, 14, 15, 16] suggested that this was in fact the case. More recently, some researchers[1, 2] have concluded that insufficient parallelism exists in real, nonscientific applications to support processors that will execute more than two instructions per cycle.

Since the amount of available parallelism should influence the design of the processor, it is important to verify how much parallelism exists. To do this, we have undertaken a study of available parallelism in optimized, compiled code. We have used nine of the ten programs in the SPEC suite,[1] a set of real applications that have become the de facto standard in compute-bound performance benchmarks. We have measured the performance of these benchmarks on several execution models, including those studied by Patt and Hwu[3, 5] and by Smith and Johnson[1] and Jouppi[2]. We have also measured the performance of these benchmarks on an unconstrained execution model in order to quantify how much parallelism exists in these benchmarks that could be exploited if the artifacts of the processor did not prevent it.

Our results show that, with undue constraints on the processor, it is difficult to sustain parallel execution of more than two instructions per cycle. On the other hand, when all constraints are removed except those required by the semantics of the program, we have found degrees of parallelism in excess of 17 instructions per cycle. Finally, we show that when the hardware is properly balanced, one can sustain an execution rate of 2.0 to 5.8 instructions per cycle on a processor that is reasonable to design today.

This paper is organized in six sections. Section 2 discusses restricted data flow, an abstract execution model that allows available instruction stream parallelism to be exploited. Section 3 describes our experiments: the simulator, the benchmarks, and the machine configurations tested. Section 4 reports the results of our simulations and discusses the influence of various machine features on performance. The model used by Smith and Johnson[1], the unconstrained case, and several machines which are realistic to implement today, are all identified. Section 5 presents a brief discussion of implementation issues. Section 6 offers some concluding remarks and discusses the future work we have planned.

[1] The Nasa7 benchmark was not simulated because this benchmark consists of seven independent loops. Due to time constraints, we omitted these loops.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-394-9/91/0005/0276 $1.50

2 The RDF Model of Execution

To exploit whatever parallelism exists in the instruction stream, one needs an execution model devoid of artifacts that limit the utilization of that parallelism. The abstract restricted data flow (RDF) paradigm is such a model. It is characterized by three parameters: window size, issue rate, and instruction class latencies.

Processing consists of systematically issuing instructions from a program's dynamic instruction stream, converting those instructions into a dynamic data flow graph, scheduling instructions for execution when their flow dependencies have been resolved, and retiring those instructions after execution has completed. The dynamic instruction stream is created by an omniscient branch predictor that always knows the way a conditional branch will execute. Instructions are issued in the order they appear in the dynamic instruction stream.

The window size is the maximum number of instructions from the dynamic instruction stream that can be present in the dynamic data flow graph at any instant of time. (Instructions can be issued as long as the number of instructions in the window is less than the window size.) The issue rate is the maximum number of instructions that can be removed from the dynamic instruction stream and entered into the window in a single cycle. We call the group of instructions that can be brought into the machine in one cycle a packet. The instruction class latencies specify the set of operations and the latency associated with each operation. In the RDF model, the number of functional units is unbounded, each can perform every desired operation, and the latency associated with each operation is specified.

The RDF model was first defined because its parameters correspond to some important practical constraints on cpu design [3]. The RDF restriction on window size corresponds to a practical limitation on the amount of buffering that can be supported. The restriction on issue rate corresponds to the limitation on instruction memory bandwidth.

Clearly, the RDF model specified above, with its omniscient branch prediction and unbounded functional units, is not realizable. Nonetheless several variations of the RDF model are interesting to study. For example, if in addition to the functional units, the window size and issue rate are also unbounded, we have an execution unit that presents no impediment to exploiting all the parallelism present in the application. We call this model an unrestricted data flow (UDF) machine. It specifies all the parallelism available in the instruction stream.

On the other hand, if we restrict the various parameters of the RDF model, we obtain upper bounds on the level of performance that is possible. For example, the RDF model gives an upper bound on the performance of a machine that can support a specific window size and issue rate if function unit capability and branch prediction were not a problem. If we couple the RDF model to a real branch predictor, we obtain an upper bound on the performance of a machine that can support a specific window size and issue rate if functional unit capability were not a problem. Finally, if we further restrict our RDF model to a functional unit configuration that is implementable, we have an execution model that corresponds to a realizable machine that efficiently exploits the available parallelism in a single instruction stream.

We first specified a realizable implementation of the RDF model, the High Performance Substrate (HPS) in 1985 [3]. HPS was originally developed as a speedup mechanism for complex instruction set architectures, although we quickly discovered its applicability to the implementation of all architectures. Today, it is continually being refined by our research group in an effort to improve its performance and reduce its cost of implementation. Its ultimate objective, emulating an (optimal) RDF machine, has not changed.

In this paper, we are concerned with the influence of the RDF parameters on the performance of the SPEC benchmarks. We will identify several implementable RDF models, with differing values for window size, issue rate, and functional unit capability.

3 Experiments

3.1 Benchmarks

The results presented in this paper are for nine integer and floating point programs from the SPEC suite: eqntott, espresso, gcc, li, doduc, fpppp, matrix300, spice2g6, and tomcatv, compiled for and run under the M88000 instruction set architecture. The benchmarks were compiled using the Green Hills FORTRAN 1.8.5 compiler or the Diab Data C Rel. 2.4 compiler with all optimizations turned on. (A particular compiler was selected for a given benchmark if that compiler produced the most efficient object code (i.e. shortest run time) when run on an MC88100–a conventional scalar processor.) All benchmarks were run unchanged with the following exceptions: cpp is not called in eqntott, gcc was run without cpp and used the output of the preprocessor as the input file. Due to time limitations, each benchmark was simulated for ten million instructions.

