Error Detection by Duplicated Instructions in Super-Scalar Processors

Nahmsuk Oh, *Member, IEEE*, Philip P. Shirvani, *Member, IEEE*, and Edward J. McCluskey, *Life Fellow, IEEE*

***Abstract*—This paper proposes a pure software technique "Error Detection by Duplicated Instructions" (EDDI), for detecting errors during usual system operation. Compared to other error-detection techniques that use hardware redundancy, EDDI does not require any hardware modifications to add error detection capability to the original system.**

EDDI duplicates instructions during compilation and uses different registers and variables for the new instructions. Especially for the fault in the code segment of memory, formulas are derived to estimate the error-detection coverage of EDDI using probabilistic methods. These formulas use statistics of the program, which are collected during compilation. EDDI was applied to eight benchmark programs and the error-detection coverage was estimated. Then, the estimates were verified by simulation, in which a fault injector forced a bit-flip in the code segment of executable machine codes. The simulation results validated the estimated fault coverage and show that approximately 1.5% of injected faults produced incorrect results in eight benchmark programs with EDDI, while on average, 20% of injected faults produced undetected incorrect results in the programs without EDDI. Based on the theoretical estimates and actual fault-injection experiments, EDDI can provide over 98% fault-coverage without any extra hardware for error detection. This pure software technique is especially useful when designers cannot change the hardware, but they need dependability in the computer system.

To reduce the performance overhead, EDDI schedules the instructions that are added for detecting errors such that "Instruction-Level Parallelism" (ILP) is maximized. Performance overhead can be reduced by increasing ILP within a single super-scalar processor. The execution time overhead in a 4-way super-scalar processor is less than the execution time overhead in the processors that can issue two instructions in one cycle.

***Index Terms*—Concurrent error detection, error detection by duplicated instructions, fault-injection experiment, fault tolerance, instruction-level parallelism, single event upset, super-scalar processor and instruction-scheduling, transient fault.**

ACRONYMS[1]

CI	comparison instruction
CFCSS	control-flow checking by software signatures
ED	error detection
EDDI	ED by duplicated instructions
FC	fault coverage
HamD	Hamming distance
ILP	instruction-level parallelism
ISA	instruction-set architecture
MI	master instruction
RESO	re-computing with shifted operands
SBB	storeless basic block
SEU	single event upset
SI	shadow instruction
UB	upper bound
VLIW	very-long instruction word
branch-I	branch instruction
normal-I	normal instruction
store-I	store instruction

Manuscript received November 1, 1999; revised May 23, 2000. This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, U.S. Office of Naval Research under Grants N00014-92-J-1782 and N00014-95-1-1047.

The authors are with the Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA (e-mail: {NsOh; Shirvani; EJM}@crc.stanford.edu).

Publisher Item Identifier S 0018-9529(02)02607-6.

[1]The singular and plural of an acronym are always spelled the same.

I. INTRODUCTION

PERMANENT faults or transient errors in a computer system can

- affect the control flow of a program;
- change the system status; or
- modify the data stored in memory.

If the system does not perform some run-time checking, an erroneous output might not be detected and might be used as a correct output. Therefore, in highly reliable and dependable computing, it is important to monitor the program to detect any abnormality in the system and take appropriate actions to avoid incorrect outputs.

On the other hand, trends in processor architecture have shown an increasing use of ILP to improve performance. In addition to pipelining individual instructions, it has become very attractive to fetch multiple instructions at the same time, and execute them in parallel to use functional units whenever possible. This form of ILP is called super-scalar execution. It provides a way to exploit available hardware resources in the system.

However, the limitation of ILP in a program prevents full use of resources, and consequently, some functional units are idle during execution. References [1] and [2] show that the performance gain is not proportional to the maximum number of instructions that are issued simultaneously, because of the limitation of ILP in a single thread of control flow. Based on the data in [2], even in an ideal machine with infinite machine resources, perfect branch prediction, and ideal register renaming, there is no appreciable improvement in speedup beyond four instruction issues per cycle as shown in Fig. 1. Therefore, if one cannot simultaneously execute more than four instructions in one clock cycle but there are enough resources, then one can use most of the idle resources for ED.

Fig. 1 shows the distribution of available instruction-level parallelism and speedup under ideal super-scalar execution. The

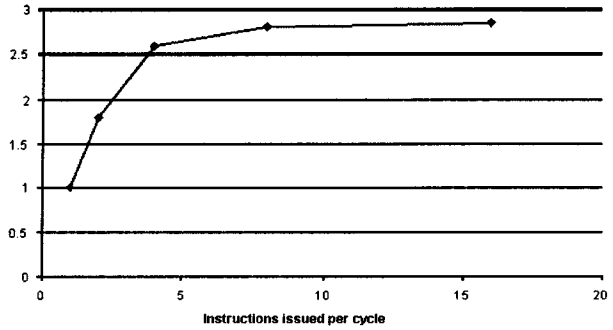

Fig. 1. Estimated speedup under ideal super-scalar execution.

graph is cited from [3] and shows the average of speedups reported in [2] for several benchmarks.

This paper focuses on concurrent ED by exploiting ILP of super-scalar architectures. Duplicated instructions in EDDI do not affect the results of the program, but detect errors in the system during run time. The instructions for ED are scheduled to use idle resources as much as possible, in order to reduce performance overhead in super-scalar architecture.

The SEU is one of the major sources of bit-flips in memory [4]. A bit-flip is an undesired change in the state of a memory cell; an SEU can cause the state of a memory cell to change from 0 to 1 or from 1 to 0. It can modify an instruction of the program, or can corrupt data stored in memory. EDDI can detect faults that can be modeled as bit-flips in the memory. For example, when an instruction is fetched from the memory, 1 bit of the data bus might get complemented due to a transient fault, and then modify the opcode field of the instruction. This error can be modeled as a bit-flip in the code segment in memory and can be detected by EDDI.

"Transient errors in functional units, control logic, address buses, and data buses" can also change the intermediate value of the computation and result in incorrect outputs. A program might deviate from its correct instruction sequence, e.g., due to a fault in a branch-I, resulting in a control-flow error. These errors can be detected by EDDI.

Section II addresses related works. Sections III and IV present the algorithm; the detailed analysis of the algorithm is in the Appendix. Section V estimates the FC, based on the MIPS architecture, during compile time. Section VI verifies the results by simulation. Section VII compares EDDI with other duplication methods.

II. Related Work

A traditional concurrent ED technique is to use massive redundancy. N modular hardware redundancy [5] and N version programming [6] are examples of massive redundancy, but these techniques incur $100(N-1)\%$ area or performance overhead. To reduce this overhead, system-level error checking methods such as designing self-checking programs [7], running a separate task for error checking [8], or using a watchdog processor [9], [10] have been proposed.

While Blough [11], [12] presented and analyzed a general procedure for ED in a complex system (data block capture and analysis monitoring process), several signature monitoring techniques have been developed; they focus on checking control flow errors in the program. "Signature monitoring" is an analysis in which a signature associated with a block of instructions is calculated and saved somewhere during compile time; the same signature is generated during run-time and compared with the saved one. This technique includes:

- Structural Integrity Checking [13],
- Path Signature Analysis [14],
- Signatured Instruction Streams [15],
- Asynchronous Signatured Instruction Streams [16],
- Continuous Signature Monitoring [17], [18],
- extended-precision check-sum method [19],
- On-line Signature Learning and Checking [20].

Most signature monitoring techniques still need dedicated hardware, such as a watchdog processor, to calculate the run-time signatures and to compare them with the saved signatures. On the other hand, CFCSS [21] is a pure software technique for checking control flow errors. Signatures are embedded into the program during compile time as part of the instructions and a run-time signature is generated, and compared with the embedded signatures when the instructions are executed.

To decrease hardware cost, time redundancy has received attention. Time redundancy methods include

- alternating logic [22],
- alternate-data retry [23],
- data complementation [24],
- RESO [25],
- time redundancy in neural networks [26].

Time redundancy reduces the amount of extra hardware at the expense of additional time. If the system can complete its computations before its specified time limit, the extra time can be used for ED. The basic concept of time redundancy is to repeat computations in such a way that errors can be detected by comparing computation results. If the errors detected are transient, they can be tolerated by performing the computations again. Executable assertions are another pure software method known to be very effective in program testing, validation, and fault tolerance [8], [27]–[29]. Block Signature Self-checking [30], Block Entry Exit Checking [31], and Error Capturing Instructions [30] are examples of pure software methods for error checking. However, time redundancy techniques usually suffer from execution time overhead and reduce performance.

Researchers have attempted to exploit the unused resources of systems for concurrent error checking. In a multitasking and multicomputer environment, idle computers are used to execute replicated tasks for error checking [32]. Application of RESO [25] to the Cray-1 has been studied using unutilized resources employing machine parallelism [33]. Using the spare capacity in super-scalar and VLIW processors to tolerate functional unit failures has been proposed [34]. These two techniques use idle resources in the system but need to modify the original processor architectures. Conversely, Available Resource-driven Control-flow monitoring (ARC) [35] does not require incorporating a specialized monitor and does not modify the original processor architecture. It is applied to a VLIW machine and exploits ILP for integrated control flow monitoring, but it detects only control flow errors.

