Error Detection by Duplicated Instructions in Super-scalar Processors

Nahmsuk Oh, Philip P. Shirvani and Edward J. McCluskey

CSL-TR 00-5 **April, 2000**	**Center for Reliable Computing** Gates Building 2A, Room 236 Computer Systems Laboratory Dept. of Electrical Engineering and Computer Science Stanford University Stanford, California 94305-9020

ABSTRACT

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

In EDDI, we duplicate instructions during compilation and use different registers and variables for the new instructions. Especially for the fault in the code segment of memory, we have derived formulas to estimate the error detection coverage of EDDI using probabilistic methods. These formulas use statistics of the program, which are collected during compilation. We applied our technique to eight benchmark programs and estimated the error detection coverage. Then, we verified the estimates by simulations, 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, we show that 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 system but they need dependability in the computer system. The *Control Flow Checking by Software Signatures* (CFCSS) technique can be used with EDDI to increase the fault coverage.

In order to reduce the performance overhead, our technique schedules the instructions that are added for detecting errors such that Instruction-Level Parallelism (ILP) is maximized. We have showed that the execution time overhead in a 4-way super-scalar processor is less than the execution time overhead in the processors that can issue 2 instructions in one cycle.

FUNDING: 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, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047

Imprimatur: Nirmal Saxena and Subhashish Mitra

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

Permanent faults or transient errors in a computer system may 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 may not be detected and may 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 utilize 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 utilization of resources and consequently, some functional units are idle during execution. Chang and Johnson [1][2] have shown 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 data presented in [2], even in an ideal machine with infinite machine resources, perfect branch prediction, and ideal register renaming, there is no significant improvement in speedup beyond four instruction issues per cycle as shown in Fig.1.1. Therefore, if we cannot simultaneously execute more than four instructions in one clock cycle but have enough resources, we can use most of the idle resources for error detection.

The focus of this paper is on concurrent error detection by exploiting ILP of super-scalar architectures. Duplicated instructions in EDDI have no effect on the results of the program but detect errors in the system during run time. The instructions for error detection are scheduled to utilize idle resources in super-scalar architecture to reduce performance overhead.

The single event upset (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 1 to 0. It may modify an instruction of the program or 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, one bit of the data bus gets complemented and modifies 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.

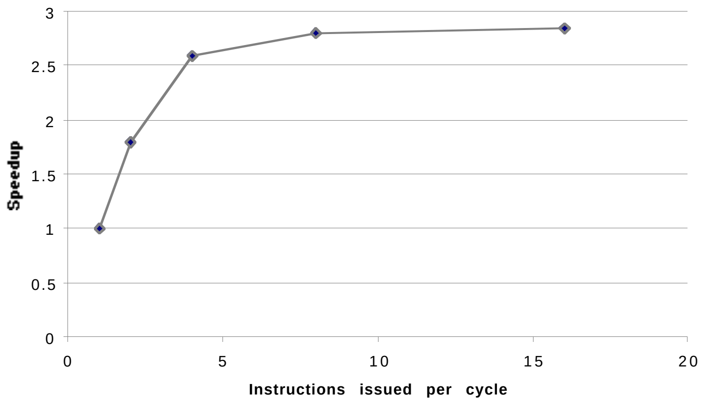


Fig. 1.1 Estimated speedup under ideal super-scalar execution. The figure shows the distribution of available instruction-level parallelism and speedup under ideal super-scalar execution. The graph is cited from [3] and shows the average of speedups reported in Johnson [2] for several benchmarks.

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 may deviate from its correct instruction sequence, for example, due to a fault in a branch instruction, resulting a control flow error. These are errors that can be detected by EDDI.

After addressing related works in Sec. 2, we present the algorithm in Sec. 3 and Sec. 4. We show the detailed analysis of the algorithm in Appendix (Sec. 10). With MIPS architecture, we estimate the fault coverage during compile time in Sec. 5 and verify the results by simulation in Sec 6. In Sec. 7, we compare our technique with different duplication methods, and finally, we conclude the paper in Sec. 8.

2. Related Work

A traditional concurrent error detection techniques is to use massive redundancy. N modular hardware redundancy [5] and N version programming [6] are examples of massive redundancy, but these techniques incur (N-1) hundred percent 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 employing a watchdog processor [9] [10] have been proposed.

While Blough [11][12] presented and analyzed a general procedure for error detection in complex system, called *data block capture and analysis monitoring process*, a number of techniques called signature monitoring techniques have been developed, which focus on checking control flow errors in the program. *Signature monitoring* is an analysis method 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 checksum method [19], and 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, Control Flow Checking by Software Signatures (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.

In an effort to decrease hardware cost, time redundancy has received attention. Time redundancy methods include alternating logic [22], alternate-data retry [23], data complementation [24], REcomputing with Shifted Operands [25] and time redundancy in neural networks [26]. Time redundancy reduces the amount of extra hardware at the expense of additional time. If the system is able to complete its computations before its specified time limit, the extra time can be used for error detection. 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 errors, 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 [27] [8] [28] [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 execution time overhead and induce performance loss.

Researchers have made attempts to exploit the unused resources of systems for concurrent error checking. In a multi-tasking and multi-computer environment, idle computers are used to execute replicated tasks for error checking [32]. Application of the RESO [25] technique to the Cray-1 has been studied using unutilized resources employing machine parallelism [33]. Utilizing the spare capacity in super-scalar and *Very Long Instruction Word* (VLIW) processors to tolerate functional unit failures has been proposed in [34]. These two techniques utilize 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 the incorporation of 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 only detects control flow errors.

The idea of duplicate instructions in a VLIW processor is previously investigated by Holm and Banerjee [36]. They assume fault-free memory operations and control operations, and duplicate only the ALU instructions to detect errors in data paths. For the duplicated ALU instructions, they use 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 is corrupted, the error cannot be detected. In EDDI, we do not assume any fault-free operations. We duplicate the variables and data structures, and use different registers for the duplicated instructions; therefore, EDDI can also detect the memory operation errors. Moreover, we interleave 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 other previous techniques, our technique is a pure software method; thus, it is not necessary to modify the original processor architecture. The targets of our technique are inter-block and intra-block control flow errors, data or code change in memory, and transient errors in functional units. Most instructions for error detection are scheduled to use idle resources exploiting super-scalar architecture. They have no dependency on original instructions and it means more ILP is added to the original program to maximize resource utilization. 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.

3. Error Detection by Duplicated Instructions

3.1 Preliminaries

Duplicated Instructions in EDDI have no effect on the result of the program but detect errors in the system during run-time. The purpose of EDDI is to check any deviation from the expected behavior of the computer system and 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 *master instruction* is the original instruction in the source code and a *shadow instruction* is the duplicated instruction added to the source code. General purpose registers and memory are partitioned into two groups for master and shadow instructions. The registers and memory for master instructions should always have the same value as the registers and memory for shadow instructions; thus, if there has been a mismatch between a pair of registers for master and shadow instruction, an error can be detected by comparing these two register values. A *comparison instruction* compares the values of the two registers and invokes an error handler if they do not match.

A simple example of source code is shown below:

```
ADD  R3, R1, R2                     ; R3 <- R1 + R2.
```

This is an addition instruction; the first operand is the target register, and the second and the third operands are the source registers, thus register R3 should take the sum of the values in registers R1 and R2. The corresponding shadow and comparison instructions are:

```
ADD   R3,  R1,  R2                        ;master instruction
ADD   R23, R21, R22                       ;shadow instruction
BNE   R3,  R23,  gotoError                ;comparison instruction
```

Suppose registers R1, R2 and R3 are the master registers and R21, R22 and R23 are the shadow registers that contain the same value as R1, R2 and 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 instruction `BNE  R3  R23  gotoError` (branch to gotoError if R3 and R23 are different) performs a comparison and transfers control to an error handler if an error has occurred.