Table 1 shows instruction classes and their simulated execution latencies. Each instruction class is listed with its abbreviation (A for floating point add, M for floating point and integer multiply, L for memory loads, etc.), its execution latency (in cycles), and a description of the instructions that belong to that class. The latencies for all but one machine were taken from Smith and Johnson[1]. We know of several machines with smaller floating point latencies, however, so in order to investigate the effect of shorter latencies, we simulated one

Instruction Class	Execution Latency	Description
(A) FP Add	6	FP add, sub, and convert
(M) Multiply	6	FP mul and INT mul
(D) Divide	12	FP div and INT div
(L) Mem Load	2	Memory loads
(S) Mem Store	-	Memory stores
(B) Branch	1	Control instructions
(T) Bit Field	1	Shift, and bit testing
(I) Integer	1	INT add, sub and logic OPs

Table 1: Instruction Classes and Latencies

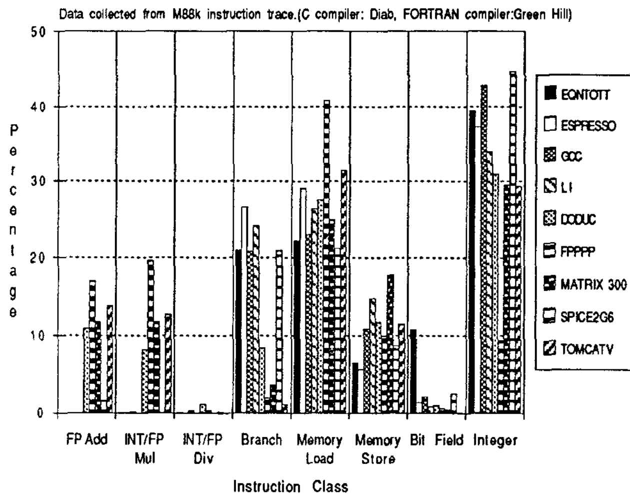

Figure 1: Benchmark Code Analysis

configuration using floating point latencies of 3, 3, and 8 for floating point add, multiply, and divide, respectively.

Figure 1 shows histograms of the dynamic frequency of each instruction class for each of the nine benchmarks. Within each instruction class, the nine vertical bars correspond respectively to the nine benchmarks listed at the right of the figure. As can be seen, approximately 33 percent of instructions are simple integer ALU operations; loads and branches comprise another 28 and 15 percent respectively. Figure 2 presents a stack chart of the same data organized by benchmark.

3.2 Machine Configurations

The instruction class frequency is particularly important for examining performance of machines such as the ones we simulated. Machine resources must reflect the instruction frequency if we are to achieve efficient utilization of functional units. Also, since issue is dependent on functional unit configuration (only one instruction can be issued to each functional unit each cycle), the machine configuration contributes to the upper bound on execution throughput.

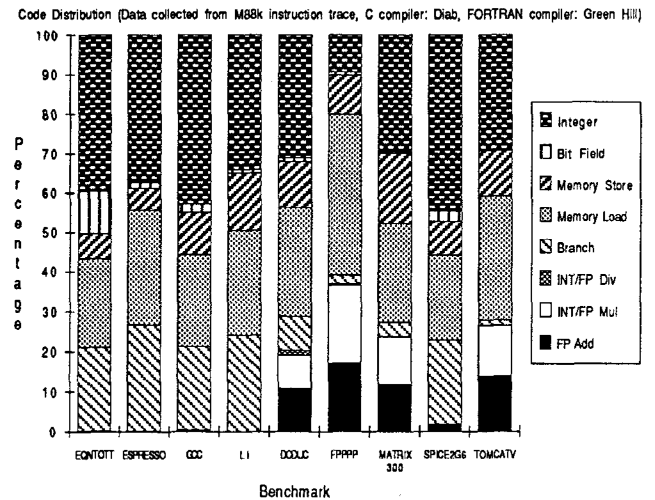

Figure 2: Instruction Class Distribution in Benchmarks

The machine configurations simulated are listed in Figure 3. Each machine is specified by its set of functional units, branch prediction mechanism, issue rate, retirement mechanism, the size of an instruction packet (i.e. a group of instructions that are issued in a single cycle), and the characteristics of the load/store pipes and instruction prefetch buffer.

With respect to its functional unit configuration, each set of parentheses corresponds to a single functional unit, and the letters within the parentheses indicate the instruction classes that the functional unit is capable of executing. In the Smith/Johnson machines, each functional unit is capable of executing only one class of instructions, so each set of parentheses contains only one letter. The UDF and RDF.I8 machines can be described as machines with an unbounded number of functional units, each of which is capable of executing all instruction classes. The remaining machines have functional units that are capable of handling multiple instruction classes. For example (see Figure 3) the machine identified as 4F2M has four functional units. One of these functional units can execute instructions from the classes A, M, D, and I, one can execute instructions from B and I, and two can execute instructions from classes L, S, T, and I.

In choosing machine configurations an effort was made to match functional unit capabilities to instruction class frequencies. In addition, the relative cost of adding functionality was considered. If infrequently used functional units could be made to service an additional (frequently used) class of instructions at nominal increase in hardware cost, the resultant machine performance would be enhanced. For example, integer ALUs service the most frequently occurring instructions and can be implemented at a nominal cost relative to the rest of the machine. As such, all functional units are

Machine Name	Functional Unit Configuration	Branch Prediction	Packet Issue Rate	Retire Mech.	# of Inst./ Packet	Load / Store	Inst. Prefetch Buffer
S&J 4	(A)(M)(D)(L)(S)(B)(T)(I)(I)(L)	85% Syth.	4	Out of Order	1	Sequential	Aligned 4 words
UDF	∞ (A,M,D,L,S,B,T,I)	100%	Infinite	Out of Order	1	Out of Order	Unaligned Fetch
RDF.I8	∞ (A,M,D,L,S,B,T,I)	100%	8	Out of Order	1	Out of Order	Unaligned Fetch
4F2M	(A,M,D,I)(B,I) 2(L,S,T,I)	100%	1	In Order	4	Out of Order	Unaligned Fetch
4F2M B85	(A,M,D,I)(B,I) 2(L,S,T,I)	85% Syth.	1	In Order	4	Out of Order	Unaligned Fetch
6F3M	(A,I)(M,D,I)(B,I) 3(L,S,T,I)	100%	1	In Order	6	Out of Order	Unaligned Fetch
8F3M	(A,I)(M,I)(D,I)(T,I)(B,I) 3(L,S,T,I)	100%	1	In Order	8	Out of Order	Unaligned Fetch
8F3M.B85	(A,I)(M,I)(D,I)(T,I)(B,I) 3(L,S,T,I)	85% Syth.	1	In Order	8	Out of Order	Unaligned Fetch
8F3M RB	(A,I)(M,I)(D,I)(T,I)(B,I) 3(L,S,T,I)	Real BP	1	In Order	8	Out of Order	Unaligned Fetch