The idea of duplicate instructions in a VLIW processor was investigated [36]. It assumes fault-free memory operations and

control operations, and duplicates only the ALU instructions to detect errors in data paths. For the duplicated ALU instructions, it uses the same source-registers. The advantage of using the same source-registers is that register pressure is kept low. However, if the source-registers of the operation are corrupted, then the error cannot be detected. EDDI does not assume any fault-free operations. It duplicates the variables and data-structures, and uses different registers for the duplicated instructions; therefore, EDDI can also detect the memory operation errors. And it interleaves the duplicated instructions in such a way that most of the control flow errors are detected.

This paper explores the use of idle resources in super-scalar processors for concurrent error checking. Unlike previous techniques, EDDI is a pure software method; thus, it is not necessary to modify the original processor architecture. The targets of EDDI are

- inter-block and intra-block control flow errors;
- data or code change in memory;
- transient errors in functional units.

Most instructions for ED are scheduled to use idle resources exploiting super-scalar architecture. They do not depend on original instructions, thus more ILP is added to the original program to maximize resource use. Programs with EDDI show high fault-secure capability: a fault in the system either has no effect on the output or an error signal is reported.

III. ED By Duplicated Instructions

A. Preliminaries

Duplicated Instructions in EDDI have no effect on the result of the program, but do detect errors in the system during runtime. The purpose of EDDI is to check any deviation from the anticipated behavior of the computer system, and to detect an error caused by faults introduced into the system.

The basic idea of error-detecting instructions is "duplication of original instructions in the program" but with different registers and variables. A MI is the original instruction in the source code; a SI is the duplicated instruction added to the source code. General purpose registers and memory are partitioned into two groups for MI and SI. The registers and memory for MI should always have the same value as the registers and memory for SI. Thus, if there has been a mismatch between a pair of registers for MI and SI, an error can be detected by comparing these two register-values. A CI compares the values of the two registers, and invokes an error handler if they do not match.

A simple example of source code is

$$\text{ADD R3, R1, R2} \quad ; \text{R3} \leftarrow \text{R1} + \text{R2}.$$

This is an addition instruction; the first operand is the target register; the second and third operands are the source registers. Thus register R3 should take the sum of the values in registers R1 and R2. The corresponding SI and CI are

```
ADD   R3,   R1,   R2          ; MI
ADD   R23,  R21,  R22         ; SI
BNE   R3,   R23,  gotoError   ; CI.
```

Let registers R1, R2, R3 be the master registers, and R21, R22, R23 be the shadow registers that contain the same value as R1, R2, R3, respectively. If the values in R1 and R21 are the same and the values in R2 and R22 are also the same, then the two sums in R3 and R23 should be equal. Otherwise, an error has occurred during the operations. A conditional branch-I BNE R3 R23 gotoError (branch to gotoError if R3, R23 are different) performs a comparison, and transfers control to an error handler if an error has occurred.

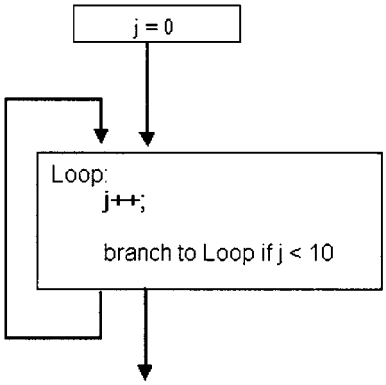

Fig. 2. Branching at the end of the loop.

Where should this CI be inserted? Let the comparison be performed immediately before storing register-values in memory, or before deciding the direction of a branch-I or jump-I. Registers are used to hold temporary values for computing, and the results of computation are stored in memory to free up the registers for later use. Only compare final calculation results that will be stored in memory for later use. There is no need to compare intermediate computation results that propagate and corrupt final results. If an error changes an intermediate result, the effect of the change probably propagates down to the final computation and corrupts the final result; if not, i.e., the error is masked out in intermediate computation, then the final result can be used as a correct answer. Thus, there are different final computation results from MI and SI, and the error can be detected by comparing the master and shadow registers.

On the other hand, registers often hold values to resolve branching directions. Fig. 2 is a typical example.

After executing instructions in the block, the branching direction is determined by the value of the variable j. If a register (master register) holds a value for this j and is corrupted while executing instructions, it will produce an unanticipated result. Thus, to detect erroneous branching, the value in the master register should be compared with the value in a shadow register that holds the same value as j.

A "basic block" is a branch-free sequence of instructions; no jumps into or out of the block except for the first and last instruction of the block. An SBB is a sequence of instructions in which there is no store-I except for the last instruction which can be a store-I or a branch-I. Fig. 3 is an example. An SBB is always inside a basic block. Within an SBB, the SI are scheduled to maximize resource use by attempting to use idle resources, which are not used by MI. If the last instruction of an SBB is a store-I, then a CI is placed before the store to compare the master and shadow register values that are stored in memory. Scheduling of SI is discussed in detail in Section IV.

```
ADD   R3,  R2,  R1   }
SUB   R3,  R3,  R4   }  storeless basic block
ST    0(SP),  R3     }
AND   R1,  R1,  R2   }
SUB   R2,  R2,  R1   }  storeless basic block
ST    0(SP),  R2     }
...
```

Fig. 3. SBB example.

B. Algorithm for EDDI

Notation (Also used in Section IV and Appendix):

I_i	instruction $\#i$ of the program $i = 1, 2, \ldots N$
$I_{k,i}$	instruction $\#i$ of SBB $\#k$
N	number of instructions in the program
V	$\{I_j;\ j = 1, 2, \ldots, N\}$ (nonempty)
E	$\{(I_i, I_j);\ i, j = 1, 2, \ldots, N\}$
G_D	dependency graph: an ordered tuple (V, E)
n_k	number of instructions in SBB $\#k$
V_k	$\{I_{k,j};\ j = 1, 2, \ldots, n_k\}$: set of n_k instructions to be scheduled in SBB $\#k$
E_k	$\{(I_{k,i}, I_{k,j});\ i, j = 1, 2, \ldots, n_k\}$: represents dependencies in SBB $\#k$
$G_{D,k}(V_k, E_k)$	a subgraph of G_D, representing dependencies within the SBB $\#k$; the $G_{D,k}$ form a partition on G_D
l	number of instructions in the program for which an illegal branch to these instructions will be detected
D_k	$\{d_j;\ j = 1, 2, \ldots, n_k\}$: set of instruction execution delays
T_k	$\{t_j;\ j = 1, 2, \ldots, n_k\}$: the start-time for the instructions, *viz,* the cycles in which the instructions start
n_{res}	resource types
q_{reg}	register use
q_{st}	$\lvert\{I_j: I_j \in S, \Psi\}\rvert/\lvert A\rvert$ $\Psi \equiv$ data stored by I_j are used later by subsequent instructions
F_k	$V_k \rightarrow \{1, 2, \ldots, n_{\text{res}}\}$: mapping of the instructions to the unique n_{res} that they use.

An I_j depends on an I_i, if I_j uses the result of I_i; thus I_j is executed after the completion of I_i. The dependency between I_i and I_j is denoted by a directed edge (an ordered pair): (I_i, I_j).

EDDI Algorithm:

1) Build a G_D of the program.
2) For each $G_{D,k}$
   2.1 Build a graph $H_{D,k}(V'_k, E'_k)$

$$V'_k \equiv \{I'_{k,j}\text{: a SI of } I_{k,j} \in V_k\}$$
$$E'_k \equiv \{(I'_{k,i}, I'_{k,j})\text{: } (I_{k,i}, I_{k,j}) \in E_k\}$$

   2.2 If the last instruction of the block $I_{k,n}$ is a store-I or a conditional branch-I, then
   Create a CI $I_{k,c}$
   Create an ordered edge $(I_{k,n}, I_{k,c})$ and $(I'_{k,n}, I_{k,c})$ to connect $H_{D,k}$ to G_D.
   End_If
   2.3 Schedule $I_{k,i}, I'_{k,i}, i = 1, 2, \ldots, n_k$, and $I_{k,c}$ according to the List Scheduling Algorithm in Section IV.

End_Algorithm

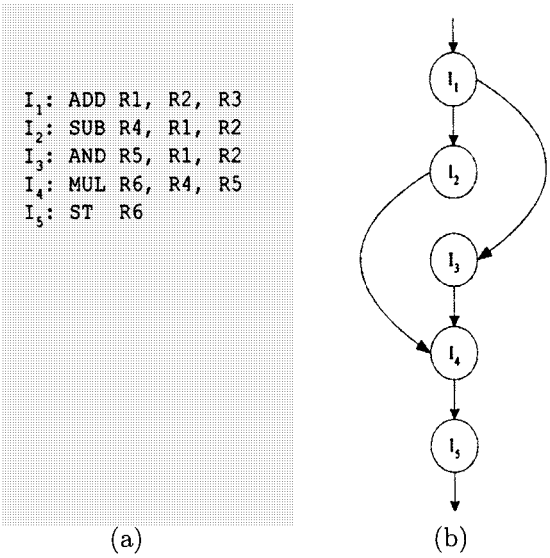

Fig. 4. An SBB. (a) Instructions in one particular SBB. (b) Dependency graph of this SBB.

Example 3A: Demonstration of the EDDI Algorithm: Fig. 4(a) is a sequence of instructions in one particular SBB of the program; Fig. 4(b) is $G_{D,k}$ (a part of G_D), and represents dependencies among instructions within this SBB. For example, 2 edges, (I_2, I_4) and (I_3, I_4) imply that I_4 needs the results of I_2 and I_3. Fig. 5(b) illustrates how to construct a $H_{D,k}$ from this SBB and attach it to G_D.

