Design of A Fault-Tolerant Microprocessor: A Simulation Approach

Kab Joo Lee
Samsung Electronics Inc.
Kihung, Kyungkido
South Korea
phone: (0331) 209-3036 / fax: (0331) 209-4920
flyingv@samsung.co.kr

Gwan Choi
Department of Electrical Engineering
Texas A&M University
College Station, TX 77843
phone: (409) 845-7407 / fax: (409) 845-2630
choi@sypar.tamu.edu

Abstract

This paper presents an approach for assessing the merits and the cost of incorporating processor-level error detection and recovery mechanisms. The approach is exemplified by implementing several fault-tolerant mechanisms into a 32-bit, MIPS R3000-compatible, RISC microprocessor and conducting simulation-based fault injection experiments. The mechanisms are triple modular redundancy (TMR), retry on duplication-comparison, and retry on parity-checking codes. Reliability gains and performance/area overheads are quantitatively evaluated for each error-detection/recovery scheme. The fault injection analysis results indicate that the highest fault coverage is achieved with the code-based retry technique.

1 Introduction

Hardware implemented fault-tolerant mechanisms can enhance the dependability of safety-critical systems by introducing real-time or near-real-time recovery at the component level. In an operating environment where a high transient-fault rate is expected, such as in space, fault containment through use of hardware fault tolerance at the component level can economically improve the system reliability. Such low-level approach can minimize expensive software-driven, system-level rollback recovery. A difficult issue in designing reliability however is quantifying the cost-benefit in design alternatives. Fault-injection simulation approaches are often used to evaluate the performance of error detection, recovery, and error containment mechanisms [1], [2], and [3]. Some of these approaches offere a high-degree of experimental controllability and observability and many evident contributions are made. However, no CAD environment integrates fault-injection simulation to evaulate the successes and weak-links of a design in a closed-loop fashion. Difficult challengies in realizing a true-design-phase simulation analysis are: 1) issuing a dynamic simulation analysis of a design description, 2) assessing realistically the feature performance and reliability, and 3) commencing the design process with the analysis results. Also, for an unbiased evaluation of fault-tolerant mechanisms, faults need to be injected into every node location in a circuit. For each location, a number of fault injection needs to be made.

This paper presents a novel simulation-based design environment wherein a simulation-guided design and an exhaustive cost-benefit analysis is possible. The environment supports both Verilog or VHDL (structural/behavioral/and mixed) hardware description formats. The core of environment is a fast fault-injection simulator that makes possible exhaustive design evaluation. The results from the simulation analysis are used to determine the quantitative merits of each fault tolerant mechanism. We illustrate the design/analysis environment by synthesizing a 32-bit RISC microprocessor (based on MIPS R3000 architecture). Hardware-implemented fault-tolerant techniques are incorporated into two main sections in the microprocessor: datapath and control unit [4], [5], [6]. The resulting designs are dedicated fault-tolerant microprocessors with hardware features that automatically detect and recover from stuck-at and transient faults. In our design approach, fault-tolerant techniques are applied to key functional units of a microprocessor to minimize the design verification overhead.

A comparison of three component-level error detection and recovery schemes is conducted: triple-modular-redundancy (TMR), code-based recovery, and duplex-detection/retry. Each scheme is incorporated into the target microprocessor design and evaluated through simulation-based fault injection experiments. Reliability gains and performance/area overheads are quantitatively evaluated for each error-detection/recovery scheme. Section 2 describes the target system, Section 3 presents the design and analysis, and Section 4 presents the results. Section 5 contains conclusion.