Figure 3: Simulated Machine Configurations
With respect to its functional unit configuration, each set of parentheses corresponds to a single functional unit, and the letters within the parentheses indicate the instruction classes that the functional unit is capable of executing. A number appearing before a set of parentheses indicates that multiple copies of the functional unit exist. In the case of our abstract machines, UDF and RDF.I8, an unbounded number of functional units are present.

capable of executing simple integer operations (except in the simulations of Smith and Johnson's machines).

3.3 Simulation Process

The simulator is capable of modeling a wide range of machines as well as execution models. The simulation process works as follows:

An instruction level simulator for the MC88100 (ISIM) reads in the object code and simulates execution, producing an instruction trace. Our RDF simulator reads in a configuration file which describes the machine to be simulated and then begins processing the dynamic instruction stream produced by ISIM. The simulator performs data dependency analysis and scheduling, and gathers execution rate statistics.

Our simulator makes several simplifying assumptions:

- Register renaming is performed. Renaming eliminates anti and output dependencies and is critical for achieving high performance with the models of execution we simulated.
- No memory renaming is performed. In the early stages of the simulator, we supported memory renaming at the byte granularity level, but found that renaming occurred so infrequently, either because of the algorithms or the actions of the compilers, that renaming did not noticeably improve performance.
- No caches are explicitly modeled in the current version of the simulator. We assume 100 percent hit rates for both I and D caches.
- No bank conflicts are modeled.
- All machines modeled are capable of performing store/load forwarding if stores and loads to the same memory location are both resident in the window. We found that this forwarding occurs very infrequently.
- All functional units are fully pipelined (i.e. able to initiate a new operation each cycle) and are mutually independent.
- Instructions are removed from the window (i.e. "retired") in whole packet units after all instructions in that packet have completed execution. This assumption corresponds to our use of checkpointing [6] as a mechanism for supporting both branch prediction miss recovery and precise interrupts.
- When a trap is encountered, the machine being simulated must stop issue, wait for all instructions currently in the window to complete, and then execute the trap instruction.

There are several key parameters that define execution in the simulator.

- Window size - This parameter limits the total number of instructions that can exist in the machine at any one time. An instruction is considered in the machine, and thus occupying space in the window, from the time it is issued until it is retired. Instructions can be in one of four states: waiting for operands, ready but waiting for the assigned functional unit to become free, executing, or waiting for retirement. Window size is given in the number of issue packets.
- Packet Issue Rate - This parameter indicates the number of packets that can be issued per cycle. A packet consists of a group of instructions that are brought into the machine in a single cycle. Normally the packet issue rate is set to one and the number of instructions issued per cycle is determined solely by the machine configuration.
- Prefetch Buffer Configuration - An instruction prefetch buffer may be modeled with specified size and refill characteristics. This was used only in the Smith/Johnson machines in order to match their configurations.
- Functional Unit Configuration - The number and capabilities of all functional units are specified in the configuration file. Each functional unit is defined by the instruction classes it is capable of executing.

- In/Out of Order Execution - This flag indicates whether instructions can be executed out–of–issue order for each functional unit. In–order execution still allows for slip to occur between functional units — i.e. operations are executed in–order within each functional unit, but out–of–order with respect to other functional units.
- Branch Prediction - There are two types of branch prediction supported - synthetic and real[11]. With synthetic branch prediction, the branch prediction accuracy is specified in the machine configuration file. As branches are encountered in the dynamic instruction stream, a random number is generated to determine whether the branch is predicted correctly or not. This is the mechanism used by Smith/Johnson [1]. With real branch prediction, an actual prediction is made and then compared to the real outcome as determined by the trace. If a prediction fails, issue is stalled until the branch is resolved. This models the performance of a checkpointing mechanism.
- Instruction Class Latency - These latencies describe the number of clock cycles required to execute a given type of instruction. The latencies we used are given in table 1. The only exception to these latencies is the machine model with the "SF" suffix. This machine used smaller floating point latencies of 3, 3, and 8 for floating point add, multiply, and divide respectively.
- Unbounded Functional Units - This flag allows the simulator to model a machine with an unbounded number of functional units. With this machine, there are effectively as many functional units as are needed in any given cycle.
- Instruction Issue Constraint - This flag indicates whether instructions are assigned to a particular functional unit at issue time (with at most one instruction going to each functional unit), or whether they are issued to a common window. The consequence of assigning instructions at issue time is that issue is constrained to match each instruction to a unique functional unit. Thus if an instruction can not be assigned to a functional unit capable of executing it, that instruction and all instructions following it cannot be issued in that cycle.

4 Simulation Results

Each benchmark was simulated under three sets of machine configurations: (1) a model faithful to the previous work of Smith and Johnson[1], (2) a model representing no artificial constraints on the processor, and (3) realizable machines obtained by restricting the values for parameters of the unconstrained machine. We will discuss the simulation results for each set of machine models in turn.