The portion of G_D for the SBB in Fig. 4 is shown in Fig. 5(a). Item 2.1 in the EDDI algorithm builds the $H_{D,k}$ in Fig. 5(b). The vertices in $H_{D,k}$ are SI $(I'_1, I'_2, I'_3, I'_4, I'_5)$ corresponding to MI $(I_1, I_2, I_3, I_4, I_5)$ in G_D. The dependencies in $H_{D,k}$ are inherited from the dependencies in $G_{D,k}$; thus the edges in $H_{D,k}$ show the same dependencies among SI as those of MI. After constructing $H_{D,k}$, a CI, I_c, is added to $G_{D,k}$ because I_5 and I'_5 are store-I, and the registers should be compared before storing the results in memory. I_c compares the results of I_5 and I'_5, i.e., its result depends on I_5 and I'_5; thus two directed edges (I_5, I_c) and (I'_5, I_c) are added, connecting $H_{D,k}$ to G_D. Now, G_D represents a new relation between MI, SI, and CI. These instructions are scheduled in item 2.3 of the EDDI Algorithm. The scheduled instructions are:

I_1 :	ADD	R1, R2, R3
I_2 :	SUB	R4, R1, R7
I'_1 :	ADD	R21, R22, R23
I_3 :	AND	R5, R1, R2
I'_2 :	SUB	R24, R21, R27
I_4 :	MUL	R6, R4, R5
I'_3 :	AND	R25, R21, R22
I'_4 :	MUL	R26, R24, R25
I_c :	BNE	R6, R26, go_to_error_handler
I_5 :	ST	R6
I'_5 :	ST	R26.

Scheduling algorithms are discussed in detail in Sections IV-A–C.

End of Example 3A.

The computational complexity of the algorithm depends on which scheduling algorithm is chosen in 2.3. The complexity

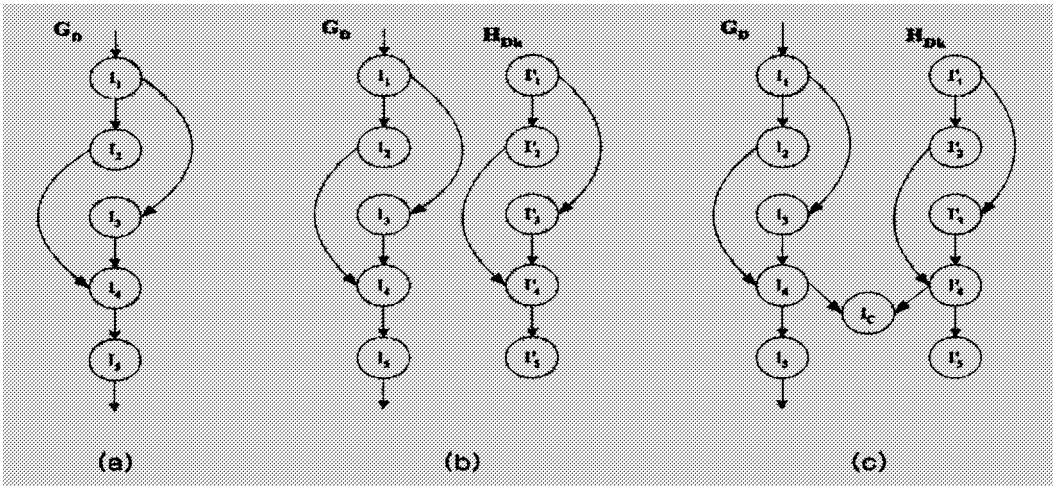

Fig. 5. Example 3A demonstrating the EDDI algorithm. (a) Dependency graph; (b) constructing $H_{D,k}$; (c) adding $H_{D,k}$ to G_D.

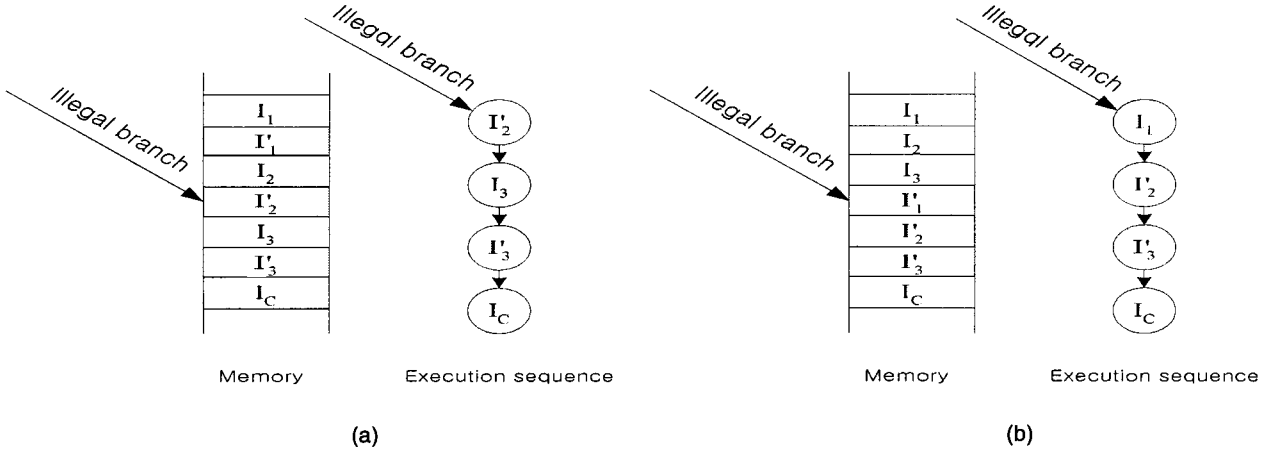

Fig. 6. Shadow instructions. (a) SI interleaved with MI in a SBB, (b) SI grouped together and placed after MI.

of building the dependency graph in line 1 is $O(n^2)$ [37, p. 321] and building $H_{D,k}$ is $O(|V| + |E|) = O(n)$; thus, if a heuristic scheduling algorithm is chosen that has complexity, $O(n)$ (such as the list scheduling algorithm in Section IV-B), the complexity of the entire algorithm is $O(n^2)$. However, if the exact optimal solution is desired, and the integer linear programming for scheduling instructions is used, the complexity of the entire algorithm is dominated by the complexity of the scheduling algorithm, which is exponential.

The correctness of the algorithm and FC are discussed in detail in the Appendix. Basically, an error caused by the faulty instruction propagates through the dependency graph to the CI, which detects the mismatch between the master and shadow pair.

Control-flow errors are detected if the control flow is transferred to the place from which there exists a path in G_D to a CI and the number of MI executed is not equal to the number of SI executed on this path.

If the number of executed MI is different from the number of executed SI, one of a pair (pairs) of MI and SI is not executed while the other is. The result propagates along the dependency path in G_D and the error is detected when a CI is executed to compare the results of the MI and SI.

For example, suppose the SI are scheduled to interleave with MI in an SBB as shown in Fig. 6(a). An illegal branch transfers the control flow to the instruction I_2': I_2' is the first instruction after the illegal branch. When the control reaches I_c, then I_2 was not executed while I_2' was executed (I_2 is not on the execution sequence) and the results compared by I_c are different. However, if an illegal branch jumps to one of I_1, I_2, I_3, it is not detected because the number of executed MI and SI are still the same. On the other hand, an illegal branch to I_2, I_3, I_1', I_2', or I_3' is detected in Fig. 6(b) because in the execution sequence, the number of MI and SI are always different. Only an illegal branch to I_1 is not detected.

In Fig. 6(a), $l = 3$, and in Fig. 6(b), $l = 5$. For simplicity, let $N = 7$ in the above example and let an illegal branch jump to one of the N instructions. Then the probability of missing a control flow error is

$$P_C = \Pr\{\text{miss a control flow error}\} = \frac{N - l}{N}.$$

This probability is 2/7 for the scheme in Fig. 6(b) and is much lower than the one in Fig. 6(a) which is 4/7. The detailed analysis is in the Appendix.

IV. Scheduling Instructions Exploiting Instruction-Level Parallelism

The SI should be scheduled with MI to maximize ED coverage and to minimize time overhead using idle resources. Resource-constrained scheduling is a well-known intractable problem. First the scheduling of MI and SI is formalized; then an exact solution method (linear integer programming) and a heuristic algorithm (list scheduling) for solving this problem are described.

A. Integer Linear Programming

A resource-constrained scheduling problem is one where the number of resources of any given type m is bounded by a set of integers $\{a_m;\ m = 1, 2, \ldots, n_{\text{res}}\}$.

A formal model of the resource-constrained scheduling problem is achieved by using binary variables with two indexes j, l:

$X_k\{x_{j,l};\ j = 1, 2, \ldots, n_k;\ l = 1, \ldots, \lambda\}$
$\lambda \equiv$ a UB on the latency of SBB $\#k$.
$x_{j,l}$ is a binary variable.
$x_{j,l} \equiv 1$ when and only when $I_{k,j}$ begins in step l of the schedule.

Then, the scheduling can be formulated as the set of equations, (1)–(5), that must be satisfied. Equations (1)–(4) are the resource-constrained minimum latency scheduling equations and are described in detail in [38].