0-8186-8212-4/97 $10.00

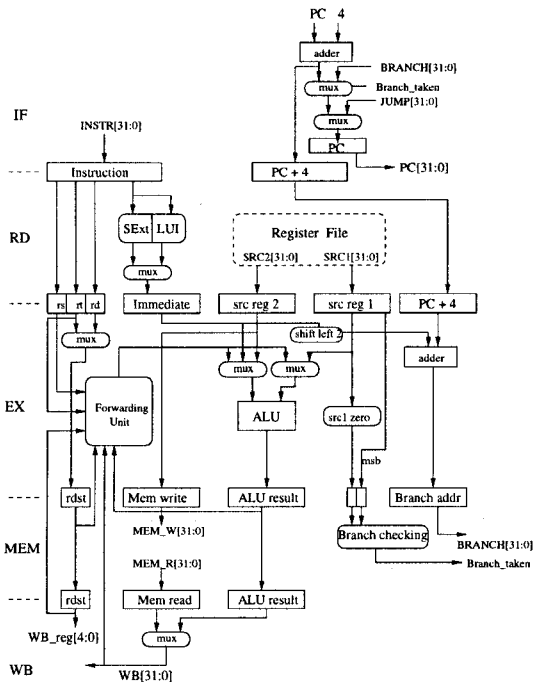

Figure 1. Pipeline Structure of the RISC32

2 Target Design

The target system is a 32-bit, 5-stage-pipelined RISC microprocessor, instruction-compatible with MIPS R2000/3000 microprocessor. We refer this system as *RISC32*. The five pipeline stages consist of: instruction fetch(IF), decode and register file read (RD), arithmetic and logic operations (EX), data cache access (MEM), and register file write back (WB). The MIPS architecture suits our design study because of the simplicity of its core datapath design and the ease of design expansion for reliability. Fig. 1 shows the pipeline structure of the target system. The control signals are not shown in this figure. The instruction set is divided into load/store instructions, arithmetic instructions (ALU immediate), arithmetic instructions (3-operand, R-type), shift instructions, multiply/divide instructions, jump/branch instructions, coprocessor instructions, and special instructions. Coprocessor instructions and special instructions are not included in the simulation model. The MIPS R2000/3000 CPU instruction details are described in [7].

Data and control hazards in the pipeline are resolved by hardware. The execution and write back stage data dependencies are resolved by the data forwarding unit. The load data hazard is detected by hardware and the pipeline is stalled until the dependency is resolved. A delay slot is used for jump instructions. The instruction immediately following the jump is in the load delay slot. The instruction in the delay slot always executes before the jump actually occurs. *Branch not taken* paradigm is used for branch prediction A branch instruction is determined to be taken in the memory access stage (MEM). If the branch is taken, the pipeline stages IF, RD, EX, are flushed and the instructions from the branch target address are fetched.

The simulation model of the target system includes Verilog structural design (gate-level) description of the pipeline data-flow components and a behavioral model of decoding and pipeline control, register file, memory system, and shift/multiplier/divider units. The components in the structural model are logic gates, latches, carry-lookahead adders, multiplexers, and functional units such as ALU and operand forwarding unit. The behavioral models are written in C and integrated into the simulator developed for fault injection experiment [8] during the compilation.

Three fault-detection/recovery techniques that improve fault containment by inclusion of hardware enhancement are investigated. Each technique is expressively designed for tolerating the most common faults, transient faults. These techniques are: (1) triple modular redundancy (TMR), (2) duplication-comparison based instruction retry, and (3) parity-code based instruction retry. The following subsections describe the details of each design approach:

Mechanism A : Triple Modular Redundancy: A TMR design [9], e.g., triplication of the ALU module with majority voting of the outputs, is considered. The advantage of TMR design is that it provides fault-tolerance without the need for separate detection and recovery functions; TMR approach provides real-time masking of the faulty-module outputs. Also TMR is faster than the code-based techniques [10] since the voting speed is independent of the information bit length because the voting is always performed on three bits. However, the design overhead includes two extra functional units and a combinational voter.

Mechanism B : Duplication-Retry: An instruction retry scheme based on ALU duplicationcomparison is also considered. The ALU in the pipeline EX stage is duplicated and the same input values are applied to both ALUs. The ALUs performs computations in parallel and the results of those computations are compared. In an event of a disagreement, an ALU error message is generated. When the error signal is asserted, the value of pipeline registers are held for one clock cycle, and the ALU computation is retried in the next clock cycle. Since the ALU operands are forwarded from MEM and WB stages, the data in those two pipeline registers have to be preserved as well as those in the IF and RD stage pipeline registers.

Mechanism C : Parity-Retry on PC: The third fault-tolerant feature considered is a parity-checked Program Counter(PC) unit. Corruption in the PC unit could result in the following situations:

1. Illegal address error; results in misaligned fetch, refer-

ence to supervisor address space.