4.1 Previous Work

For each of the nine benchmarks (Figures 4 through 12), the curve labeled SJ4 presents the results of modeling the assumptions and machine configuration of Smith and Johnson's machine 4[1]. (All four machine configurations were simulated, but only SJ4 results are presented for clarity in the figures. SJ4 was selected because it demonstrated the highest performance of the four Smith/Johnson machine models.) In this model, a four instruction wide prefetch buffer is filled with memory words aligned on a four-instruction-word boundary. Thus, as many as 3 of the 4 instructions in the prefetch buffer may not be required. Instruction issue, the act of bringing instructions into the machine, is not constrained by functional unit configuration. Thus, for example, more than one integer instruction can be placed in the window in a cycle even though there is only a single integer ALU (The integer instructions will of course be scheduled for different cycles). The functional units execute instructions in data-dependent order and instructions are removed from the window upon completion. Loads and stores are executed in order and store instructions are executed only after all previously issued instructions have completed. Branch prediction accuracy is 85 percent (synthetic) as in [1].

While a direct comparison is not possible due to different instruction set architectures and different compilers, one notes from Figures 4 through 12, that the results of our simulations are consistent with results published by Smith and Johnson. Our simulations show performance between 1.7 and 2.1 IPC for the integer benchmarks, and between 1.4 and 3.1 IPC across the floating point benchmarks. Smith and Johnson show performance in terms of speedup over a scalar processor, and report the harmonic mean for their entire benchmark suite. To provide a rough comparison between our results and those published by Smith and Johnson, the IPC of the superscalar machine executing a particular benchmark is divided by the IPC of a scalar machine.

Several details of the Smith/Johnson machines impede high performance unnecessarily. The most obvious limit to achieving high performance is the issue rate of four. The machine is comprised of nine independent, pipelined functional units, allowing for a peak execution rate of nine IPC. However, since a maximum of four instructions are issued per cycle, peak execution has been reduced to four IPC.

Aside from the four instruction wide fetch limit, there are three additional machine characteristics that limit performance: a poorly balanced machine configuration, an overly–constrained prefetch buffer, and a sequential

load/store execution constraint.

A significant source of performance degradation in the Smith/Johnson machines is an imbalance in the machine configurations. The scarcity of integer units gives rise to a performance bottleneck. If integer operations comprise nearly 38 percent of the instructions in the integer benchmarks and the machine contains only one integer unit, then it is unreasonable to expect a speedup of much greater than two simply because the integer ALU will be saturated. This is in fact what happens in our simulations — the integer units are constantly busy and the execution time for the program is determined largely by contention for the integer ALU resource.

Another bottleneck to performance is the behavior of the prefetch buffer. Because the buffer is aligned with respect to memory, any branch targets in the middle of cache lines will reduce the number of useful instructions fetched. Assuming a uniform distribution and small basic blocks, on average only 2.5 useful instructions will be fetched per cycle by such a prefetch buffer. As discussed in [1], if the prefetch buffer were expanded to fetch two lines from the i-cache, and perform alignment so as to produce four useful instructions (in the absence of the end of basic blocks), performance would improve. Another shortcoming of the prefetch buffer is that it is only refilled when completely empty. If, for instance, the buffer contains four instructions but can only issue two due to a full window, the prefetch buffer does not attempt to fetch new instructions into the empty buffer space. The next cycle, regardless of the available window space, the prefetch buffer can only issue two instructions. This inefficiency can be eliminated at the cost of a more complex prefetch controller by prefetching anytime there is room in the prefetch buffer. These two features unnecessarily limit the ability of the machine to issue instructions and thus impede execution throughput.

Finally, in the Smith/Johnson model, execution of loads occur strictly in sequential issue order with respect to other loads, and stores are executed only after all previous instructions (i.e. instructions issued before the store) have been executed. These constraints are applied in order to maintain a consistent state in the memory system. This is unnecessary, however, with the support of a checkpointed write buffer as described in [6].

When all of the changes suggested above are implemented in a machine model, the improvements have a synergistic effect and performance improves significantly. For instance, removing the sequential load/store constraint allows for greater performance to be gained from the addition of extra load/store pipes. The result of implementing all of these is a performance increase of 30 to 50 percent (.6 to 1.0 IPC) over the Smith/Johnson model, across the four integer benchmarks tested (gcc, li, eqntott, and espresso).