Equation (1) shows that the start time of each instruction is unique.

Equation (2) shows that the dependency relations in $G_{D,k}$ must be satisfied.

The resource constraint must be satisfied at every cycle.

An $I_{k,j}$ is executing at cycle l when $\sum_{m=l-d_j+1} x_{j,m} = 1$.

Equation (3) shows that the number of all instructions at cycle l of type k must be $\leq a_k$.

One more condition is added in (5); it requires that the "number of MI" > "number of SI" until step $\lambda - 1$ in SBB $\#k$.

The "number of MI" = "number of SI" at step λ when scheduling is completed.

Linear Integer Programming Algorithm: Objective function: minimize $\sum_j t_j$ such that constraints (1)–(5) are satisfied

$$\sum_l x_{j,l} = 1, \qquad j = 1, 2, \ldots, n. \tag{1}$$

$$\sum_l l \cdot x_{i,l} - \sum_l l \cdot x_{j,l} - d_j \geq 0$$
$$i, j = 1, 2, \ldots, n_k\text{: } (I_{k,i}, I_{k,j}) \in E_k. \tag{2}$$

$$\sum_{j:F(I_j)=m} \sum_{q=l-d_j+1} x_{j,q} \leq a_m$$
$$m = 1, 2, \ldots, n_{\text{nes}},\ l = 1, \ldots, \lambda. \tag{3}$$

$$x_{j,l} \in \{0, 1\}, \qquad j = 1, 2, \ldots, n_k,\ l = 1, \ldots, \lambda. \tag{4}$$

$$\sum_{i:\text{MI}} \sum_{q=1}^{l} x_{i,q} - \sum_{j:\text{SI}} \sum_{q=1}^{l} x_{j,q} > 0$$
$$l = 1, \ldots, \lambda - 1. \tag{5}$$

```
End_Algorithm
```

B. List Scheduling

The disadvantage of the integer linear-programming formulation is the computation complexity of the problem: generally it is NP-complete [38]. The number of variables, and the number of inequalities and their tightness, affect the ability of computer programs to find a solution. Heuristic algorithms have been developed to solve this intractable problem. Consider the List-Scheduling algorithm [39] to schedule MI and SI.

```
List-Scheduling Algorithm
m_i, s_i  ≡ number of MI, SI up to instruction
  #i within SBB #k.
ReadyList = instructions with 0 predeces-
  sors in G_{D,k}
Loop until ReadyList is empty
  Let x be the instruction* in ReadyList
  with highest priority
  Schedule x as early as possible within
  the three constraints:
    dependencies in G_{D,k}
    resource constraints
    If x is not the last instruction, Then
  m_i − s_i ≠ 0, End_If
  Update predecessor count of x's suc-
  cessor nodes
  Update ReadyList
  End_Loop
  End_Algorithm
```

* The priority list for instruction x can be the maximum execution delays from x to any instruction, resource requirements, the program source code order, or arrangement of MI and SI, depending on heuristic urgency measure.

The heuristic nature of list scheduling might not find an exact optimum solution, but the computational complexity of this List-Scheduling algorithm is $O(n)$. In general [39, p. 211], solutions of list scheduling do not differ much in latency from the optimum ones, for those (small) examples whose optimum solution are known.

C. Interleaved Scheduling

A super-scalar processor with out-of-order execution capability can: 1) dynamically schedule the instructions by solving dependencies using hardware, and 2) simultaneously issue multiple instructions in 1 clock-cycle. To reduce the cost of static scheduling discussed in Sections IV-A–B, the dynamic scheduling capability of the processor can be used. A super-scalar processor has a buffer, instruction-window, in which it decodes and places the fetched instructions. At the same time, it examines instructions in the window to find instructions that can be issued in the same cycle. The instruction window serves as a pool of instructions; and the more ILP exists in this pool, the more instructions can be executed concurrently. Therefore, if putting as many MI and SI as possible in the instruction window, gives more chance to execute more instructions at the same time. Instructions should be arranged such that control flow ED cov-

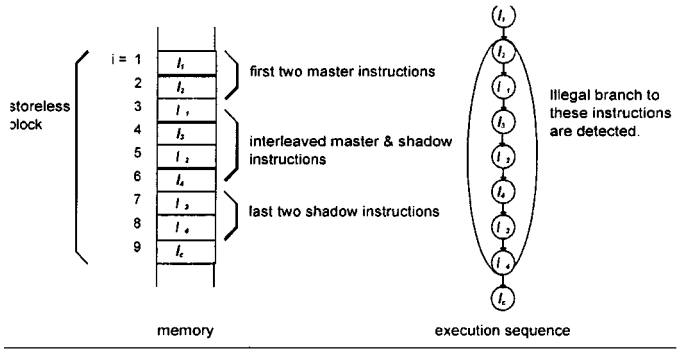

Fig. 7. MI and SI arrangement to detect a control-flow error.

erage is maximized. The proposed scheme for achieving these two objectives together is shown in Fig. 7.

If MI is interleaved with SI, then the same number of MI and SI can be put in the instruction window. Because there is no data dependency between the MI and SI, this scheme provides maximum ILP in the instruction window. However, as in the Fig. 6(a) example, a simple interleaving scheme misses half of the control-flow errors because an illegal branch to MI is not detected. Instead, if two MI are placed in the first two slots of the SBB, and MI and SI are interleaved in the middle, and two SI are placed in the last two slots, then the condition $m_i - s_i \neq 0$ in the Algorithm can be satisfied until the last SI in the block. For the example in Fig. 7, $m_i - s_i > 0$ is true until $i = 7$ and $m_i - s_i \neq 0$ is satisfied. Therefore, an illegal branch to instruction I_2, I_3, I_4, I_1', I_2', I_3', or I_4' is detected because, after the illegal branch, it fails to execute one of the pair of MI and SI on the execution path to a CI.

Hardware dynamically schedules the instructions in the instruction window finding available instructions that can be issued at the same cycle. In this scheme, the scheduling to use idle resources is done by hardware while control-flow ED is achieved by arrangement of instructions.

V. Estimated FC

Based on probability methods, formulas are derived for estimating the ED coverage of EDDI. These formulas and their derivation are in the appendix. This technique can provide the ED coverage when EDDI is implemented in the program during compilation. This estimate is very useful to predict the ED coverage before actually running the program, especially in computers for space applications.

The estimated FC can be obtained from (13) in the Appendix. Eight benchmark programs were used: FFT, matrix multiplication, Fibonacci number (numeric programs), Hanoi, Compress, Shuffle, Quick sort, and Insert sort (nonnumeric programs). The estimation method was applied to these benchmark programs.

An original program is compiled by gcc [40] with level-2 optimization option (O2). Most compiler-optimization techniques that do not involve a space-speed tradeoff (such as loop unrolling and function inlining) are performed in level-2 optimization. The compiler was modified to allocate registers for MI and SI. First, the compiler produced the assembly code of the program. Then, SI and CI were added in assembly code by the post-processor; the resultant assembly-code with EDDI was compiled into object code by an assembler.

Based on the fraction of branch-I and store-I after EDDI is added to the programs, the probability of missing an error is calculated for the benchmark programs; Table I shows the results.

The parameters described in the appendix are determined as follows. In the MIPS Instruction Set Architecture II [41], the immediate fields for the target address of branch-I and jump-I take 16 bits and 26 bits of the 32-bit instruction, respectively. Based on these numbers, it is assumed that:

$$|W_{\text{off}}|/n_b = 0.50 \text{ for branch-I,}$$
$$|W_{\text{off}}|/n_b = 0.81 \text{ for jump-I.}$$

Let $q_{\text{st}} = 0.95$: 95% of the stored data are used later in the program execution. This is a reasonable number because most of the data in these programs are repeatedly used during execution.

Let $q_{\text{reg}} = 0.90$: the register use is 90% because level-2 compiler-optimization tries to maximize the register use.

Table I shows the probabilities of each case in the appendix, and reports the FC of the technique. The HamD between branch-I and store-I is greater than 1; thus, $\Pr_u\{T_S, T_B\}$ and $\Pr_u\{T_B, T_S\} = 0$ and are omitted from the table.

The estimated numbers show that EDDI has about 98% to 99% ED coverage in 7 out of the eight benchmark programs. EDDI is a pure software method: it does not require the incorporation of a specialized hardware nor does it alter the original processor architecture; however, based on the results in Table I, it achieves high ED coverage as other hardware techniques do. In most of the benchmark programs, the $\Pr_u\{T_B, T_N\}$ and $\Pr_u\{T_B, T_B\}$, which are related to control-flow errors, constitute the larger part of the undetected errors. CFCSS [21] technique is a signature monitoring method that can be used to lower these probabilities, but it increases the performance-overhead by adding signature-checking instructions to the original program.

VI. Experimental Results

The same eight benchmark programs in Section V are chosen for the experiment to verify the estimated coverage that is calculated Section V.

The source files are compiled, and the target machine codes are generated without ED instructions. A fault injector forced 1 bit-flip in the code segment of the machine code. For each iteration of this simulation, the location of the bit-flip is determined by a random number generator. This machine code is executed on SGI Octane that uses the 4-way super-scalar R10000 MIPS processor. Table II shows the result of 500 iterations.