2. Code-space reference; results in control-flow corruption, illegal address, illegal opcode, execution timeout.

3. Out of the address space reference; results in illegal opcode, decoding error.

The fault-tolerant PC unit consists of a 32-bit parity-checked adder, and other supporting recovery structures. The predicted value of output check bit is computed from the values of input check bits. Then the add operation is checked by comparing this predicted value of the check bit with the one obtained from the output [11]. In an event of disagreement, a parity error signal is generated. The error signal is used as a pipeline control to hold the pipeline register values of the IF stage for one cycle, resulting in retrying the IF stage.

3 Design Evaluation

This section describes the simulation methodology, the workload, and the selection of fault injection parameters including the fault model, fault locations, and fault injection time.

3.1 Simulator

To accurately characterize fault handling behavior of each fault-tolerant mechanisms, a novel simulator is developed. Fault propagation in a system can result in a corruption of program counter which deviates the program-control flow path or a functional retry that requires the simulation to back up several clock cycles. In order to follow the correct (either faulty or none-faulty) control flow of the machine, the simulation engine is designed to take input dynamically from a set of inputs supplied to the simulator in parallel. The dynamic control-flow allows user to capture different error propagation scenarios like system time-outs, retry, and program deviation. The simulator engine also performs the bit-wise parallel simulation [12] of the multiple input streams to capture a number of possible fault behavior in a single simulation pass.

3.2 Workloads

Four integer programs, two matrix computation programs and two sorting programs, are selected as target programs. The four workloads are: (1) a sorting program that uses the bubble sorting algorithm (Bubble), (2) a sorting program that uses the quick sorting algorithm (Quick), (3) a matrix addition program that adds two matrices (Madd), (4) a matrix multiplication program that multiplies two matrices (Mmul). These workload programs are written in C, and the gcc compiler is used to generate the MIPS assembly code. The MIPS simulator SPIM S20 [13] then generates a trace of the MIPS instructions. The SPIM S20 is used because it translates assembler pseudo-instructions to MIPS instructions when it uploads the program and data in memory. The memory-mapped MIPS instructions are taken as the workloads in our experiments. The workload characteristics are listed in the Table 1.

Test Programs	Madd	Mmul	Bubble	Quick
code size (bytes)	372	432	384	568
static data (bytes)	32	32	44	44
stack (bytes)	68	76	24	225
execution (cycles)	296	588	2863	2271

Table 1. Workload program characteristics

3.3 Fault Model, Fault Injection Time and Locations

The fault model selected for simulation experiments is a gate-level, transient, single bit-flip fault model. Although device-level fault injection is a more accurate approach to the actual single event upset (SEU) phenomenon than flipping logic of gate outputs [14] and [2], the time required for the electrical-level fault simulations are prohibitive for a large number of faults on a relatively large circuits. Also, as the transistor feature size decreases, and the clock speed continues to improve, transients produce an erroneous value at a gate output for the entire clock period, even longer. With the logic-level bit-flip duration set to a single clock cycle, the results obtained from our simulation experiment can serve as the worst case scenarios of the SEU phenomenon. The single bit-flip fault is used in our experiment although the simulation supports multiple bit faults; the fault dictionary approach [14] show that over 92% of logical error pattern are single bit-flips. Also, we limit our experiments to the transient faults, since over 85% of failures in computer systems are known to have attributed by transients [15].

We inject the bit-flip faults into all node locations in the circuit. Studies [2], [1] which evaluate error detection mechanisms have injected faults for the locations having a higher likelyhood of resulting in an error. Although such biased approach exercises error detection mechanisms effectively, the overall fault tolerance of the system, i.e., faults can occur at any location in the system, is difficult to obtain. Hence, we subject all locations assuming that each location has the same probability of having a fault. This not only eliminates the bias for the selection of fault locations, it also enables us to compare different design schemes. Fault injection times are also uniformly distributed over the workload execution time.