We have to determine where we insert this comparison instruction. We propose that comparison should be performed right before storing register values in memory or deciding the direction of a branch or jump instruction. Registers are used to hold temporary values for computation and the results of computation are stored in memory to free up the registers for later usage. We only compare final calculation results that will be stored in memory for later use. We do not need to compare intermediate computation results that propagate and corrupt final results. If an error changes an intermediate result, the effect of the change will probably propagate down to the final computation and it will also corrupt the final result (if not, i.e., the error is masked out in intermediate computation, and we can take the final result as a correct answer.) Thus, we will have different final computation results in master and shadow instructions and the error can be detected by comparison of the master and shadow registers.

On the other hand, registers often hold values to resolve branching directions. A typical example is shown in Fig. 3.1.

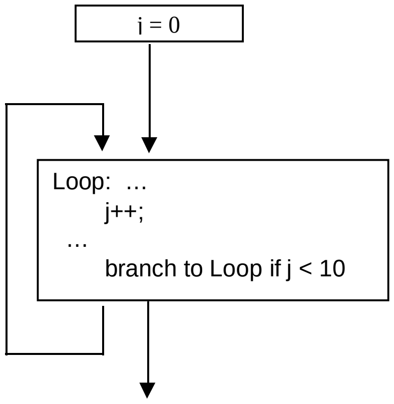


Fig. 3.1 Branching at the end of the loop.

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 variable *j* and is corrupted while executing instructions, it will produce an unexpected result. Thus, in order 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. A *storeless basic block* is a sequence of instructions in which there is no store instruction except for the last instruction and the last instruction may be a store instruction or a branch instruction. An example is shown in Fig. 3.2. A storeless basic block is always inside a basic block. Within a storeless basic block, shadow instructions are scheduled to maximize resource utilization by attempting to use idle resources, which are not used by master instructions. If the last instruction of a storeless basic block is a store instruction, a comparison instruction is placed before the store to compare the master and shadow register values that will be stored in memory. Scheduling of shadow instructions will be discussed in detail later in Sec. 4.

```
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. 3.2 Storeless basic block example.

3.2 Algorithm for EDDI

Let I_i be the *i*th instruction of the program and $i = 1, 2, \ldots N$, where N is the number of instructions in the program. An instruction I_j is *dependent* on an instruction I_i, if I_j uses the result of I_i, thus I_j will be 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). A *dependency graph* G_D is an ordered tuple (V, E) consisting of a nonempty set $V = \{I_j\ ;\ j = 1, 2, \ldots, N\}$ and a set $E = \{(I_i, I_j)\ ;\ i, j = 1, 2, \ldots, N\}$. $G_{Dk}(V_k, E_k)$ is a subgraph of G_D denoting dependencies within the *k*th storeless basic block, consisting of $V_k = \{I_{kj}\ ;\ j = 1, 2, \ldots, n_k\}$ and $E_k = \{(I_{ki}, I_{kj})\ ;\ i, j = 1, 2, \ldots, n_k\}$ where n_k is the number of instructions in the *k*th storeless basic block. The G_{Dk}'s form a partition on G_D.

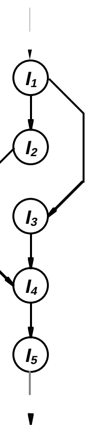
The algorithm adding instructions for error detection is:

```
1  Build a dependency graph $G_D$ of the program
2  For each $G_{Dk}$ {
3          Build a graph $H_{Dk}(V_k', E_k')$ where
4                  $V_k' = \{I_{kj}'$ ; a shadow instruction of $I_{kj} \in V_k\}$
5                  $E_k' = \{(I_{ki}', I_{kj}')\ ;\ (I_{ki}, I_{kj}) \in E_k\}$
6          If $I_{kn}$ is a store or a conditional branch instruction
7                  Create a comparison instruction $I_{kc}$
8                  Create an ordered edge $(I_{kn}, I_{kc})$ and $(I_{kn}', I_{kc})$ to connect $H_{Dk}$ to $G_D$.
9          Schedule $I_{ki}, I_{ki}', i = 1,2,\ldots,n_k$ and $I_{kc}$ according to the algorithm described in Sec.
4
10 }
```

The following example should serve to clarify the algorithm.

```
          ...
I1: ADD      R1, R2, R3
I2: SUB      R4, R1, R2
I3: AND      R5, R1, R2
I4: MUL      R6, R4, R5
I5: ST       R6
          ...
```

(a)



(b)

Fig. 3.3 (a) Instructions in one particular storeless basic block
(b) Dependency graph of this storeless basic block

Fig. 3.3 (a) shows a sequence of instructions in one particular storeless basic block of the program and (b) is G_{Dk}, a part of the dependency graph G_D, representing dependencies among instructions within this storeless basic block. For example, two edges, (I_2, I_4) and (I_3, I_4) imply that I_4 needs the results of I_2 and I_3. Fig. 3.4 illustrates how to construct a subgraph H_{Dk} within a storeless basic block and attach it to G_D.

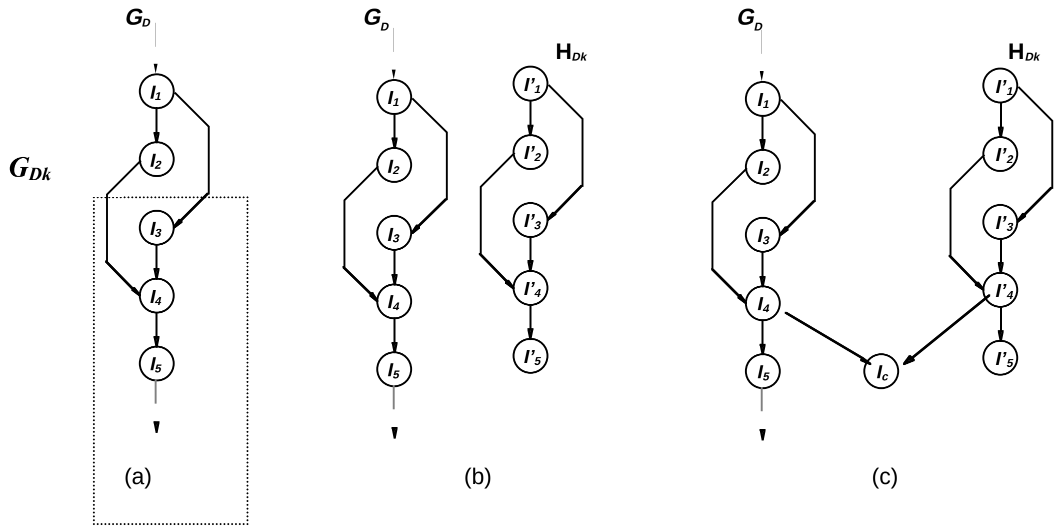


Fig. 3.4 (a) Dependency graph (b) Constructing H_{Dk} (c) Adding H_{Dk} to G_D.

The portion of G_D for the storeless basic block in Fig. 3.4 is shown in (a). Lines 3, 4 and 5 in the algorithm build a graph H_{Dk} in (b) and the vertices in H_{Dk} are shadow instructions (I_1', I_2', I_3', I_4',

I_5') corresponding to master instructions (I_1, I_2, I_3, I_4, I_5) in G_D. The dependencies in H_{Dk} are inherited from the dependencies in G_{Dk}, thus the edges in H_{Dk} show the same dependencies among shadow instructions as those of master instructions. After constructing H_{Dk}, a comparison instruction denoted as I_c is added to G_{Dk} because I_5 and I_5' are store instructions 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_{Dk} to G_D. Now, G_D represents new relation between master, shadow and comparison instructions. These instructions are scheduled in line 9. The scheduled instructions are shown in the following:

```
I1:   ADD   R1, R2, R3
I2:   SUB   R4, R1, R2
I1':  ADD   R21, R22, R23
I3:   AND   R5, R1, R2
I3':  AND   R25, R21, R22
I4:   MUL   R6, R4, R5
I2':  SUB   R24, R21, R22
I4':  MUL   R26, R24, R25
Ic:   BNE   R6, R26, go_to_error_handler
I5:   ST    R6
I5':  ST    R26
```

Scheduling algorithms will be discussed in detail later in Sec. 4.

The computational complexity of the algorithm depends on which scheduling algorithm is chosen in line 9. The complexity of building the dependency graph in line 1 is $O(n^2)$ [37 pp.321] and building H_{Dk} (line 3 – 8) is $O(|V| + |E|) = O(n)$; thus, if we choose a heuristic scheduling algorithm that has the complexity of $O(n)$ (such as the list scheduling algorithm in Sec. 4.2), the complexity of the entire algorithm is $O(n^2)$. However, if we want the exact optimal solution and choose the integer linear programming for scheduling instructions, the complexity of the entire algorithm is dominated by the complexity of scheduling algorithm and turns to be exponential.

The correctness of the algorithm and fault coverage will be discussed in detail in Sec. 10. Basically, an error caused by the faulty instruction will propagate through the dependency graph to the comparison instruction, which will detect the mismatch between the master and shadow pair.

Control flow errors will be detected if the control flow is transferred to the place from which there exists a path in G_D to a comparison instruction and the number of master instruction executed is not equal to the number of shadow instructions executed on this path.

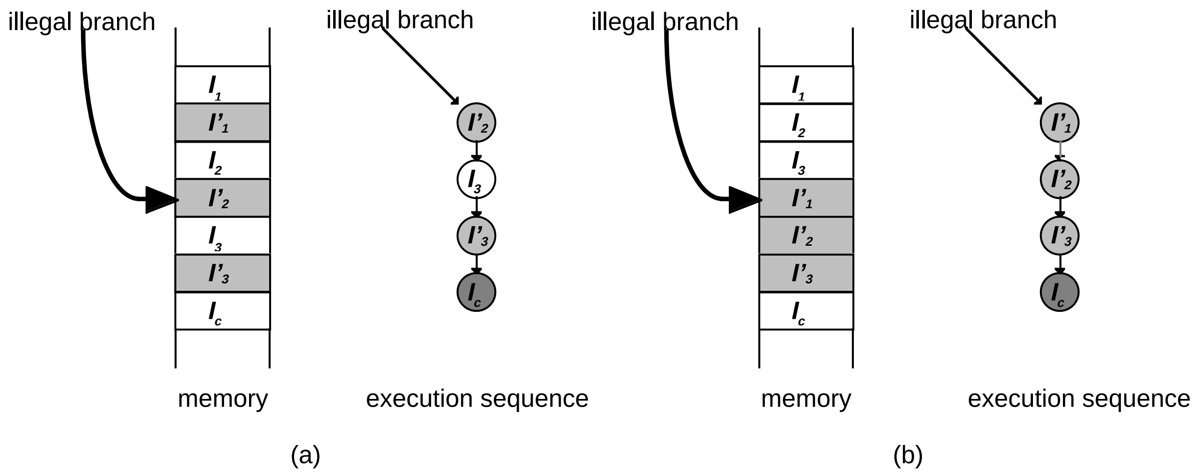


Fig. 3.5 (a) shadow instructions interleaved with master instructions in a storeless basic block

(b) shadow instructions grouped together and placed after master instructions

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

For example, suppose the shadow instructions are scheduled to interleave with master instructions in a storeless basic block as shown in Fig. 3.5 (a). An illegal branch transfers the control flow to the instruction I_2', i.e., I_2' is the first instruction after the illegal branch. When the control reaches I_c, 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 will be different. However, if an illegal branch jumps to one of I_1, I_2, and I_3, it is not detected since the number of executed master and shadow instructions 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 (b) since in the execution sequence, the number of master and shadow instructions are always different. Only an illegal branch to I_1 is not detected.

Let us define l as the number of instructions in the program for which an illegal branch to these instructions will be detected. Thus l is 3 in (a) and l is 5 in (b). For simplicity, suppose that $N = 7$ in the above example and an illegal branch jumps 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 (b) and is much lower than the one in (a), which is 4/7. The detailed analysis is presented in Appendix (Sec. 10).

4. Scheduling Instructions Exploiting Instruction-Level Parallelism

The shadow instructions should be scheduled with master instructions to maximize error detection coverage and to minimize time overhead using idle resources. Resource-constrained scheduling is a well known intractable problem. We will first formalize the scheduling of master and shadow instructions and consider an exact solution method (linear integer programming), then describe heuristic algorithms for static as well as dynamic scheduling.

4.1 Integer Linear Programming

G_{Dk} (V_k, E_k) is a subgraph of G_D denoting dependencies within the kth storeless basic block, where $V_k = \{ I_{kj} ; j = 1, 2, \ldots, n_k \}$ is a set of n_k instructions to be scheduled in kth storeless basic block and the edge set $E_k = \{ (I_{ki} , I_{kj}) ; i, j = 1, 2, \ldots, n_k \}$ represents dependencies. Let $D_k = \{ d_j ; j = 1, 2, \ldots, n_k \}$ be the set of instruction execution delays and $T_k = \{ t_j ; j = 1, 2, \ldots, n_k \}$ denote the *start time* for the instructions, i.e., the cycles in which the instructions start. There are n_{res} types of resources and we denote the mapping of the instructions to the unique resource types that they use by function $F_k : V_k \rightarrow \{1, 2, \ldots, n_{res}\}$. A resource constrained scheduling problem is one where the number of resources of any given type m is bounded by a set of integers $\{ a_m ; m = 1, 2, \ldots, n_{res} \}$.

A formal model of the scheduling problem under resource constraints can be achieved by using binary variables with two indices: $X_k = \{ x_{jl} ; j = 1, 2, \ldots, n_k ; l = 1, \ldots, \lambda \}$ where λ represents an upper bound on the latency of kth storeless basic block. A binary variable x_{jl} is 1 only when the instruction I_{kj} starts in step l of the schedule. Then, the scheduling can be formulated as a set of equations (from (1) to (5)) that have to be satisfied. The first four equations are the resource-constrained minimum latency scheduling equations and described in detail in [38]. Briefly, equation (1) tells us that the start time of each instruction is unique and equation (2) shows us that the dependency relations in G_{Dk} must be satisfied. The resource constraint must be satisfied at every cycle. An instruction I_{kj} is executing at cycle l when $\sum_{m=l-d_j+1} x_{jm} = 1$. The number of all instructions at cycle l of type k must be less than or equal to a_k. This is shown in

(3). We add one more condition in (5) and restrict the number of master instructions to being always greater than the number of shadow instructions until step λ-1 in the *k*th storeless basic block. The number of master and shadow instructions should be equal at step λ when scheduling is completed.

Objective function: minimize $\sum_j t_j$ such that

Constraints:

(1) $$\sum_l x_{jl} = 1, j = 1, 2, \ldots, n$$

(2) $$\sum_l l \cdot x_{il} - \sum_l l \cdot x_{jl} - d_j \geq 0, i, j = 1, 2, \ldots, n_k : (I_{ki}, I_{kj}) \in E_k$$

(3) $$\sum_{j:F(I_j)=m} \sum_{q=l-d_j+1}^{l} x_{jq} \leq a_{m,} m = 1, 2, \ldots, n_{res}, l = 1, \ldots, \lambda$$

(4) $$x_{jl} \in \{0,1\}, j = 1, 2, \ldots, n_k, l = 1, \ldots, \lambda$$

(5) $$\sum_{i:\text{master instruction}} \sum_{q=1}^{l} x_{iq} - \sum_{j:\text{shadow instruction}} \sum_{q=1}^{l} x_{jq} > 0, l = 1, \ldots, \lambda - 1$$

4.2 List Scheduling

The disadvantage of the integer linear programming formulation is the computational complexity of the problem: generally it is an NP-complete problem [38]. The number of variables, 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. We consider the *list scheduling* algorithm [39] to schedule master and shadow instructions.

Let m_i and s_i denote the number of master and shadow instructions up to the *i*th instruction within the *k*th storeless basic block.