The numbers in row 2 of the table (incorrect result) indicate the number of faults that cause the programs to produce "incorrect results that look like correct ones to the observer." Row 3 means "that erroneous result is repeatedly produced" because the fault creates an infinite loop in the program. In row 4, the processor does not respond to the observer; thus one must manually stop the processor. The number in row 5 shows the fraction of faults that are detected by the operating system in the machine. A segmentation fault and failed assertion are examples of faults detected by the operating system. Sometimes, despite the bit-flip in the code segments, the programs produce correct results; row 6 shows the fraction of these cases. This case can

TABLE I
ESTIMATED FC FOR BENCHMARK PROGRAMS

	Branch-I	Store-I	$P_u(T_S,T_S)$	$P_u(T_N,T_B)$	$P_u(T_B,T_N)$	$P_u(T_B,T_B)$	FC
FFT	8.2%	5.4%	0.0044	0.0003	0.0015	0.0056	98.8%
Hanoi	11.3%	13.9%	0.0113	0.0006	0.0021	0.0078	97.8%
Compress	10.6%	11.1%	0.0089	0.0014	0.0020	0.0143	97.3%
Quick sort	12.0%	6.8%	0.0056	0.0006	0.0022	0.0151	97.7%
Fibonacci	17.3%	2.0%	0.0008	0.0016	0.0032	0.0447	95.1%
Insert sort	13.6%	5.7%	0.0046	0.0010	0.0026	0.0151	97.7%
Matrix mul	10.3%	4.5%	0.0036	0.0008	0.0019	0.0066	98.7%
Shuffle	6.5%	8.2%	0.0066	0.0004	0.0012	0.0081	98.4%

TABLE II
RESULTS OF FAULT INJECTION INTO ORIGINAL PROGRAMS

	FFT	Hanoi	compress	quick-sort	Fibonacci	insert-sort	matrix mul	shuffle
Incorrect result	38%	6%	17%	16%	16%	13%	25%	7%
Infinite erroneous result	2%	3%	5%	1%	0%	0%	0%	0%
Processor hung	6%	1%	2%	0%	2%	1%	2%	0%
Detected by OS	24%	41%	36%	42%	53%	45%	48%	31%
Correct result	30%	49%	40%	41%	35%	41%	25%	62%
Total	100%	100%	100%	100%	100%	100%	100%	100%
Undetected incorrect output	46%	9%	24%	17%	18%	14%	27%	7%

TABLE III
RESULTS OF FAULT-INJECTION INTO THE PROGRAMS WITH EDDI

	FFT	Hanoi	compress	quick-sort	Fibonacci	insert-sort	matrix mul	shuffle
Incorrect result	1.8%	0.6%	1.4%	0.6%	1.0%	0.6%	1.6%	0.4%
Infinite erroneous result	0.4%	0.2%	0.2%	0%	0%	0%	0%	0.2%
Processor hung	0%	0.6%	0.2%	0.2%	0.4%	0.4%	0.2%	0.4%
Detected by EDDI	29.0%	15.8%	17.0%	22.6%	15.2%	19.8%	24.6%	32.6%
Detected by OS	22.0%	30.6%	32.4%	29.2%	23.8%	13.2%	27.4%	25.8%
Correct result	46.8%	52.2%	28.8%	47.4%	49.6%	52.6%	46.2%	40.6%
Total	100%	100%	100%	100%	100%	100%	100%	100%
Incorrect output undetected	2.2%	1.4%	1.8%	0.8%	1.4%	1.0%	1.8%	1.0%

happen, for example, when the bit-flip is in the unused field of an instruction and therefore has no effect on the results. Row 7 gives the total fraction of faults that produced incorrect outputs without being detected (the sum of rows 2, 3, and 4). On average, in the programs without EDDI, 20% of the injected faults produced incorrect outputs and were not detected, as shown in Table II.

EDDI is included in the eight benchmark programs by the compiler postprocessor that was developed. A bit-flip is injected into the generated machine code, then the corrupted machine code is executed. Table III shows the results for 500 injected faults.

Columns 1 and 3 in Fig. 8 illustrates the fraction of faults that produce undetected incorrect outputs in the original program and in the program with EDDI.

Column 2 shows the estimated fraction of faults that are missed by EDDI. EDDI shows high ED capability: approximately 98.5% of injected faults in most programs produced incorrect outputs without being detected. On the other hand, original programs without EDDI produced undetected incorrect results that average 20% of the total faults.

The experiment validated the estimated error coverage as illustrated in Fig. 8. The error coverage calculated in Section V is about 98.2% in eight benchmark programs; the error coverage in the experiment is about 98.5% in the benchmark programs.

Because extra instructions are added to the original assembly code, the program with EDDI suffers from "an increase in code size and loss of performance" compared to the original program. Fig. 9 shows the execution time overhead. In addition, because general purpose registers are used as shadow registers, more register spilling occurs with EDDI; more spilling causes more performance overhead because it increases the number of memory operations. For example, matrix multiplication spills more registers than Fibonacci. Matrix multiplication needs as many registers as possible to hold the matrix elements, but Fibonacci needs just a few registers to keep the last two Fibnonacci numbers.

Because instructions are duplicated, the overhead is anticipated to be 100%. However, as shown in Fig. 9, for most of the programs, the execution time overhead is less than 100%. The reason for this low performance overhead is because a pair of MI and SI are always independent of each other; thus, SI add more parallelism to the program and fill the empty slot of the pipeline. Therefore, higher use of the processor resources is achieved.

A 4-way super-scalar processor can exploit this parallelism better than a 2-way super-scalar processor. Fig. 9 shows this ef-

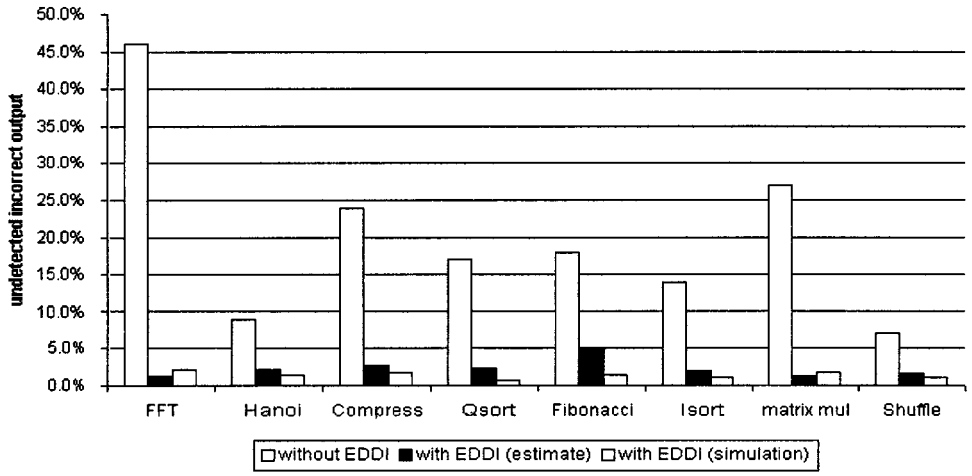

Fig. 8. Fraction of faults that produced incorrect outputs without being detected.

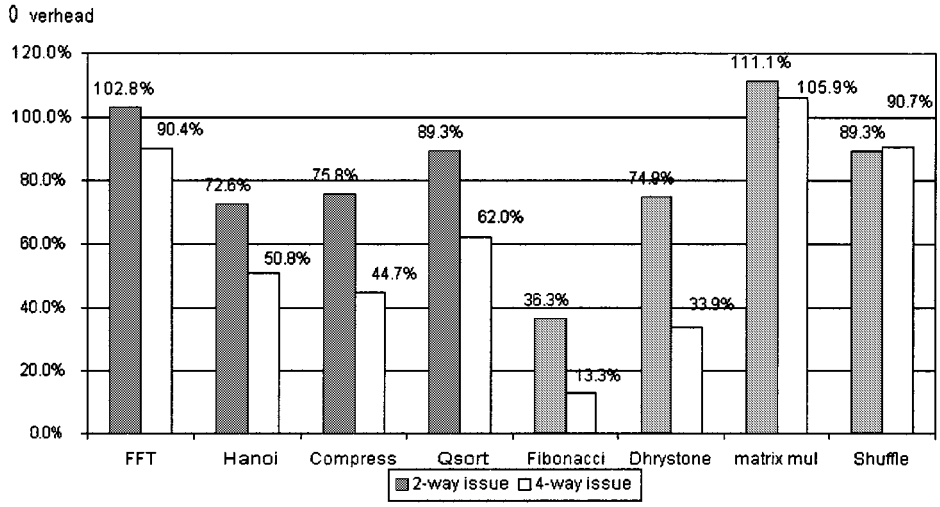

Fig. 9. Execution-time overhead in R4400 (2-way issue) and R10000 (4-way issue).

fect and the execution times for running the programs in two different processors. The execution-time overhead in the R10000 (a 4-way issue processor used in SGI Octane) is less than that in the R4400 (a 2-way issue processor used in SGI Indigo). EDDI has less execution time overhead in processors that use more aggressive super-scalar architectures.

VII. Comparison of Different Source-Level Duplication Methods

The technique in this paper duplicates the instructions in an assembly source code to achieve ED capability. An alternative way might be: duplicating source code in a high level language such as C or Pascal. All the variables, assignments, calculations as well as arguments, are duplicated to compare the pair of variables or results of calculations [42]. Consider two options for comparing the duplicated variables with the original variables in the source code:

- comparing the final outputs of the program,
- comparing the variables when they are updated.