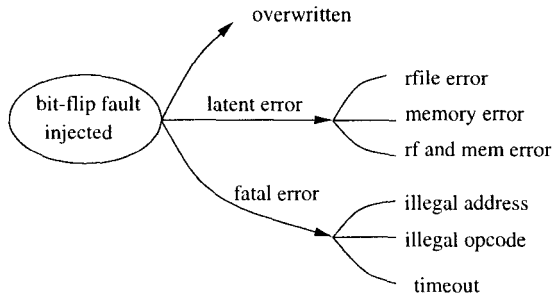

Figure 2. Error Behavior

3.4 Error Behavior and Classification

Error manifestation are classified as illustrated in Fig. 2. The outcome of each fault injection simulation is: overwritten, latent, or fatal. A fault injection simulation ends abnormally when an injected fault results in a fatal error. The fatal errors monitored are illegal address, illegal opcode, and execution timeout. There are two possible simulation outcomes when a fault simulation ends normally without causing any fatal errors: (1) a bit-flip fault is overwritten, thus the results are correct, (2) a bit-flip fault causes errors that are still latent after the simulation ends normally, the result may or may not be correct. In the second case, the simulation outcome are compared to those of fault-free simulation. It is possible for programs to complete and the results be incorrect without causing a fatal error. It is also possible for programs to complete and the results be correct, however there may be other latent errors resident in the register file, memory, or both in the register file and memory.

Address error, illegal opcode error, and execution timeout are considered exceptions in our simulation experiments. Upon monitoring of these errors, program control incurring the exception terminates.

4 Analysis

This section includes the results from the simulation experiments. The four selected workloads are executed on each processor model, and the impact of faults on processor reliability and program behavior is analyzed. Section 4.1 presents results of the error propagation in each microprocessor model. The results include the number of overwritten, latent, and fatal errors. Also, fault coverage gains by using each of the three fault tolerant mechanisms are evaluated and compared in Section 4.1. Section 4.2 presents the overhead associated with each mechanism in terms of gate count. Also, the cost-benefits related to each mechanism are summarized from the data in Section 4.1. Section 4.3 presents the simulation time taken to perform the fault injection simulations.

4.1 Fault Coverage Evaluation

Three fault injection simulations are performed for each location of fault injection area in order to approximately characterize the workload behavior. The time for each fault injection is randomly determined over the workload execution time. A total of 46248, 75828, 80982, and 49728 fault injections are carried out for RISC32, Mechanism A, Mechanism B, and Mechanism C, respectively. Overall, a total of 252,696 fault injection simulations are performed. Each injection results in either a flip-to-1 (*ft1*) or a flip-to-0 (*ft0*) fault.

Table 2 shows the total number of overwritten, latent, and fatal errors for each target system. Approximately 69% of all injected faults are overwritten in RISC32. Among the three fault-tolerant mechanism, the Mechanism A shows the most reliability improvement. Over 84% of injected faults resulted in overwritten errors using Mechanism A in RISC32. The proportion of fatal error dropped down to approximately 12%, which is about a 50% gain over RISC32 in this error category. A gain of 56% is observed in the latent error category (the proportion of the latent error is reduced from 8.5% to 3.7%.) Mechanism B improves the reliability of the RISC32 a little less than Mechanism A for fatal error category. Approximately 13% of injected faults lead to fatal errors (a gain of 44% over RISC32), and 3.2% for latent errors. The Mechanism B experiment results in a similar proportion of overwritten errors 84.2%. The reliability gains using Mechanism C are relatively small but very cost effective. Approximately 78% of injected faults are overwritten. The proportion of fatal error is reduced to 14.4% and latent error to 7.4%, a gain of 37% and 13% over RISC32, respectively. This is because protecting the program counter unit prevented the processor from having fatal error such as address errors and illegal opcode errors.

Initial observation of data indicates that: (1) a large percentage of the injected transient faults remains latent in memory, (2) there is a strong relationship between latent memory and register-file errors, and 3) the probability of error propagation only to register file remaining latent is very small (2% to 6%). Consequently, the probability of having a memory error is high when a register-file error occurs, because the corrupted register file values are used in subsequent computations. The percent-chance of having a memory error when a register-file corruption occurres are 94%, 88%, 85%, and 94% for RISC32, Mechanism A, Mechanism B, and Mechanism C, respectively,

It is also clear how effective each mechanism against fatal errors. In Mechanism A and B, the pipeline ALU is protected. Thus, using Mechanism A or Mechanism B would reduce the number of fatal errors resulting from faulty ALU outputs (i.e., the ALU computes memory address for load/store instructions and data values to be writ-