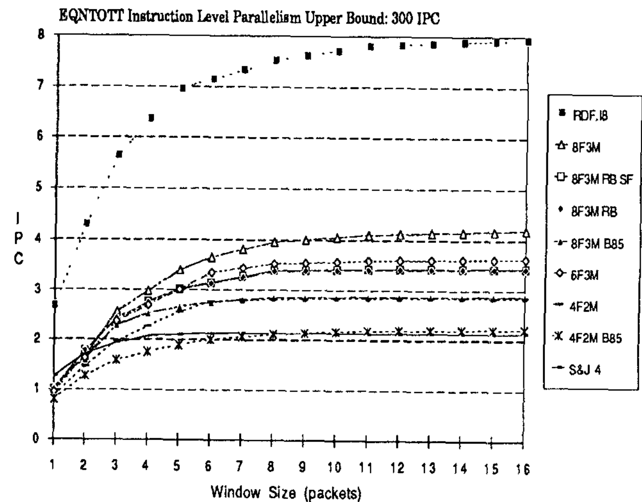

Figure 4: Instruction Level Parallelism of EQNTOTT

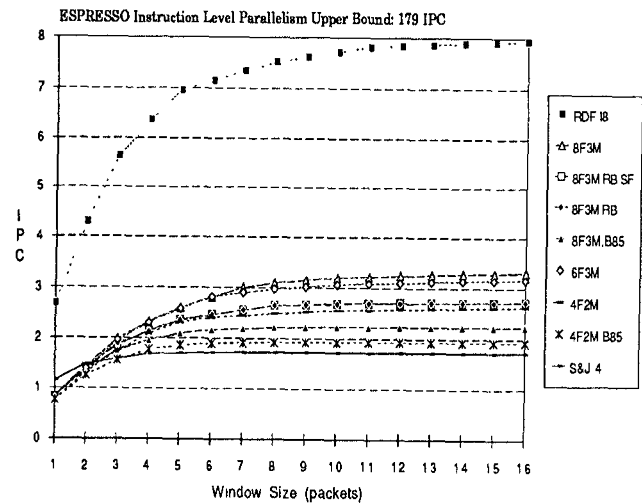

Figure 5: Instruction Level Parallelism of ESPRESSO

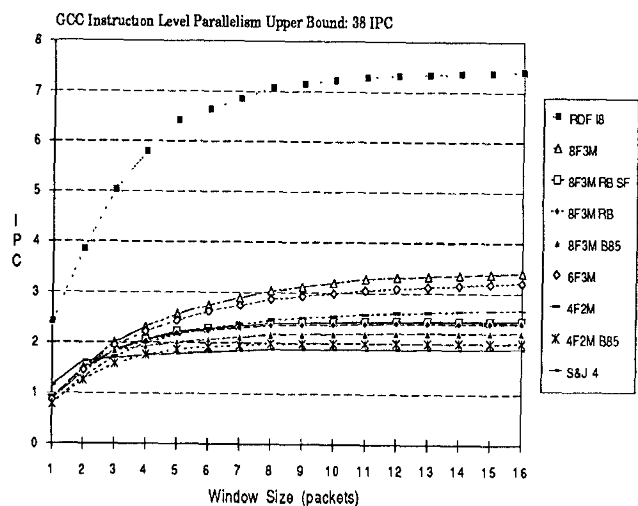

Figure 6: Instruction Level Parallelism of GCC

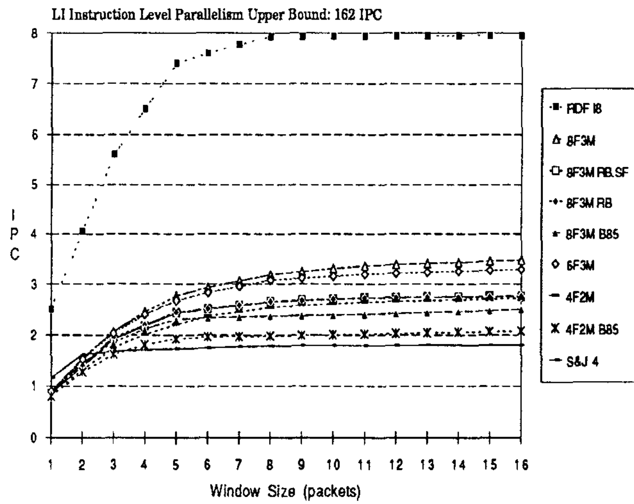

Figure 7: Instruction Level Parallelism of XLISP

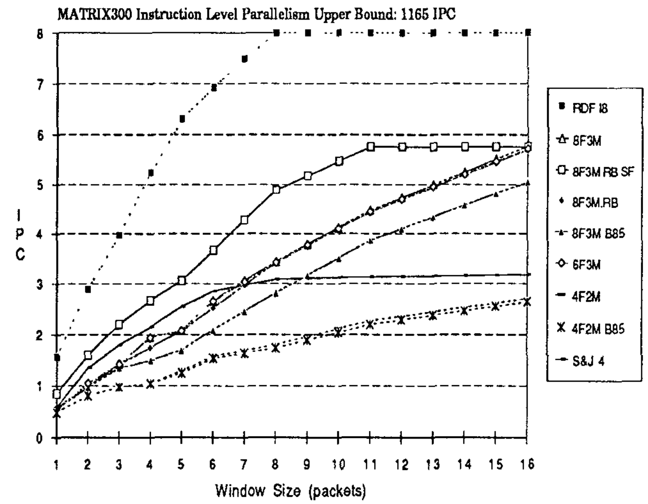

Figure 10: Instruction Level Parallelism of MATRIX300

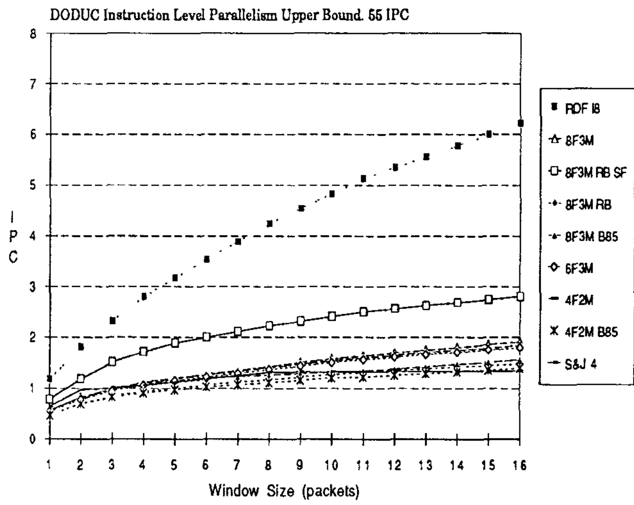

Figure 8: Instruction Level Parallelism of DODUC

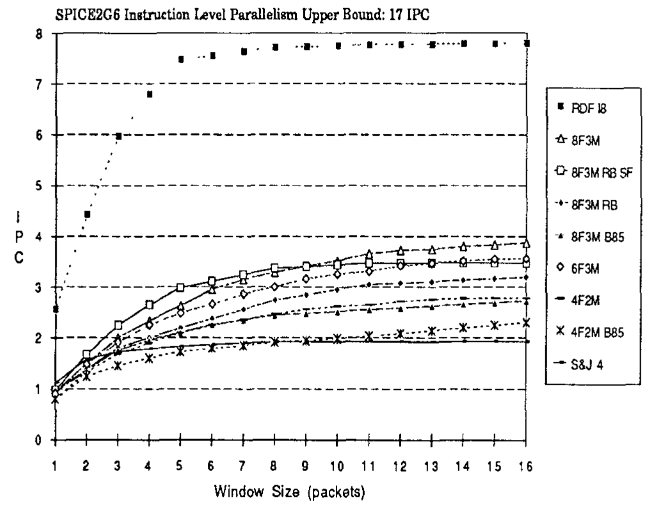

Figure 11: Instruction Level Parallelism of SPICE2G6

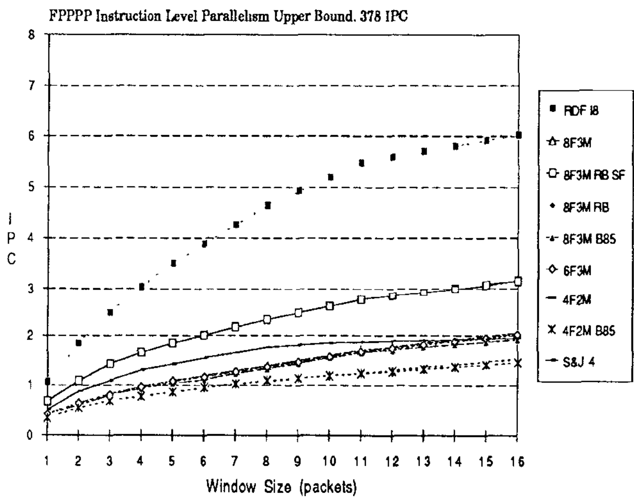

Figure 9: Instruction Level Parallelism of FPPPP

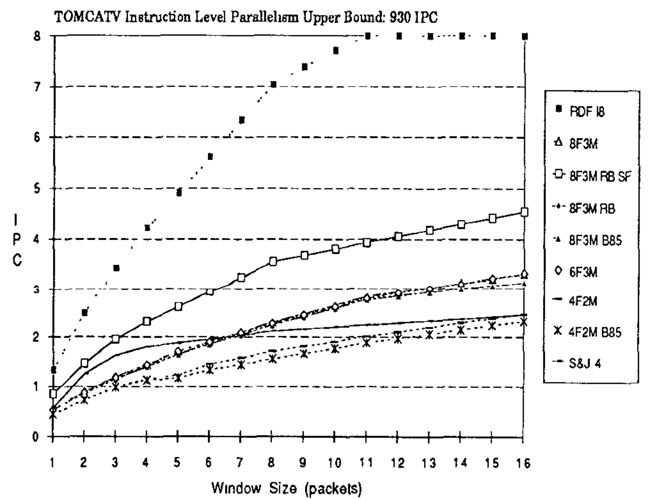

Figure 12: Instruction Level Parallelism of TOMCATV

We also note that we have simulated models based on the assumptions of Jouppi[2] and obtained results similar to his published data (less than two IPC). The primary impediments to high performance for Jouppi's machine are in–order execution and no speculative execution beyond branches. In–order execution imposes a serious limitation on performance by effectively introducing false dependencies between an instruction and all instructions that precede it in the I–stream. Lack of speculative execution limits the machine's ability to utilize parallelism to that which is within a basic block.

4.2 The RDF Machine

For each of the nine benchmarks, Figures 4 through 12, the highest performance curve shows the performance of a restricted dataflow machine with an issue rate of eight instructions per cycle. This abstract machine issues eight instructions each cycle regardless of instruction class, until the window is full. Functional units, each capable of executing any type of instruction, remove instructions from the window out–of–order as they complete execution. No other restrictions are placed on execution. Branch prediction is omniscient. Clearly this is an unrealizable machine model. However, the simulation results provide an upper bound for machines that approximate the RDF model with an issue rate of eight. Note that performance approaches an asymptote of eight IPC.

If we do not restrict window size or fetch rate, the simulator provides an absolute upper bound for that particular benchmark. This is the Unrestricted Data Flow Machine (UDF). In this case, execution is constrained only by true dependencies. This value, reported at the top of each of graph, ranges from 17 to 1165 IPC and indicates all the parallelism that exists in the program segments traced.

4.3 The Middle Ground

4.3.1 Configurations

For each of the nine benchmarks, Figures 4 through 12, the middle curves show the performance of a series of machine configurations under various assumptions. The assumptions that drove the selection of this set of machine configurations were based on a desire to pick configurations that are implementable, coupled with an interest in knowing the effect of better branch prediction schemes. Consequently, each of the machine configurations combines a realistic window size, issue rate, and set of functional units with one of the following branch predictors: 100 percent accurate, 85 percent accurate (synthetic), and a real branch predictor.