The first has the advantage of minimum performance overhead compared to the second, because, in the first, the CI are placed only at the final output stage while, in the second, the CI are inserted whenever the variables are updated. However, comparing the values when they are updated has higher FC than comparing only final output results. For example, if a branch-I is corrupted and causes an infinite loop during the execution, the final output comparison cannot detect this error.

This section compares three techniques:

- assembly source code duplication (EDDI);
- C source code duplication with comparison after updating variables;
- C source code duplication with comparison of the final results.

Fig. 10 shows the execution time overhead of the three techniques on SGI Octane (R10000). Technique #3, comparison at the final stage, shows the best performance because it has the fewest CI. However, fewer CI has more chance of missing errors in the middle of execution.

Fig. 11 shows the FC of the three techniques. Technique #1, EDDI, shows better coverage than the others do. Technique #2, comparison after every update, has higher coverage than the technique #3, comparison at the final stage, but has lower cov-

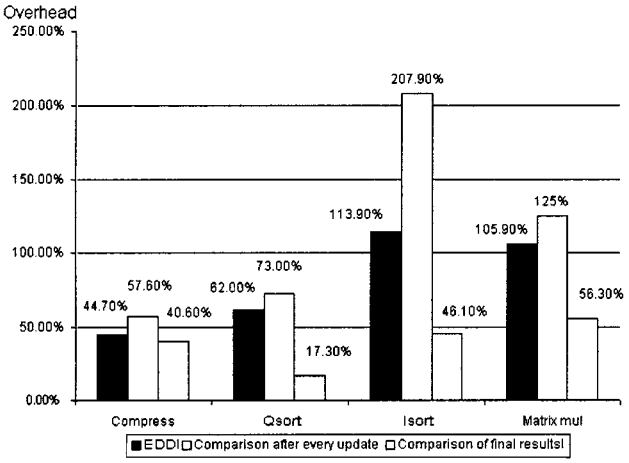

Fig. 10. Execution-time overhead in various levels of duplication.

erage than the EDDI because EDDI has finer grain ED capability in assembly level. The fine-grain comparison in EDDI results in lower ED latency. Consequently, it has higher chance of detecting faults that might cause cases of undetected errors such as infinite loop, hang-up, or illegal function call. These are cases other than errors that can propagate or get masked. Overall, EDDI shows the highest FC with medium performance overhead.

APPENDIX
ED COVERAGE ESTIMATE

Notation (also, see Section III-B.):

n_b	number of bits in one instruction
N	total number of instructions in the program
T_S, T_B, T_N	[store-I, branch-I, normal-I] type
$\tau(I)$	a function that maps instruction I to its instruction type
S	$\{I_j : \tau(I_j) = T_S, j = 1, 2, \ldots, N\}$
B	$\{I_j : \tau(I_j) = T_B, j = 1, 2, \ldots, N\}$
M	$\{I_j : I_j \notin B \cup S, j = 1, 2, \ldots, N\}$
W_{op}	set of bits in the opcode field of an instruction
W_{reg}	set of bits in the register field of an instruction
W_{off}	set of bits in the offset field of an instruction
$\overline{q}_x$	$1 - q_x$, for any subscript x
h_{bi}, h_{si}	number of [branch-I, store-I] in the ISA that have HamD 1 from instruction i
(T_A, T_B)	change of instruction from type T_A to type T_B
$\Pr_u\{(T_A, T_B)\}$	$\Pr\{$a fault induces (T_A, T_B), and it is not detected by EDDI$\}$

Any fault that can be modeled as a bit-flip in the data segment of memory can be detected by EDDI because there are always pairs of master and shadow variables in the data segment. The fault can be detected by comparing the two values of the master and shadow variables. However, it is somewhat difficult to determine the exact FC of this technique in the code segment of memory, because the effect of faults depends on the run-time behavior of the program. For instance, one cannot tell whether a conditional branch will be taken or not until it is actually executed with the run-time values of the condition variables. The ED coverage also depends on the input data sets, because the dynamic values of the registers and memory are strongly affected by the input data. However, based on the probabilistic method, the FC can be estimated. This appendix describes how to estimate and predict the FC before running the program.

A bit-flip can replace the original instruction with an incorrect instruction and cause abnormal behavior in the system. An upset in the opcode field of an instruction could change one operation to another operation, and produce an incorrect result. A bit-flip in an operand field of an instruction could change a register number causing the processor to perform an operation with an incorrect value. The goal is to detect these kinds of errors.

Definitions:

- normal-I: an instruction that is neither a branch-I nor a store-I
- HamD of two numbers a, b: $|a \oplus b|$.

Instructions in a RISC architecture generally have two or three fields in them as shown in Fig. 12; opcode, register, and constant or offset field. If the opcode field is corrupted, then an instruction is changed to another instruction either of a different or the same type. The former is the case of:

$$(T_N, T_S), (T_N, T_B), (T_S, T_N), (T_S, T_B), (T_B, T_N), \quad \text{or} \quad (T_B, T_S).$$

The latter is the case of:

$$(T_N, T_N), (T_B, T_B), \quad \text{or} \quad (T_S, T_S).$$

For example, an and instruction changed to a store-I is (T_N, T_S), and an and operation changed to an or operation is (T_N, T_N). If the other fields—not the opcode field—are corrupted, then the original operation of the instruction is preserved but the source or destination of the operation is changed. This is the case for

$$(T_N, T_N), (T_B, T_B), \quad \text{or} \quad (T_S, T_S).$$

One example of (T_N, T_N) is "and r2 r1 r3" changed to "and r7 r1 r3" where the destination register field is corrupted.

All 9 possible cases are discussed and analyzed to estimate error-coverage. The probability that an error occurs and escapes detection is calculated for each case; then FC is calculated by subtracting it from 1. There might be some events that are not considered when calculating the probability but they can be ignored when their probability is very low.

Cases 1: (T_N, T_N), (T_N, T_S) These cases are detected because there always exists a path in G_D from the faulty instruction to a CI. There always exists a MI and Si pair for normal-I. If an instruction is faulty, the result of executing two instructions would be different (if the results are same, they can be regarded as a correct result). The erroneous result propagates along the dependency path in G_D, corrupting results of the instructions in the path. The correct result of the other instruction of MI and SI pair also propagates along a path in G_D, and these two paths meet in the CI. The two different results are compared, and the error of the faulty instruction is detected. If an error is masked during the execution of instructions along the path, and does not corrupt the final result, then the two compared results are the same and they are regarded as correct.

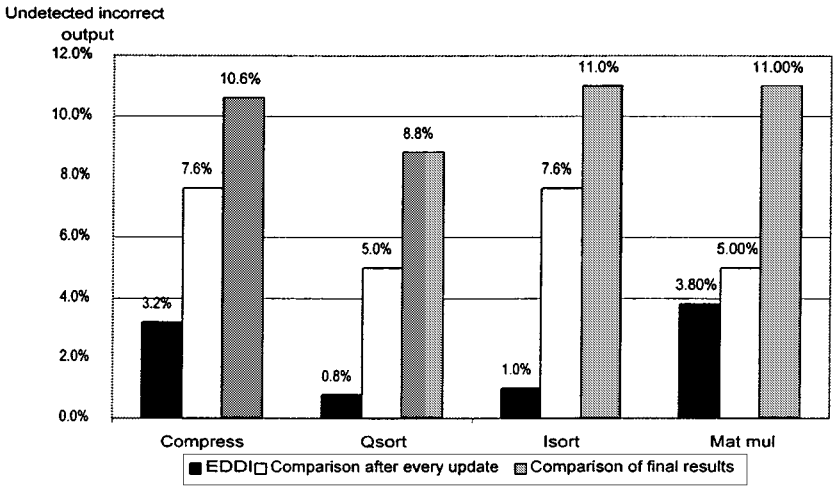

Fig. 11. Undetected incorrect outputs in various levels of duplication.

Case 2: $(T_S, T_S)(T_S, T_N)$*:* These cases are detected if the data stored by either of the corrupted or noncorrupted store-I in the MI-SI pair is used by a subsequent instruction. One of the master and SI pair stores a correct result in memory. If one of the MI and SI loads the incorrect value from memory while the other loads the correct value, the load instruction loading the incorrect value can be considered as a faulty normal-I. The error is detected as in case 1.

For (T_S, T_N), even if the incorrect stored data are not used later, this error is detected if the normal-I, which results from the error in the store-I, modifies a live value in a master or shadow register; otherwise, it is not detected. If this special case is ignored and it is assumed that the error is not detected only if the stored data are not used, then the probability of these two cases not being detected is (6).

Notation:

I_f the corrupted instruction

b_f the corrupted bit in I_f

$\overline{q}_{\rm st}$ ratio of stored data that are not used later in the program execution

k the ratio of possible normal-I, which can be created by a single bit-flip in the opcode field, to the total number of possible instructions in the ISA

$$\begin{aligned}\Pr\{(T_S, T_S)\} &\approx \Pr\{I_f \in S\} \cdot \Pr\{b_f \notin W_{\rm op}\} \cdot \overline{q}_{\rm st} \\ &= \frac{|S|}{N} \cdot \left(1 - \frac{W_{\rm op}}{n_b}\right) \cdot \overline{q}_{\rm st}; \\ \Pr\{(T_S, T_N)\} &\approx \Pr\{I_f \in S\} \cdot k \cdot \Pr\{b_f \in W_{\rm op}\} \cdot \overline{q}_{\rm st} \\ &= \frac{|S|}{N} \cdot k \cdot \frac{W_{\rm op}}{n_b} \cdot \overline{q}_{\rm st} \end{aligned} \tag{6}$$

The rest of the cases are related to control-flow errors; a normal-I or store-I is changed to a branch-I, or a branch-I is changed to another instruction.