```
1  ReadyList = instructions with 0 predecessors in $G_{Dk}$
2  Loop until ReadyList is empty {
3          Let $x$ be the instruction in ReadyList with highest priority
4          Schedule $x$ as early as possible satisfying the following constraints:
5                  dependencies in $G_{Dk}$
6                  resource constraints
7                  if $x$ is not the last instruction
8                       $m_i - s_i \neq 0$
9         Update predecessor count of $x$'s successor nodes
10        Update *ReadyList*
11        }
```

The priority list for the instruction *x* in line 3 can be the maximum execution delays from *x* to any instruction, resource requirements, the program source code order, or arrangement of master and shadow instructions, depending on heuristic urgency measure.

The heuristic nature of list scheduling may not find a exact optimum solution, but the computational complexity of the algorithm above is *O(n)*. In general, as shown in [39 pp. 211], solutions of list scheduling have been shown not to differ much in latency from the optimum ones, for those (small) examples whose optimum solution are known.

4.3 Interleaved Scheduling

A super-scalar processor with out-of-order execution capability is able to dynamically schedule the instructions by solving dependencies using hardware and simultaneously issue multiple instructions in one clock cycle. In order to reduce the cost of static scheduling discussed above, we can exploit the dynamic scheduling capability of the processor. The super-scalar processor has a buffer called *instruction window,* in which it decodes instructions and places, and at the same time, examines instructions in the window to find instructions that can be issued at 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 we put as many master and shadow instructions as possible in the instruction window, we have more chance to execute more instructions at the same time. Furthermore, instructions should be arranged such that control flow error detection coverage is maximized. The scheme we propose for achieving these two objectives together, is shown in Fig. 4.1.

If we interleave master instructions with shadow instructions, we can put the same number of master and shadow instructions in the instruction window. Since there is no data dependency between the master and shadow instructions, this scheme will provide maximum ILP in the instruction window; however, a simple interleaving scheme will miss half of the control flow errors because an illegal branch to master instructions is not detected. Instead, if we place two master instructions in the first two slots of the storeless basic block, interleave master instructions with shadow instructions in the middle, and place two shadow instructions in the last two slots, the condition $m_i - s_i \neq 0$ in line 8 can be satisfied until the last shadow instruction in the block. For the example shown in Fig. 4.1, $m_i - s_i > 0$ 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 since, after the illegal branch, it fails to execute one of the pair of master and shadow instruction on the execution path to a comparison instruction.

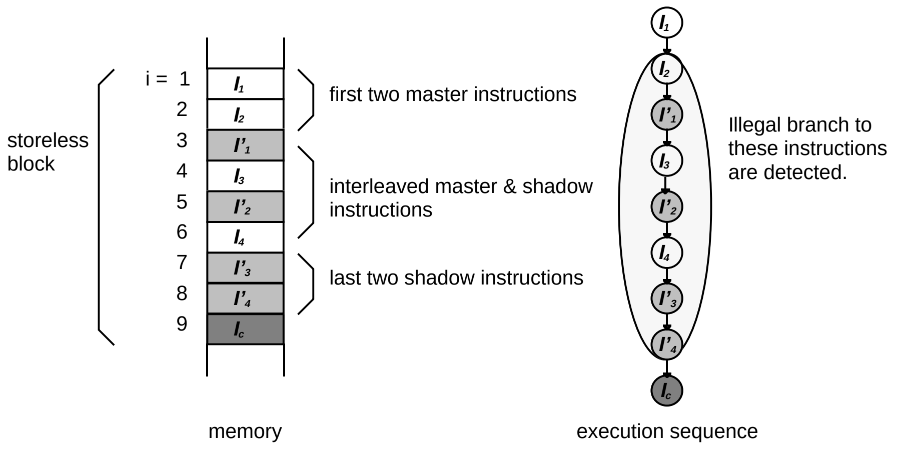

storeless block
i = 1
2
3
4
5
6
7
8
9
I1
I2
I'1
I3
I'2
I4
I'3
I'4
Ic
first two master instructions
interleaved master & shadow instructions
last two shadow instructions
Illegal branch to these instructions are detected.
memory
execution sequence

Fig. 4.1 Master and shadow instruction arrangement to detect a control flow error.

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 utilize idle resources is done by hardware while control flow error detection is achieved by arrangement of instructions.

5. Estimated Fault Coverage

Based on probability methods, we have derived formulas for estimating the error detection coverage of EDDI. These formulas and their derivation can be found in Appendix (Sec. 10). This technique can provide us the error detection coverage when EDDI is implemented in the program during compilation. This estimate is very useful when we need to predict the error detection coverage before we actually run the program, especially in the computers for space applications.

The estimated fault coverage can be obtained from equation (13) in Appendix. We took eight benchmark programs: FFT, matrix multiplication, Fibonacci number (numeric programs) Hanoi, Compress, Shuffle, Quick sort, and Insert sort (non-numeric programs). We applied the estimation method 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 master and shadow instructions. First, the compiler produced the assembly code of the program. Then, shadow instruction and comparison instructions are added in assembly code by our post-processor and the resultant assembly code with EDDI is compiled into object code by an assembler.

Based on the percentage of branch instructions and store instructions after EDDI is added to the programs, the probability of missing an error is calculated for the benchmark programs, and the results are shown in Table I.

The parameters described in Appendix are determined as follows. In the MIPS Instruction Set Architecture II [41], the immediate fields for the target address of branch and jump instruction take 16 bits and 26 bits of the 32 bit instruction, respectively. Based on these numbers, we assume $\frac{|W_{off}|}{n_b} = 0.50$ for branch instructions and $\frac{|W_{off}|}{n_b} = 0.81$ for jump instructions. We assume that $q_{st} = 0.95$, i.e., 95% of the stored data are used later in the program execution. This is a reasonable number considering most of the data in the above programs are repeatedly used during execution. We also assume that $q_{reg} = 0.9$, i.e., the register utilization is 90% since level 2 compiler optimization tries to maximize the register utilization. Table I shows the probabilities of each case discussed in the Appendix and finally reports the fault coverage of the technique. The Hamming distance between branch instructions and store instructions is greater than one; thus, $\Pr_u(T_S, T_B)$ and $\Pr_u(T_B, T_S)$ are zero and are omitted from the table.