Design	Error Percentage(%)					Total
	ErrClass	Madd	Mmul	Bubl	Quck	Error
Base	fatal	23.5	22.6	17.8	27.0	22.7
	latent	11.4	10.3	10.0	2.3	8.5
	masked	65.1	67.1	72.2	70.7	68.8
A	fatal	12.0	11.5	9.8	13.5	11.7
	latent	4.8	4.5	4.4	1.2	3.7
	masked	83.2	84.0	85.8	85.3	84.6
B	fatal	12.9	12.5	10.3	14.9	12.6
	latent	3.8	3.8	4.0	1.2	3.2
	masked	83.3	83.7	85.7	83.9	84.2
C	fatal	15.2	14.1	9.1	19.1	14.4
	latent	9.5	9.3	8.8	1.9	7.4
	masked	75.3	76.6	82.1	79.0	78.2

Table 2. Total number of errors: (a) RISC32 - 46248 injections, (b) Mechanism A - 75828 injections, (c) Mechanism B - 80892 injections, (d) Mechanism C - 49728 injections

ten in write-back registers.) Both Mechanism A and Mechanism B decrease the number of illegal address errors, 4080 to 2920 (20% reduction) for Mechanism A, 3802 (7% reduction) for Mechanism B. The number of timeout errors is also decreased from 1980 to 1489 (25% reduction) for Mechanism A and 1866 (6% reduction) for Mechanism B. The number of illegal opcode errors is not affected [1] since an ILL-OP error occurs either when the program counter gets corrupted or when an error is propagated to the pipeline RD stage.

Mechanism C (retry on parity-checked PC unit) is effective for reducing the number of all three fatal errors, especially the illegal opcode errors. The number of illegal opcode errors is decreased to 1611 (64% reduction). The number of illegal address errors is dropped from 4080 to 3683 (10% reduction), Also, there is about 6.4% decrease in the number of timeout errors.

4.2 Overhead

Table 3 shows the gate count [2] of the structural descriptions of each pipeline design. The gate count of behavioral models are not estimated, thus not included in this figures. Mechanism A (ALU in TMR configuration) cost 64% gate count overhead. It is interesting to see that Mechanism B (retry based on ALU duplication-comparison) has more overhead, 72%, than Mechanism A. This is because of the multiplexers added to the pipeline registers in the entire pipeline datapath. Mechanism C (retry based on parity-checked PC unit) cost only 7% gate overhead. In this approach, multiplexers are added only to the registers in between the IF and the RD stages.

Design	Baseline	A	B	C
gate count	3958	6492	6812	4248
gate overhead	1.0	1.64	1.72	1.07

Table 3. Gate count

Mechanism A exhibited least timing overhead. For TMR approach, two gate delay (due to majority voting logic) and interconnection delay between ALUs and the voter is added to the EX pipeline stage. The delay overhead associated with Mechanism B is from the comparator for ALU otuputs, which is implemented in a five-level XOR tree. In addition, the error detection signal has to travel through the pipeline to select the multiplexer inputs for pipeline registers. In Mechanism C, delay overhead due to the parity checking circuitry has to be taken into account during the pipeline design.

4.3 Simulation time

The simulator was compiled and run on two different machines, a Sun SPARC 4 workstation and an SGI Power Challenge 10000 XL machine with a shared memory parallel machine based on the MIPS R10000 processor. It took, on the average, 68 seconds to perform one fault injection simulation, resulting in 72.8 hours to run fault injection experiment for all locations, for RISC32 model on the Sun Sparc 4 workstation. Also, it took approximately 115, 108, and 74 seconds to perform one fault simulation for Mechanism A, Mechanism B, and Mechanism C, respectively. Note that fault simulation for Mechanism A took more time than that of Mechanism B. This is because fatal errors were more often monitored for Mechanism B causing the early termination of the fault simulation. The exhaustive simulation took about 73 hours (3 days and 1 hour) for RISC32 using a Sun Sparc 4 workstation. It took approximately 202 hours (8 days and 10 hours) to perform exhaustive fault simulation for Mechanism B and Mechanism C; the simulation had to be terminated several times due to the unstability of simulation host.

5 Conclusions

Three fault-tolerant mechanisms are evaluated and compared. These mechanisms are triple modular redundancy (TMR), retry on duplication-comparison, and retry on

[1] The number of illegal opcode errors was slightly increased due to the random selection of fault injection times.