The model of execution simulated in these experiments differs from the Smith/Johnson model in that instructions are entered and removed from the window in issue packet units, only after all instructions in the packet have been completed, and in the order in which the packets were issued. This is in contrast to the Smith/Johnson model where instructions are removed out–of–order as soon as the instruction has completed execution. This constraint, required by the mechanism we chose for implementing checkpointing, reduces the effective window size. While this is not significant for the integer benchmarks, it impacts the performance of the floating point benchmarks. As seen in the figures, the Smith/Johnson machine performs better than our machines on the floating point intensive benchmarks for smaller window sizes because it makes more effective use of the window.

4.3.2 Results

Four-functional-unit machines with two memory units and either 85 percent or 100 percent branch prediction accuracy are shown to demonstrate the performance advantages of a properly balanced machine with no prefetch buffer constraints. Performance of the machine with 100 percent branch prediction ranges from 2.7 to 2.9 IPC for the integer benchmarks and 1.5 to 2.8 IPC for the floating point benchmarks. Performance of the machine with 85 percent branch prediction ranges from 1.9 to 2.2 IPC (integer) to 1.4 to 2.7 IPC (floating point).

A six-functional-unit configuration with three memory units is presented with 100 percent branch prediction. Performance ranges from 3.2 to 3.6 IPC for the integer benchmarks and from 1.8 to 5.7 for the floating point benchmarks.

Several eight functional unit machines with three memory units are also presented. These machines each have different branch prediction characteristics: 100 percent synthetic, 85 percent synthetic, and real branch prediction. The real mechanism predicts the outcome of a branch based on the history of that branch in conjunction with dynamically gathered branch characteristics of the running program. Performance of these machines suffers from the one branch per cycle limitation we assumed for the issue mechanism. The average size of basic blocks (five) limits the advantage of additional functional units. Performance ranges from approximately 2.2 to 2.9 IPC for the integer benchmarks using 85 percent branch prediction, and from 1.6 to 5.0 for the floating point benchmarks. With the real branch predictor, performance of between 2.4 and 3.4 (integer), and 1.9 and 5.8 (floating point) is obtained. Because of the significant impact of floating point latencies on the performance of the floating point benchmarks, and the existence of several machines with smaller latencies, we repeated the experiment for the eight functional unit machine using smaller latencies. With the real branch predictor, the performance of the smaller latency ma-

chine ranges from 2.8 to 5.8 on the floating point benchmarks. The performance on the integer benchmarks remains unchanged.