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

Table I. Estimated fault coverage for benchmark programs.

	Branch instructions	Store instructions	$P_u(T_S,T_S)$	$P_u(T_N,T_B)$	$P_u(T_B,T_N)$	$P_u(T_B,T_B)$	Fault coverage
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%

6. Experimental Results

The same eight benchmark programs in Sec. 5 are chosen for the experiment to verify the estimated coverage calculated in the previous section. Our target machines are SGI Indigo with MIPS R4400 processor and SGI Octane that employs the 4-way super-scalar R10000 Mips processor. First, the source files are compiled and the target machine codes are generated without error detection instructions. A fault injector forced a bit flip in the code segment of the machine code. The location of the bit flip is determined by a random number generator. This machine code is executed and the result is shown in Table II.

The numbers in the second row 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. The third row means that erroneous result is repeatedly produced because the fault creates an infinite loop in the program. In the fourth row, the processor does not respond to the observer, so we have to manually stop the processor. The numbers in the fifth row show the number 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. Although a bit flip is inserted into the machine code, a correct result may be produced. The number of these cases is shown in the sixth row. This case can happen because a bit flip, occurred in unused bits of a `nop` instruction that fills the empty slot of the pipeline, has no effect on the output, or the non-used field of the instruction has a bit flip so that the effect of the bit flip is nullified. The last row denotes the total number of faults that produced incorrect outputs without being detected (the sum of the second, the third and the fourth rows). On average, in the programs without EDDI, 20% of the injected faults produced incorrect outputs and were not detected as shown in Table II.

Table II. Results of fault injection into original programs.

	FFT	hanoi	com-press	quick sort	fibon-acci	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%
Incorrect output undetected	46%	9%	24%	17%	18%	14%	27%	7%

Now, EDDI is included in the eight benchmark programs by the compiler postprocessor that we have developed. A bit flip is injected into the generated machine code, then the corrupted machine code is executed. The results are shown in Table III.

Table III. Results of fault injection into the programs with EDDI.

	FFT	hanoi	com-press	quick sort	fibon-acci	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%

The first and third columns in the graph in Fig. 6.1 illustrates the percentage of faults that produce undetected incorrect outputs in the original program and in the program with EDDI. The second column shows the estimated percentage of faults that are missed by EDDI. EDDI shows high error detection 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. 6.1. The error coverage calculated in Sec. 5 was around 98.2% in eight benchmark programs, and the error coverage in the experiment is about 98.5% in the benchmark programs.

Since 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 one. The execution time overhead is shown in Fig. 6.2. In addition, since we use general purpose registers as shadow registers, more register spilling occurs with EDDI; more spilling causes more performance overhead since it increases the number of memory operations. For example, matrix multiplication spills more registers than Fibonacci. Matrix multiplication needs as many registers as possible for numeric calculation, but Fibonacci needs just a few registers to keep the value and repeatedly compare the register values.

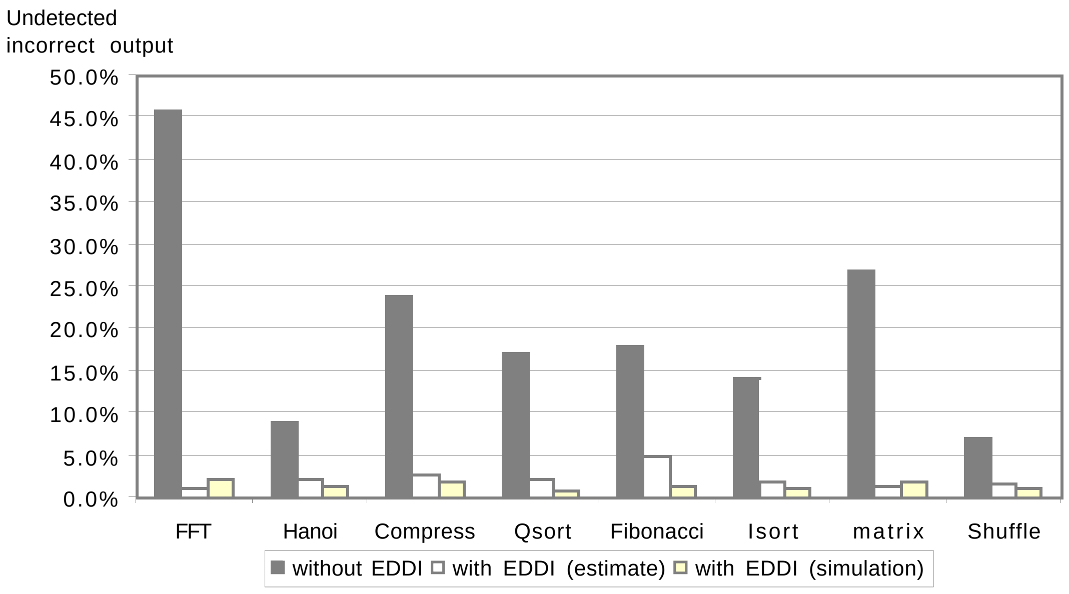


Fig. 6.1. Percentage of faults that produced incorrect outputs without being detected.

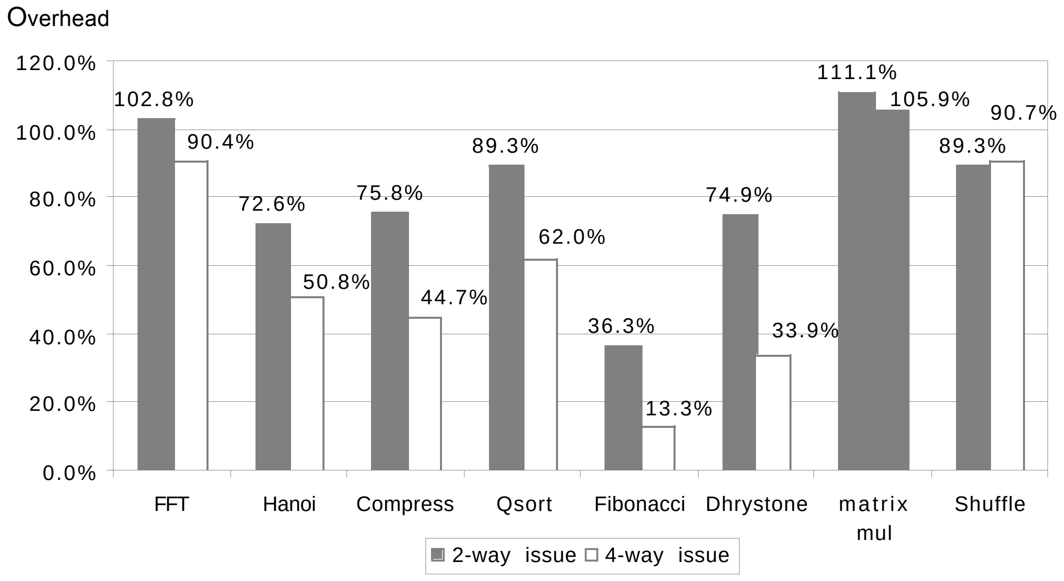


Fig. 6.2. Execution time overhead in R4400 (2-way issue) and R10000 (4-way issue)

Notice that for most of the programs, the overhead in execution time is less than 100%. Since we have duplicated instructions, the overhead might be close to or greater than 100%. There are two reasons for this low performance overhead in time. First, shadow instructions fill the empty slot of the pipeline; thus, we achieve higher utilization of the processor resources. Second, because a pair of master and shadow instruction are always independent of each other, shadow instructions add more parallelism to the program. A 4-way super-scalar processor can exploit this parallelism better than a 2-way super-scalar processor. This effect can be seen in Fig. 6.2 where the execution times for running the programs in two different processors are shown. We can observe that the execution time overhead in the R10000 is less than the one in the R4400: a 4-way issue machine has less time overhead than a 2-way issue machine. EDDI has less execution time overhead in processors that employ more aggressive super-scalar architectures.

7. Comparison of Different Source Level Duplication Methods

Our technique duplicates the instructions in an assembly source code to achieve error detection capability. The alternative way might be duplicating source code in the high level source language such as C or Pascal. All the variables, assignments, calculations as well as arguments are duplicated in order to compare the pair of variables or results of calculations. We consider two options for comparing the duplicated variables with the original variables in the source code: comparing the final outputs of the program and comparing the variables when they are updated. The former has the advantage of minimum performance overhead compared to the latter since, in the former, comparison instructions are placed only at the final output stage while, in the latter, comparison instructions are inserted whenever the variables are updated. However, comparing the values when they are updated has higher fault coverage than comparing only final output results. For example, if a branch instruction is corrupted and causes an infinite loop during the execution, the final output comparison cannot detect this error.