[2] The primitive gates are NOT, AND, OR, NAND, NOR, XOR, XNOR, DFF

parity-checked PC unit. Each fault tolerant mechanism was incorporated into a 32-bit RISC architecture and simulated using a gate-level pipeline model written in Verilog HDL. Transient faults were injected into the gate outputs of the pipeline structure, and their impacts to the processor was monitored and analyzed. Key results obtained from the simulation experiments are:

- The results showed that retry on parity-checked PC unit was most cost-effective in reliability gains among the tested mechanisms. The gate count was increased by 7%, and the probability of having fatal errors were decreased 37%.
- In our experiment, fault tolerance through error masking was more cost-effective than two step, error detection and recovery, fault tolerance technique. The TM-Red ALU had less cost and more fault coverage than retry on ALU duplication-comparison. This suggests that the cost associated with error recovery, not only error detection, has to be carefully taken into account during the system validation.

These results aids in understanding the cost-benefits of different component-level fault-tolerant mechanisms in microprocessors. We have limited the fault injection area to the structural pipeline models. Faults are not injected in behavioral models, i.e., faults in some of these units (especially the control) would affect the fault-tolerant mechanisms, reducing the coverage. Investigation of effects of faults in the behavioral models remains as the future work.

6 Acknowledgment

This work was supported partially by the Advanced Research Projects Agency under grant DABT63-94-C-0045 and Texas Higher Education Coordinating Board under grant ATP-999903-100. The findings, opinions, and recommendations expressed herein are those of the authors and do not necessarily reflect the position or policy of the United States Government, State of Texas, or Texas A&M University and no official endorsement should be inferred.

References

[1] E. W. Czeck and D. P. Siewiorek, "Effects of Transient Gate-level Faults on Program Behavior," *Proc. FTCS-20*, pp. 236-243, Chapel Hill, NC, 1990.

[2] Gwan S. Choi and R. K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis," *IEEE Trans. on Computers*, Vol. 41, No 12, pp. 1515-1526, December 1992.

[3] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson and J. Karlsson, "Fault Injection into VHDL Models," *Proc. FTCS-24*, pp. 66-75, Austin, TX, 1994.

[4] I. Elliott and I. Sayers, "Implementation of 32-bit RISC Processor Incorporating Hardware Concurrent Error Detection and Correction," in *IEE Proceedings*, Vol. 137, Pt. E, No. 1, Jan. 1990.

[5] Y. Tamir and M. Tremblay, "High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback," *IEEE Trans. on Computers*, Vol. 39, pp. 548-554, April 1990.

[6] M. Nicolaidis, "Evaluation of a Self-Checking Version of the MC 68000 Microprocessor," *Microprocessing and Microprogramming*, Vol 20, pp. 235-247, 1987.

[7] G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice-Hall, 1992.

[8] Kab Joo Lee and Gwan Choi, "Fast Simulation For Fault-Handling Evaluation", Technical Report, KLGC001, Dept. of Electrical Engineering, Texas A&M Univ., October 1995.

[9] J. von Neumann, "Probabilistic Logics and Synthesis of Reliable Organisms from Unreliable Components," Automata Studies in *Annual of Mathematical Studies*, No. 34, pp. 43–98, Princeton University Press, 1956.

[10] E. Fujiwara and D. K. Pradhan, "Error-Control Coding in Computers," *IEEE Computer*, pp. 63–72, July 1990.

[11] F.F. Sellers, M.Y. Hsiao, and L.W. Bearnson, "Error Detecting Logic for Digital Computers," McGraw-Hill, New York, 1968.

[12] H. Fujiwara, "Logic Testing and Design for Testability," Cambridge, MA, The MIT Press, 1985.

[13] J. Larus, "SPIM S20: A MIPS R2000 Simulator," Technical Report, University of Wisconsin-Madison, Madison, Wisconsin, August 1992.

[14] Gwan S. Choi, R. K. Iyer and D. G. Saab, "Fault Behavior Dictionary for Simulation of Device-level Transients," in *Proc. IEEE ICCAD-93*, pp. , November 1993.

[15] H. Ball and F. Hardy, "Effects and Detection of Intermittent Failures in Digital Systems," *Proc. 1969 FJCC, AFIPS Conference Proceedings*, vol. 35, pp. 329-335, 1969.