4.3.3 Analysis

Performance, measured in instructions executed per clock cycle, can be modeled as:

$$ipc = I * \mu * \delta$$

where I is the maximum issue rate in instructions per clock cycle, μ is the *static issue efficiency* factor, and δ is the *dynamic execution efficiency* factor. Both μ and δ range between zero and one inclusive.

The maximum issue rate I is determined by the designer and establishes an upper bound on performance. The static issue efficiency factor μ is determined by the dynamic instruction stream (which is program/data dependent) and by the issue restrictions of the machine. For example, a restriction of at most one branch per clock cycle and no instructions issued beyond a branch will reduce the number of instructions that can be issued in a single clock below the maximum value I unless the dynamic instruction stream has one branch every I instructions. Other restrictions, such as those caused by functional unit conflicts (e.g., only one multiply issue per clock due to the presence of only one multiplier) will further reduce μ. Clearly, μ can be improved through compiler assistance. A compiler for a superscalar machine could reorganize code such that issue restriction effects are minimized. It should be emphasized that for the results presented, no superscalar optimizations were performed.

The effects of the static issue efficiency factor can be clearly seen in the results by comparing the performance levels of the RDF.I8 machine to that of the 8F3M machine. The RDF machine is limited only by window size and operation latencies. The 8F3M machine adds issue restrictions, such as one branch per issue cycle and only 3 memory ports. Otherwise, for any given window size, and with the exception of retirement policy, both machines are identical. The range for this factor is approximately .33 to .72.

The dynamic execution efficiency represents the fraction of clock cycles in which instructions were effectively issued. This is determined by four factors: instruction pointer availability, instruction availability, branch misprediction losses, and full window frequency. In a pure RDF model, the first three factors play no part since unbounded instruction bandwidth and branch prediction omniscience is assumed. In a real machine however, these factors have a major effect on performance.

In certain circumstances, the location from which to fetch instructions may not be known. For example, in machine architectures which permit a jump to a location stored in a register, the jump cannot be performed until the register value is known. The machine must stall issue from the time the jump is issued until the register value is known. We define this condition as a lack of instruction pointer availability.

Lack of instruction availability will reduce the number of instructions that can be issued. This can arise because of instruction cache misses, memory bank conflicts, etc. We point out that there is a dichotomy between instruction traffic and data traffic in Von Neumann machines. It is possible to buffer against data cache misses. For example, we can allow a subsequent memory load to access the data cache even after a prior memory load has missed the cache (lock-up free). There is no corresponding way to deal with instruction cache misses. This condition was not modeled in our simulations.

Branch misprediction effectively causes issue stalls. From the time a mispredicted branch is issued until it is determined that it has been mispredicted, all instructions issued after the branch must be discarded. From the point of view of being able to retire instructions, this is equivalent to having stalled instruction issue after the mispredicted branch instead of predicting it. There are two remedies for this effect. First, reduce the frequency of misprediction by either eliminating branches or by using a better branch prediction algorithm. Second, reduce the latency of branch issue to branch execute. This has the effect of reducing the number of instructions discarded after a mispredicted branch.

The effect of branch misprediction is clearly visible in figures 4, 5, 6, 7, and 11. It is most notable in gcc (figure 6) for the 8F3M machines: the performance of the 8F3M machine with omniscient branch prediction is almost 50 percent greater than the 8F3M.RB machine.

Finally, full window effects arise when *instruction lifetime* exceeds the window size. Instruction lifetime is measured as the time from issuance of an instruction into the window until it is removed from the window by retirement. At a minimum, instruction lifetime is equal to its latency. This time can be increased if the instruction must wait for operands after it is issued. This effect can be minimized by either having the compiler attempt to schedule code so that operands are likely to be available as an instruction is issued or by increasing the window size. The effect of window overflow can be seen in figures 4-12 at small window sizes. Once the window size increases beyond some point, other factors such as static issue efficiency, branch misprediction losses, etc. dominate.

However, there are code fragments for which code reorganization or window size increases will have little effect. Consider the Livermore Loops kernel #5:

```
DO 10 i = 2,n
    X(i)= Z(i)*(Y(i) - X(i-1))
10  CONTINUE
```

This simple recurrence relation will require at least two loads, a floating point subtract, a floating point multiply, a store, and an increment, compare and branch: perhaps 8 instructions per iteration. On a superscalar degree 8 machine, it takes just one clock to issue each iteration. However, each iteration will take a minimum of 12 clocks to execute using the latencies given in Table 1. Since each iteration is dependent on the prior iteration, a full window condition will quickly arise. For large 'n', the number of iterations, the machine will achieve steady state at an issue rate of one clock out of twelve. Inside this loop, the dynamic execution efficiency δ is only .08! The only remedy for such situations is to reduce the latencies of the executed instructions or recode the loop. The doduc and fpppp benchmarks exhibit this type of latency induced problem.

5 Implementation Issues

Several aspects of implementing a superscalar machine based on RDF principles are not trivial. Among these difficult issues are instruction delivery and multiple simultaneous data accesses.

Instruction delivery is the problem of determining what instructions must be fetched, fetching them, and delivering them to the proper function units for execution. Aside from bandwidth issues, branches and machine issue restrictions (e.g., only one floating point multiplier issue per clock) make instruction delivery difficult. We briefly address these issues here.