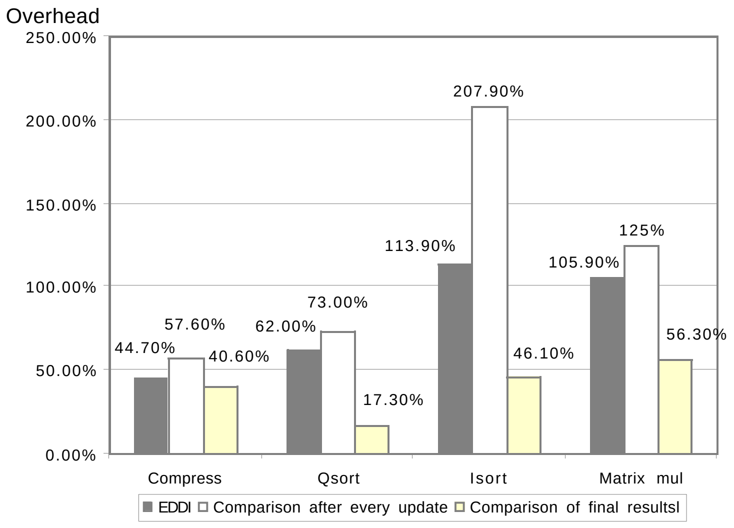


Fig. 7.1 Execution time overhead in different level of duplications

In this section, we compare three techniques: assembly source code duplication (EDDI), C source code duplication with comparison after updating variables, and C source code duplication with comparison of the final results. Fig. 7.1 shows the execution time overhead of the three techniques. The third technique, comparison at the final stage, shows the best performance because it has the fewest comparison instructions. However, the minimum number of comparison instructions has more chance to miss errors in the middle of execution.

Fig. 7.2 shows the fault coverage of the three techniques. The first technique, EDDI, shows better coverage than the others do. The second technique, the comparison after every update, has higher coverage than the third one, the comparison at the final stage, but shows lower coverage than the EDDI because the comparison in high level language sometimes generates more instructions and basic blocks when compiled. In contrast, in EDDI, only one branch instruction completes the comparison.

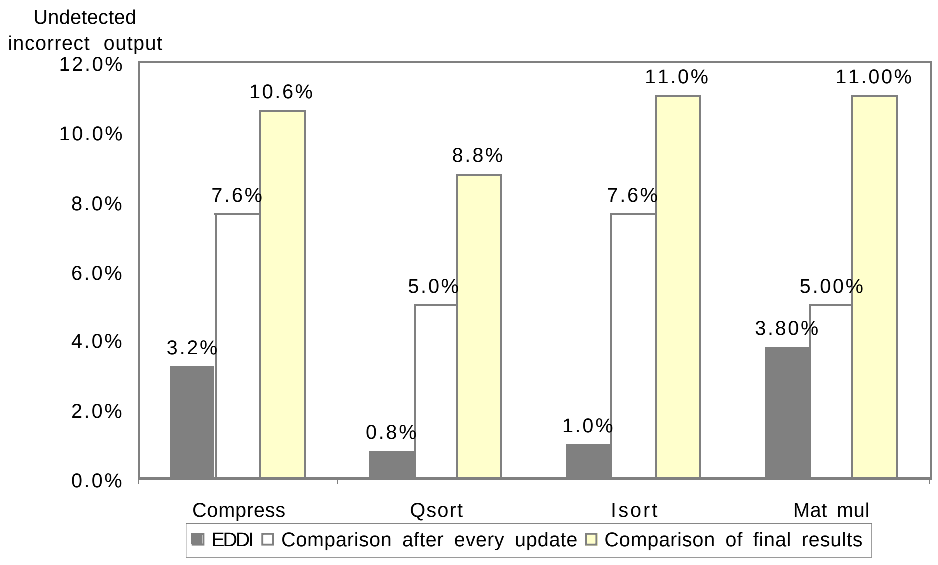


Fig. 7.2 Undetected incorrect outputs in different levels of duplication

Overall, EDDI shows the highest fault coverage with medium performance overhead. It is because EDDI handles duplication at the assembly level; therefore, it has finer grain error detection capability than other techniques at the C source level.

8. Discussion

The EDDI technique proposed in this paper is a pure software method that achieves high fault coverage in the computer system in which adding any extra hardware or modifying the existing hardware is not possible. Our simulation as well as our probabilistic estimate shows that our technique can achieve high error detection coverage.

We developed an algorithm for adding EDDI to a program and derived formulas that can be used to quickly estimate the fault coverage of EDDI while compiling the program. We verified the estimates by fault injection simulations. In the simulations, a fault injector forced a bit flip in the executable machine code. In eight benchmark programs without EDDI, on average, 20% of the injected faults produced incorrect results. However, in the program augmented by EDDI, only an average of 1.5% of injected faults produced undetected incorrect results. This result shows that EDDI can achieve high fault coverage only by pure software technique.

In super-scalar architecture, duplicated instructions can be scheduled to use idle resources of the processor in order to reduce performance overhead. Experimental result shows that approximately half of the benchmark programs have less than 50% execution time overhead when they run in a 4-way super-scalar processor. This execution time overhead might be high for the application programs that need high performance, but if high reliability is required without any help of special hardware, EDDI would be one of the best candidate techniques to adopt.

9. Acknowledgements

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, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047. The authors wish to thank Dr. Nirmal Saxena for his valuable suggestions and also Subhasish Mitra for his helpful comments.

10. Appendix

Error Detection Coverage Estimate

Any faults 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 fault coverage of our technique in the code segment of memory since the effect of faults depends on the run time behavior of the program. For instance, we 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 error detection coverage also depends on the input data sets since the dynamic values of the registers and memory are strongly affected by the input data. However, based on the probabilistic method, we can get an estimate of the fault coverage. In this section, we will describe how to estimate and predict the fault coverage 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 and perform an operation with an incorrect value. The goal is to detect these kinds of errors. Let us define *normal instruction* as an instruction that is neither a branch nor a store instruction. ISA stands for *Instruction Set Architecture* and *Hamming distance* of two numbers a and b is $|a \oplus b|$.

Denoting the total number of instructions in the program by N and the number of bits in one instruction by n_b, let us define nonempty sets and parameters to be used later:

T_S, T_B, T_N :	store, branch, and normal instruction type
τ :	a function that maps an instruction I to its instruction type
S :	$S = \{I_j : \tau(I_j) = T_S,\ j = 1,2,...,N\}$
B :	$B = \{I_j : \tau(I_j) = T_B,\ j = 1,2,...,N\}$
M :	$M = \{I_j : I_j \notin B \cup S, j = 1,2,...,N\}$
W_{op} :	the set of bits in the opcode field of an instruction
W_{reg} :	the set of bits in the register field of an instruction.
W_{off} :	the set of bits in the offset field of an instruction.
q_{st} :	$\dfrac{\left|\{I_j : I_j \in S,\ \text{data stored by } I_j \text{ are used later by subsequence instructions}\}\right|}{|S|}$
q_{reg}:	register utilization
h_{bi} :	number of branch instructions in the ISA that have Hamming distance 1 from instruction i
h_{si} :	number of store instructions in the ISA that have Hamming distance 1 from instruction i
(T_A, T_B):	a change of instruction from type T_A to type T_B
$\text{Pr}_u\{(T_A, T_B)\}$	the probability that a fault induces (T_A, T_B), but it is not detected by EDDI.