Case 3: (T_N, T_B), (T_S, T_B)*:*

Notation:

E_h

event: HamD 1 from the branch-I

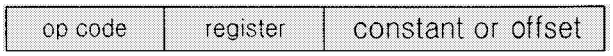

Fig. 12. Fields in instructions of RISC architecture.

For a single-bit error, instructions with E_h can be changed to a branch-I, thus this case of error is restricted to the instructions that have E_h. Normal-I and store-I always exist as a MI and SI pair. If one of the pair is changed to a branch-I and the other is executed, it is similar to case 1, and the error might be detected. However, if one of the pair is changed to a branch-I, and then causes an illegal jump to another location, but the other is not executed, then this error is undetectable. Therefore, if these cases are only detected when the number of executed MI and SI are different, the estimated probability is

$$\begin{aligned}&\Pr_u\{(T_N, T_B)\} + \Pr_u\{(T_S, T_B)\} \\ &\quad = \Pr\{E_h\} \cdot P_C \\ &\quad = \left[\frac{1}{N} \cdot \sum_{i \in M \cup S} \frac{h_{\rm bi}}{n_b}\right] \cdot \frac{N - l}{N}. \end{aligned} \tag{7}$$

P_C is explained in Section III-B.

Case 4: (T_B, T_N)*:* This happens when a bit-flip occurs in the opcode field. If the branch-I is changed to a normal-I or store-I, the register field will be the destination of the operation. For example, an instruction "bne r1 r2 label1" is altered to "and r1 r2 r3"; thus r1 becomes the destination register for and operation. If this corrupted instruction is in the live range of r1, this instruction modifies the value in r1, and this error propagates and is detected when the CI is executed. Thus, a UB estimate is

$$\begin{aligned}\Pr\{(T_B, T_N)\} &\approx \Pr\{I_f \in B\} \cdot \Pr\{b_f \in W_{\rm op}\} \cdot \overline{q}_{\rm reg} \\ &= \frac{|B|}{N} \cdot \frac{W_{\rm op}}{n_b} \cdot \overline{q}_{\rm reg}. \end{aligned} \tag{8}$$

For example, if 90% of registers are always live during execution, the probability hat the register specified by the register field has a live value is 0.9, thus $\overline{q}_{\rm reg} = 0.1$. If the program is fully optimized, the compiler tries to maximize the register use, and $\overline{q}_{\rm reg}$ can be kept low.

Case 5: (T_B, T_S)*:* If the HamD between a branch-I and store-I is greater than 1, then this case cannot happen, assuming 1 single-bit error. On the other hand, if the HamD is 1, then a bit-flip in that bit position introduces this case. The changed instruction stores a value in the location specified by the offset field. If this stored value is used later, the error is detected, but if it is not used, it is undetected

$$\Pr\{(T_B, T_S)\} \approx \frac{1}{N} \cdot \left[\sum_{i \in B} \frac{h_{\mathrm{si}}}{n_b}\right] \cdot \overline{q}_{\mathrm{st}}. \tag{9}$$

Case 6: (T_B, T_B)*:* There are possibilities:

- bit-flip in offset, or
- bit-flip in register field.

The probability that a fault occurs in the offset field and is not detected is:

$$\Pr\{I_f \in B\} \cdot \Pr\{b_f \in W_{\mathrm{off}}\} \cdot P_C = \frac{|B|}{N} \cdot \frac{|W_{\mathrm{off}}|}{n_b} \cdot \frac{N-l}{N}. \tag{10}$$

If the register field is corrupted, this fault might, or might not, cause an rror. Calculate the UB of the probability of missing the error because exact prediction of the conditional jump is not feasible, as shown in examples 1 and 2.

- Example 1: $\mathrm{Branch\_greater\,r1\,r3\,label1}$.

  If the register field is changed and r1 is altered to r2, it is difficult to know that the control flow is corrupted or not unless the "values of the registers in run time" are known. For example, let r3 be loaded with zero. If r2 has positive values while r1 has negative values during execution time, the branch is taken, which is incorrect. Only a simulation can give the exact number, but simulation is very expensive.
- Example 2: $\mathrm{Branch\_not\_equal\,r1\,r2\,label1}$.

  If the register field is changed, and r1 is altered to r3, but the value in r3 is different from the value in r2, this fault does not change the control flow. If there is equal probability for the numbers in the registers, then the probability of changing the control flow in one execution of this branch is almost 0, because the probability that r2 and r3 have the same value is almost 0. However, only the run-time values of $\mathrm{r1}, \mathrm{r2}, \mathrm{r3}$ determine whether there is a control-flow error during execution of the whole program.

  Examples 1 and 2 show that it is very difficult to predict whether the fault affects the control-flow or not, without running the program. Therefore, assuming that the faults in the register field of the branch-I are not detected at all, the probability in (11) is a UB of missing this case

$$\Pr\{I_f \in B\} \cdot \Pr\{b_f \in W_{\mathrm{reg}}\} = \frac{|B|}{N} \cdot \frac{|W_{\mathrm{reg}}|}{n_b}. \tag{11}$$

  Adding (10) and (11), the probability that (T_B, T_B) happens but is not detected is:

$$\Pr\{(T_B, T_B)\} = \frac{|B|}{N} \cdot \frac{|W_{\mathrm{off}}|}{n_b} \cdot \frac{N-l}{N} + \frac{|B|}{N} \cdot \frac{|W_{\mathrm{reg}}|}{n_b}. \tag{12}$$

All Cases: Summing (6)–(9) and (12), the UB of probability that a fault occurs in the code segment and is not detected is

$$\begin{aligned} P_u &\approx \Pr\{(T_S, T_S)\} + \Pr\{(T_S, T_N)\} + \Pr\{(T_N, T_B)\} \\ &\quad + \Pr\{(T_S, T_B)\} + \Pr\{(T_B, T_N)\} + \Pr\{(T_B, T_S)\} \\ &\quad + \Pr\{(T_B, T_B)\} \\ &= \frac{|S|}{N} \cdot \left[\left(1 - \frac{|W_{\mathrm{op}}|}{n_b}\right) + k \cdot \frac{|W_{\mathrm{op}}|}{n_b}\right] \cdot \overline{q}_{\mathrm{st}} \\ &\quad + \left[\frac{1}{N} \cdot \sum_{i \in M \cup S} \frac{h_{\mathrm{bi}}}{n_b}\right] \cdot \frac{N-l}{N} + \frac{|B|}{N} \cdot \frac{|W_{\mathrm{op}}|}{n_b} \cdot \overline{q}_{\mathrm{reg}} \\ &= \left[\frac{1}{N} \cdot \sum_{i \in B} \frac{h_{\mathrm{si}}}{n_b}\right] \cdot \overline{q_{\mathrm{st}}} + \frac{|B|}{N} \cdot \frac{W_{\mathrm{off}}}{n_b} \cdot \frac{N-l}{N} + \frac{|B|}{N} \cdot \frac{W_{\mathrm{reg}}}{n_b}. \end{aligned} \tag{13}$$

ACKNOWLEDGMENT

The authors thank Dr. N. Saxena for his valuable suggestions and S. Mitra for his helpful comments.

REFERENCES

[1] P. P. Chang *et al.*, "IMPACT: An architectural framework for multiple-instruction issue processors," in *Proc. 18th Int. Symp. Computer Architecture (ISCA)*, vol. 19, 1991, pp. 266–275.

[2] M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[3] D. E. Culler and J. P. Singh, *Parallel Computer Architecture*: Morgan Kaufmann, 1999.

[4] J. Cusick, "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors," *IEEE Trans. Nuclear Science*, vol. NS-32, pp. 4206–4211, Dec. 1985.

[5] A. L. Hopkins, Jr. *et al.*, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," *Proc. IEEE*, vol. 66, pp. 1221–1239, Oct. 1978.

[6] L. Chen and A. Avizienis, "N-Version programming: A fault-tolerance approach to reliability of software operation," in *IEEE 8th Int. Symp. Fault Tolerant Computing (FTCS-8)*, 1978, pp. 3–9.

[7] P. K. Lala *et al.*, "On self-checking software design," in *IEEE Proc. SOUTHEASTCON*, 1991, pp. 331–335.

[8] A. Ersoz, D. M. Andrews, and E. J. McCluskey, "The Watchdog Task: Concurrent Error Detection Using Assertions," Center for Reliable Computing, Stanford Univ., CA, USA, CRC-TR-85-8, 1985.

[9] D. J. Lu, "Watchdog processors and VLSI," in *Proc. National Electronics Conf.*, vol. 34, 1980, pp. 240–245.

[10] A. Mahmood and E. J. McCluskey, "Watchdog processor: Error coverage and overhead," in *Digest, The 15th Ann. Int. Symp. Fault-Tolerant Computing (FTCS-15)*, 1985, pp. 214–219.