Given a fetch address, the technique as proposed in [1], that of fetching several cache lines simultaneously, is a good way of providing the raw bandwidth necessary to issue several instructions in one clock. For example, for a superscalar degree four engine, assuming that each cache line holds four instructions, fetching two sequential cache lines guarantees that four instructions can be delivered regardless of the alignment of the fetch address within the first cache line.

Once a group of instructions has been fetched, the difficulty lies in quickly determining where to fetch the next packet. Issue constraints determine how many instructions can be issued, and thus what the next fetch address should be. These constraints, however, are not resolved until after decode. Furthermore, the presence of branches in the packet determine the end of the packet as well as a possibly non-sequential next address. The branch needs to be predicted and the branch target calculated in time to perform the next fetch. These problems can be circumvented through the use of a decoded instruction cache (DIC) [10, 3]. Rather than caching undecoded instructions in an instruction cache, an instruction packet can be predecoded and stored in the DIC. The DIC, in addition to storing decoded instructions, can store branch prediction information, branch target addresses, and packet sizes.

Another implementation problem is that of providing multiple memory accesses per machine cycle. We briefly describe two techniques which can be used to provide the necessary memory bandwidth. First, a small fully associative cache backed by a conventional cache can be used. The small cache would be implemented with multiple access ports. Its small size would make the cost of the multiple ports less prohibitive. Another organization uses one, single ported cache per memory request unit. Standard cache consistency techniques can be used to keep the caches consistent with each other in the presence of memory stores.

6 Concluding Remarks

This paper is only a beginning in the demonstrated viability of a single instruction stream processor capable of delivering execution rates in excess of the two instructions per cycle limit stated in [1, 2]. Our simulation results have shown performance up to more than 3 instructions per cycle on large integer applications on processors that are reasonable to implement today.

Note that, in the interests of producing data that is consistent with the software and hardware mechanisms available today, we have imposed limitations on the performance that can be achieved. When these limitations are removed, the performance of single instruction stream processors will improve substantially.

For example, we have required that instructions be removed from the window in whole packet increments. Packets are removed only in the order in which the packets were issued, and only after all instructions in the packet have completed execution. In this way, completed instructions consume valuable window space while waiting to be removed. With a different implementation of checkpointing, it is possible to eliminate this inefficiency and make better use of the window, thus improving performance.

Second, we have assumed only one branch per cycle, and no instructions following the branch being issued in the same cycle with the branch. This limits the available parallelism to the average size of a basic block. Allowing multiple branches per cycle would increase the amount of available parallelism.

Furthermore, our results of 2.0 to 5.8 instructions per cycle come from a restricted data flow engine that has a limited window size and issue rate, consistent with what is reasonable today. As levels of integration and bandwidth capabilities increase, window sizes and issue rates will increase correspondingly. In the limit, our unbounded window size and issue rate machine (the UDF) shows instructions per cycle in the 17 to 1165 range While we are not suggesting that this is possible (yet), we expect numbers well in excess of 5 instructions per

cycle.

Finally, and perhaps most importantly, it is worth re-emphasizing that all of our results have been obtained using compilers not optimized for superscalar issue. With architecture-specific compiler assistance to perform code motion to provide higher issue density, performance can be expected to increase further. With compiler support to produce larger granularity execution units [12], performance should increase still further.

The bottom line is that processors can be implemented today that deliver more than twice the performance suggested in [1, 2], and the limits to what will be deliverable tomorrow by single instruction stream processors is still an open question.

References

[1] M.D. Smith, M.Johnson, and M.A. Horowitz, "Limits on Multiple Instruction Issue", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (April 1989), pp.290-302.

[2] N.P. Jouppi, and D. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (April 1989), pp.272-282.

[3] Y.N. Patt, W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction.", *Proceedings of the 18th Annual Workshop on Microprogramming*,(December 1985), pp.103-108.

[4] Y.N. Patt, W. Hwu, and M. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture.", *Proceedings of the 18th Annual Workshop on Microprogramming*,(December 1985), pp.109-116.

[5] Y.N. Patt, and W. Hwu, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality.", *Proceedings of the 13th Annunal Symposium on computer Architecture*, (June 1986), pp.297-307.

[6] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-order Execution Machines", *IEEE Transactions on Computers*, (December 1987), pp.1496-1514.

[7] W.M. Johnson, "Super-Scalar Processor Design", Technical Report No. CSL-TR-89-383, Stanford University, (June 1989).

[8] Norman P. Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance", *IEEE Transactions on Computers*, Vol. 38, No. 12, (December 1989), pp.1645-1658.

[9] G. S. Sohi and S. Vajaperyam, "Instruction Issue Logic for High-Performance Interuptable Pipelined Processors.", *Proceedings of the 14th Annual Symposium on Computer Architecture*, (June 1987), pp. 27-34.

[10] D. R. Ditzel, A. D. Berenbaum and H.R. McLellan, "The Hardware Architecture of the CRISP Microprocessor.", *Proceedings of the 14th Annual Symposium on Computer Architecture*, (June 1987), pp. 309-319.

[11] Tse–Yu Yeh, "Adaptive Training Branch Prediction ", Technical Report, University of Michigan, (1991).

[12] Steve Melvin and Yale Patt, "Exploiting Fine–Grained Parallelism Through Combined Hardware and Software Techniques", *Proceedings of the 18th Annual Symposium on Computer Architecture*, (May 1991)

[13] Robert Colwell, Robert Nix, John O'Donnell, David Papworth, and Paul Rodman, "A VLIW Architecture for a Trace Scheduling Compiler.", *IEEE Transactions on computers* , (August 1988), 967-979.

[14] Alexandru Nicolau and Joseph Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures", *IEEE Transactions on computers C-33, 11*, (November 1984), 968-976.

[15] Edward M. Riseman and Caxton C. Foster, "The Inhibition of Potential Parallelism by Condition Jumps.", *IEEE Transactions on computers C-21, 12*, (December 1972), 1405-1411.

[16] 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 Transactions on computers C-21, 12*, (December 1972), 1293-1310.

[17] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions", *IEEE Transactions on computers C-19, 10*, (October 1970), 889-895.