Instructions in a RISC architecture generally have two or three fields in them as shown in Fig. 10.1; opcode, register, and constant or offset field. If the opcode field is corrupted, 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), or (T_B, T_S). The latter is the case of (T_N, T_N), (T_B, T_B) or (T_S, T_S). For example, an `and` instruction changed to a store instruction 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, the original operation of the instruction is preserved but the source or destination of the operation is changed. This is the case of (T_N, T_N), (T_B, T_B) or (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.

opcode	register	constant or offset

Fig. 10.1. Fields in instructions of RISC architecture.

We will discuss and analyze all nine possible cases to estimate error coverage of our technique. The probability that an error occurs and escapes detection will be calculated for each case; then we can get the fault coverage by subtracting it from one. There might be some events that are not considered when calculating the probability but we can ignore them when the probabilities of them are very low.

Case 1: (T_N, T_N) (T_N, T_S)

This case will be detected since there always exists a path in G_D from the faulty instruction to a comparison instruction. There always exists a master and shadow instruction pair for normal instructions. If one of them 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 master and shadow pair also propagates along a path in G_D and these two paths will meet in the comparison instruction. The two different results are compared and the error of the faulty instruction is detected. If an error is masked during the execution the instructions along the path and does not corrupt the final result, the two compared results are the same and they are regarded as correct.

Case 2: (T_S, T_S) (T_S, T_N)

These cases will be detected if the data stored by either of the corrupted or non-corrupted store instruction in the master-shadow pair is used by a subsequent instruction. One of the master and shadow instruction pair stores a correct result in memory. If one of the master and shadow instructions 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 instruction. The error is detected as in Case 1.

In the case of (T_S, T_N), even if the incorrect stored data is not used later, this error will be detected if the normal instruction, which results from the error in the store instruction, modifies a live value in a master or shadow register. Otherwise, it will not be detected. If we ignore this special case and assume that the error is not detected only if the stored data is not used, the probability of these two cases not being detected will be (1). Let us denote the corrupted instruction by I_f and the corrupted bit in I_f by b_f, then:

$$\begin{aligned}
\Pr_u\{(T_S,T_S)\} &\approx \Pr\{I_f \in S\} \cdot \Pr\{b_f \notin W_{op}\} \cdot (1-q_{st}) \\
&= \frac{|S|}{N} \cdot (1-\frac{|W_{op}|}{n_b}) \cdot (1-q_{st}) \\
\Pr_u\{(T_S,T_N)\} &\approx \Pr\{I_f \in S\} \cdot k \cdot \Pr\{b_f \in W_{op}\} \cdot (1-q_{st}) \\
&= \frac{|S|}{N} \cdot k \cdot \frac{|W_{op}|}{n_b} \cdot (1-q_{st})
\end{aligned} \tag{6}$$

where (1- q_{st}) represents the ratio of stored data that are not used later in the program execution and k is the ratio of possible normal instructions, which can be created by a single bit flip in the opcode field, to total number of possible instructions in the ISA.

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

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

Assuming a single bit error, instructions with Hamming distance 1 from the branch instruction can be changed to a branch instruction, thus this case of error is restricted to the instructions that have Hamming distance 1 from the branch instruction. Let us call this event E_h. Normal and store instructions always exist as a master and shadow pair. If one of the pair is changed to a branch instruction 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 instruction and causes an illegal jump to another location, but the other is not executed, this error is undetectable. Therefore, if we assume that these cases are only detected when the number of executed master and shadow instructions are different, the estimated probability is:

$$\Pr_u\{(T_N, T_B)\} + \Pr_u\{(T_S, T_B)\} = \Pr\{E_h\} \cdot P_c = \{\frac{1}{N} \sum_{i \in M \cup S} \frac{h_{bi}}{n_b}\} \cdot \frac{N-l}{N} \tag{7}$$

Case 4: (T_B, T_N)

This case happens when a bit flip occurs in the opcode field. If the branch instruction is changed to a normal or store instruction, 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 will modify the value in `r1` and this error will propagate and be detected when the comparison instruction is executed. Thus, we can estimate

$$\Pr_u\{(T_B, T_N)\} \approx \Pr\{I_f \in B\} \cdot \Pr\{b_f \in W_{op}\} \cdot (1 - q_{reg}) = \frac{|B|}{N} \cdot \frac{|W_{op}|}{n_b} \cdot (1 - q_{reg}) \tag{8}$$

where q_{reg} represents the register utilization. For instance, if 90% of registers are always live during execution, the probability that the register specified by the register field has a live value is 0.9, thus ($1 - q_{reg}$) would be 0.1. If the program is fully optimized, the compiler tried to maximize the register utilization and ($1 - q_{reg}$) could be kept low.

Case 5: (T_B, T_S)

If the hamming distance between a branch and store instruction is greater than one, this case cannot happen assuming one single bit error. On the other hand, If the hamming distance is one, 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 will be detected, but if it is not used, it would go undetected. Thus we can estimate

$$\Pr_u\{(T_B, T_S)\} \approx \frac{1}{N} \cdot \{\sum_{i \in B} \frac{h_{si}}{n_b}\} \cdot (1 - q_{st}) \tag{9}$$

as an upper bound.

Case 6: (T_B, T_B)

There are two possible cases: corrupted in offset or register field. The probability that a fault occurs in the offset field and is not detected is:

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

Now, if the register field is corrupted, this fault may cause an error or no error. We will take an upper bound of the probability of missing the error since we cannot exactly predict the conditional jump as shown in the following examples:

```
Branch_greater r1 r3 label1
```

If the register field is changed and `r1` is altered to `r2`, it is difficult to tell that the control flow will be corrupted or not unless we know the values of the registers in run time. For example, if `r2` has positive values while `r1` has negative values during entire execution time, the branch will always be taken, which is incorrect. This analysis is only available by simulation and it will give us the exact number, but the cost is very expensive. Let's take another example:

```
Branch_not_equal r1 r2 label1
```

If the register field is changed and r1 is altered to r3, but if the value in r3 is still different from the value in r2, this fault does not change the control flow. If we assume equal probability for the numbers in the registers, we can guess that the probability of changing the control flow is almost zero since the probability that r2 and r3 have the same value is almost zero. As shown in the examples, 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 instruction is not detected at all, the probability in (11) will be an upper bound of missing this case.

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

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

$$\Pr_u\{(T_B, T_B)\} \approx \frac{|B|}{N} \cdot \frac{|W_{off}|}{n_b} \cdot \frac{N-l}{N} + \frac{|B|}{N} \cdot \frac{|W_{reg}|}{n_b} \tag{12}$$

Now, Summing up the equations from (6) to (12), the upper bound of probability that a fault occurs in the code segment and is not detected is estimated as

$$P_u \approx \Pr_u(T_S,T_S)+\Pr_u(T_S,T_N)+\Pr_u(T_N,T_B)+\Pr_u(T_S,T_B)+\Pr_u(T_B,T_N) + \Pr_u(T_B,T_S)+\Pr_u(T_B,T_B)$$

$$(13)\quad = \frac{|S|}{N}\{(1-\frac{|W_{op}|}{n_b})+k\cdot\frac{|W_{op}|}{n_b}\}\cdot(1-q_{st})+\{\frac{1}{N}\sum_{i\in M\cup S}\frac{h_{bi}}{n_b}\}\cdot\frac{N-l}{N}+\frac{|B|}{N}\cdot\frac{|W_{op}|}{n_b}\cdot(1-q_{reg})$$

$$= \frac{1}{N}\{\sum_{i\in B}\frac{h_{si}}{n_b}\}(1-q_{st})+\frac{|B|}{N}\cdot\frac{|W_{off}|}{n_b}\cdot\frac{N-l}{N}+\frac{|B|}{N}\cdot\frac{|W_{reg}|}{n_b}$$

11. References

[1] Chang, P.P., et al., "IMPACT: An Architectural Framework for Multiple-Instruction Issue Processors," *Proc. 18th Int'l Symposium on Computer Architecture (ISCA)* 19(3), pp.266-275, 1991.

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

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

[4] Cusick, J., "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors," *IEEE Transactions on Nuclear Science*, Vol. NS-32, No. 6, 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," *IEEE 8th FTCS*, pp. 3-9, 1978.

[7] Lala, P.K., et al., "On self-checking software design*," IEEE Proc. Of SOUTHEASTCON '91*, Williamsburg, VA, pp. 331-335, Apr. 7-10, 1991

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

[9] Lu, D. J., "Watchdog Processors and VLSI," *Proceedings of the National Electronics conference*, Vol. 34, pp. 240-245, Chicago, Illinois, October 27-28, June 1980.

[10] Mahmood, Aamer and E. J. McCluskey, "Watchdog Processor: Error Coverage and Overhead," *Digest, The Fifteenth Annual Int'l Symposium on Fault-Tolerant Computing (FTCS-15),* pp. 214-219, Ann Arbor, Michigan, June 19-21, 1985.

[11] Blough, D. M. and Gerald M. Masson, "Performance Analysis of a Generalized Concurrent Error Detection Procedure," *IEEE Trans. on Computers,* vol. 39, no. 1, pp. 47-62, Jan. 1990.

[12] Blough, D. M. "Fault Detection and Diagnosis in Multiprocessor Systems," Ph.D. dissertation. Baltimore: Johns Hopkins University, 1988.

[13] Lu, D. J., "Watchdog Processor and Structural Integrity Checking," *IEEE Transactions on Computers*, vol. C-31, No. 7, pp. 681-685, July 1982.

[14] Namjoo, M., "Techniques for Concurrent Testing of VLSI Processor Operation," *Digest of Papers, IEEE Test Conf.*, pp. 461-468, Nov. 1982.

[15] Shen, J. P. and M. A. Schuette, "On-line Self-Monitoring Using Signatured Instruction Streams," *Int'l Test Conf. proceedings*, pp. 275-282, Oct. 1983

[16] Eifert, J. B. and J. P. Shen, "Processor Monitoring Using Asynchronous Signatured Instruction Streams," *Digest of Papers, 14th Annual Int'l Conf. on Fault-Tolerant Computing*, pp. 394-399, June 1984.

[17] Wilken, K., and J.P. Shen, "Concurrent Error Detection Using Signature Monitoring and Encryption: Low-Cost Concurrent-Detection of Processor Control Errors," *Dependable Computing for Critical Applications, Springer-Verlag, A. Avizienis, J.C. Laprie (eds)*, Vol. 4, pp. 365-384, 1989.

[18] Wilken, K. and J. P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent-Detection of Processor Control Errors," *IEEE Trans. on Computer Aided Design*, Vol. 9, No. 6, pp. 629-641, June 1990.

[19] Saxena, N. R., and E. J. McCluskey, "Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums," *IEEE Trans. on Computers*, Vol. 39, No. 4, pp. 554-559, April 1990.

[20] Madeira, H. and J. G. Silvia, "On-line Signature Learning and Checking," *Dependable Computing for Critical Applications 2*, Springer-Verlag, J. F. and R. D. Schlichting (eds), pp. 395-420, 1992.

[21] Oh, N., Shirvani, P., and McCluskey, E. J.,"Control Flow Checking by Software Signatures," Center for Reliable Computing Technical Report, 1999.

[22] D. Reynolds and G. Metze, "Fault detection capabilities of alternating logic," *IEEE Transactions on Computers*, vol. C-27, pp.1093-1098, Dec. 1978

[23] Shedletsky, J.J. et al., "Error correction by alternate-data retry," *IEEE Transactions on Computers*, vol. C-27, no.2, pp. 106-112, Feb.1978.

[24] Takeda, K. et al. "Logic design of fault-tolerant arithmetic units based on the data complementation strategy," *10th International Symposium on Fault-Tolerant Computing,* pp. 348-350, Oct. 1980.

[25] Patel, J.H. et al. "Concurrent Error Detection in ALUs by REcomputing with Shifted Operands," *IEEE Transactions on Computers*, vol.C-31, no.7, pp. 589-595, July 1982.

[26] Yuang-Ming Hsu et al. "Time redundancy for error detecting neural networks," *Proceedings IEEE International Conference on Wafer Scale Integration*, pp. 111-121, Jan. 1995.

[27] Andrews, D., "Using executable assertions for testing and fault tolerance," *9th Fault-Tolerance Computing Symp*., Madison, WI, June 20-22, 1979.

[28] G. A. Kanawati, V.S.S. Nair, N. Krishnamurthy, J.A.Abraham, "Evaluation of Integrated System-Level Checks for On-Line Error Detection," *Proceedings IEEE International Computer Performance and Dependability Symposium*, pp. 292-301, 1996

[29] Christophe Rabejac, J-P Blanquart, and J-P Queille, "Executable Assertions and Timed Traces for On-Line Software Error Detection," *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, pp. 138-47, Jun, 1996.

[30] Miremadi, G., J. Karlsson, J. U. Gunneflo, and J. Torin, "Two Software Techniques for On-line Error Detection," *Digest of Papers, 22nd Annual Int'l Symp. on Fault_Tolerant Computing*, pp. 328-335, July 1992.

[31] Miremadi, G., J. TJ. Ohlsson, M. Rimen, and J. Karlsson, "Use of Time, Location and Instruction Signatures for Control Flow checking," *Proc. Of the DCCA-5 Int'l Conf., Springer-Verlag Series for Dependable Computing Systems*, Sep. 1995.

[32] J. C. Fabre, Y. Deswarte, J-C. Laprie, and D. Powell, "Saturation: Reduced idleness for improved fault-tolerance," *IEEE 18th FTCS* pp. 200-205, June 1988.

[33] G. Sohi, et al., "A study of time-redundant fault tolerance techniques for high-performance pipelined computers," *IEEE 19th FTCS*, pp. 436-443, Jun, 1989.

[34] Blough, D. M. and A. Nicolau, "Fault Tolerance in Super-scalar and VLIW Processors", *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems,* pp. 193-200, July 1992.

[35] Schuette, M. A., J. P. Shen, "Exploiting Instruction-Level Parallelism for Integrated Control-Flow Monitoring," *IEEE Trans. on Computers,* Vol. 43, No. 2, pp.129-140, Feb. 1994.

[36] Holm, J. G. and P. Banerjee, “Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions,” *Proc. International Conference on Parallel Processing (ICCP)*, pp. 192-195, 1992.

[37] Patterson, D. A. and J. L. Hennesy, “Computer Architecture A Quantitative Approach,” Morgan Kaufmann, 1996.

[38] Garey, M. R. and D. S. Johnson, “Computers and Intractibility: a Guide to the Theory of NP-Completeness,” New York: W. H. Freeman and Company, 1979.

[39] Giovanni De Micheli, “Synthesis and Optimization of Digital Circuits,” McGraw-Hill, 1994.

[40] Stallman, R. “Using and Porting GNU CC,” Free Software Foundation, 1998.

[41] Heinrich, J. “MIPS R4000 Microprocessor User’s Manual,” MIPS Technologies, Inc. 1994