[11] D. M. Blough and G. M. Masson, "Performance analysis of a generalized concurrent error detection procedure," *IEEE Trans. Computers*, vol. 39, no. 1, pp. 47–62, Jan. 1990.

[12] D. M. Blough, "Fault detection and diagnosis in multiprocessor systems," Ph.D. dissertation, Johns Hopkins University, 1988.

[13] D. J. Lu, "Watchdog processor and structural integrity checking," *IEEE Trans. Computers*, vol. C-31, pp. 681–685, July 1982.

[14] M. Namjoo, "Techniques for concurrent testing of VLSI processor operation," in *Digest of Papers, IEEE Test Conf.*, Nov. 1982, pp. 461–468.

[15] J. P. Shen and M. A. Schuette, "On-line self-monitoring using signatured instruction streams," in *Proc. Int. Test Conf.*, Oct. 1983, pp. 275–282.

[16] J. B. Eifert and J. P. Shen, "Processor monitoring using asynchronous signatured instruction streams," in *Digest of Papers, 14th Ann. Int. Conf. Fault-Tolerant Computing*, June 1984, pp. 394–399.

[17] K. Wilken and J. P. Shen, "Concurrent error detection using signature monitoring and encryption: Low-cost concurrent-detection of processor control errors," in *Dependable Computing for Critical Applications*, A. Avizienis and J. C. Laprie, Eds: Springer-Verlag, 1989, vol. 4, pp. 365–384.

[18] K. Wilken and J. P. Shen, "Continuous signature monitoring: Low-cost concurrent-detection of processor control errors," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 629–641, June 1990.
[19] N. R. Saxena and E. J. McCluskey, "Control-flow checking using Watchdog assists and extended-precision checksums," *IEEE Trans. Computers*, vol. 39, pp. 554–559, Apr. 1990.
[20] H. Madeira and J. G. Silvia, "On-line signature learning and checking," in *Dependable Computing for Critical Applications*, J. F. and R. D. Schlichting, Ed: Springer-Verlag, 1992, vol. 2, pp. 395–420.
[21] N. Oh, P. Shirvani, and E. J. McCluskey, "Control-Flow Checking by Software Signatures," Center for Reliable Computing, Stanford Univ., CA, CRC-TR-00-4 (CSL TR num 00-800), May 2000.
[22] D. Reynolds and G. Metze, "Fault detection capabilities of alternating logic," *IEEE Trans. Computers*, vol. C-27, pp. 1093–1098, Dec. 1978.
[23] J. J. Shedletsky *et al.*, "Error correction by alternate-data retry," *IEEE Trans. Computers*, vol. C-27, pp. 106–112, Feb. 1978.
[24] K. Takeda *et al.*, "Logic design of fault-tolerant arithmetic units based on the data-complementation strategy," in *10th Int. Symp. Fault-Tolerant Computing*, 1980, pp. 348–350.
[25] J. H. Patel *et al.*, "Concurrent error detection in ALU's by REcomputing with shifted operands," *IEEE Trans. Computers*, vol. C-31, pp. 589–595, July 1982.
[26] Y.-M. Hsu *et al.*, "Time redundancy for error detecting neural networks," in *Proc. IEEE Int. Conf. Wafer-Scale Integration*, 1995, pp. 111–121.
[27] D. Andrews, "Using executable assertions for testing and fault tolerance," in *9th Int. Symp. Fault-Tolerant Computing*, 1979.
[28] G. A. Kanawati, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Evaluation of integrated system-level checks for on-line error detection," in *Proc. IEEE Int. Computer Performance and Dependability Symp.*, 1996, pp. 292–301.
[29] C. Rabejac, J.-P. Blanquart, and J.-P. Queille, "Executable assertions and timed traces for on-line software error detection," in *Proc. 26th Int. Symp. Fault-Tolerant Computing*, 1996, pp. 138–147.
[30] G. Miremadi, J. Karlsson, J. U. Gunneflo, and J. Torin, "Two software techniques for on-line error detection," in *Digest of Papers, 22nd Ann. Int. Symp. Fault-Tolerant Computing*, 1992, pp. 328–335.
[31] G. Miremadi, J. T J. Ohlsson, M. Rimen, and J. Karlsson, "Use of time, location and instruction signatures for control flow checking," in *Proc. DCCA-5 Int. Conf., 1985*.
[32] J. C. Fabre, Y. Deswarte, J-C. Laprie, and D. Powell, "Saturation: Reduced idleness for improved fault-tolerance," in *IEEE 18th Int. Symp. Fault-Tolerant Computing (FTCS-18)*, 1988, pp. 200–205.
[33] G. Sohi *et al.*, "A study of time-redundant fault tolerance techniques for high-performance pipelined computers," in *IEEE 19th Int. Symp. Fault-Tolerant Computing (FTCS-19)*, 1989, pp. 436–443.
[34] D. M. Blough and A. Nicolau, "Fault tolerance in super-scalar and VLIW processors," in *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1992, pp. 193–200.
[35] M. A. Schuette and J. P. Shen, "Exploiting instruction-level parallelism for integrated control-flow monitoring," *IEEE Trans. Computers*, vol. 43, pp. 129–140, 1994.
[36] J. G. Holm and P. Banerjee, "Low cost concurrent error detection in a VLIW architecture using replicated instructions," in *Proc. Int. Conf. Parallel Processing (ICCP)*, 1992, pp. 192–195.
[37] D. A. Patterson and J. L. Hennesy, *Computer Architecture: A Quantitative Approach*: Morgan Kaufmann, 1996.
[38] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*: H. Freeman, 1979.
[39] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[40] R. Stallman, *Using and Porting GNU CC*: Free Software Foundation, 1998.
[41] J. Heinrich, *MIPS R4000 Microprocessor User's Manual*: MIPS Technologies Inc., 1994.
[42] M. Rebaudengo and M. Reorda, "Soft-error detection through software fault-tolerance techniques," in *Int. Symp. Defect and Fault Tolerance in VLSI Syst.*, 1999, pp. 210–218.

Nahmsuk Oh received the B.E. degree (1996) in electronics engineering from Yonsei University, Korea; the M.S.E.E. degree (1998) from Stanford University, the Ph.D. degree in electrical engineering and the Ph.D. minor in Computer Science (2001) from Stanford University.

He is currently with Synopsys Inc., Mountain View, CA. His research interests include fault tolerant computing, computer architecture, logic synthesis, and VLSI design/test.

Dr. Oh is a member of the IEEE.

Philip P. Shirvani received the B.S. degree (1991) in electrical engineering from Sharif University of Technology, Tehran, Iran, and the M.S. (1996) and Ph.D. degrees, both in electrical engineering, from Stanford University.

He was Assistant Director for the Stanford ARGOS project at the Center for Reliable Computing, Stanford University. His research interests include computer architecture, fault-tolerant computing, and VLSI design and test. He is currently with Atheros, Inc., Sunnyvale, CA.

Dr. Shirvani is a member of the IEEE.

Edward J. McCluskey received the A.B. degree (*summa cum laude,* 1953) in mathematics and physics from Bowdoin College, and the B.S. (1953), M.S. (1953), and Sc.D. (1956) degrees in electrical engineering from MIT. The degree of Doctor Honoris Causa was awarded in 1994 by the Institut National Polytechnique de Grenoble.

He worked on electronic switching systems at the Bell Telephone Laboratories from 1955 to 1959. Then he moved to Princeton University, where he was Professor of Electrical Engineering and Director of the University Computer Center. In 1966, he joined Stanford University, where he is Professor of Electrical Engineering and Computer Science, as well as Director of the Center for Reliable Computing. He founded the Stanford Digital Systems Laboratory (now the Computer Systems Laboratory) in 1969 and the Stanford Computer Engineering Program (now the Computer Science M.S. degree program) in 1970. The Stanford Computer Forum (an Industrial Affiliates Program) was started by Dr. McCluskey and two colleagues in 1970 and he was its Director until 1978. He developed the first algorithm for designing combinational circuits—the Quine–McCluskey logic minimization procedure as a doctoral student at MIT. At Bell Labs and Princeton, he developed the modern theory of transients (hazards) in logic networks and formulated the concept of operating modes of sequential circuits. His Stanford research focuses on logic testing, synthesis, design for testability, and fault-tolerant computing. Prof. McCluskey and his students at CRC worked out many key ideas for fault-equivalence, probabilistic modeling of logic networks, pseudo-exhaustive testing, and watchdog processors. He collaborated with Signetics researchers in developing one of the first practical multivalued logic implementations and then worked out a design technique for such circuitry. He has published several books and book chapters, as well as hundreds of papers. His most recent book is *Logic Design Principles With Emphasis on Testable Semicustom Circuits,* (Englewood Cliffs, NJ: Prentice-Hall, 1986).

Dr. McCluskey served as the first President of the IEEE Computer Society. His most recent honors include election to the National Academy of Engineering, 1998, the 1996 IEEE Emanuel R. Piore Award, IEEE Computer Society Golden Core Member. In 1984, he received the IEEE Centennial Medal and the IEEE Computer Society Technical Achievement Award in Testing. In 1990, he received the EURO ASIC 90 Prize for Fundamental Outstanding Contribution to Logic Synthesis. The IEEE Computer Society honored him with the 1991 Taylor L. Booth Education Award. He is a Fellow of the IEEE, AAAS, and ACM.