SEU Characterization of Digital Circuits Using Weighted Test Programs

Philip P. Shirvani and Edward J. McCluskey

01-4

May 2001

Center for Reliable Computing
Gates Room # 239, MC 9020
Gates Building 2A
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract:

Estimating the single event upset (SEU) cross section of individual units of a digital system is important in designing a fault tolerant system and predicting its overall error rate. Hardware approaches to this SEU characterization are expensive and sometimes intrusive. Previous software approaches tend to use test programs targeting specific units. However, these methods are not suitable for all functional units because several units are always used during the execution of any program. Another method is to classify the errors based on their properties and associate each class with a specific functional unit. However, since an error can be caused by many different faults, error classification can only do a crude association of errors with functional units. Because of these limits, previous methods do not always yield good estimates.

In this report, a new method is proposed that uses weighted test programs in conjunction with multiple linear regression to alleviate some of the limits of previous software approaches. In this method, error classification is only used when the error source is unique. We also propose using the frequency dependency of SEU cross section in separating the cross sections of sequential and combinational logic circuits.

Funding:

This work was supported in part by the National Aeronautics and Space Administration (NASA) and administered through the Jet Propulsion Laboratory (JPL), California Institute of Technology under Contract No. 1216777.

Imprimaturi: Nirmal Saxena, Subhasish Mitra and Nahmsuk Oh

SEU Characterization of Digital Circuits Using Weighted Test Programs

Philip P. Shirvani and Edward J. McCluskey

CRC Technical Report No. 01-4

May 2001

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305

Abstract

Estimating the single event upset (SEU) cross section of individual units of a digital system is important in designing a fault tolerant system and predicting its overall error rate. Hardware approaches to this SEU characterization are expensive and sometimes intrusive. Previous software approaches tend to use test programs targeting specific units. However, these methods are not suitable for all functional units because several units are always used during the execution of any program. Another method is to classify the errors based on their properties and associate each class with a specific functional unit. However, since an error can be caused by many different faults, error classification can only do a crude association of errors with functional units. Because of these limits, previous methods do not always yield good estimates.

In this report, a new method is proposed that uses weighted test programs in conjunction with multiple linear regression to alleviate some of the limits of previous software approaches. In this method, error classification is only used when the error source is unique. We also propose using the frequency dependency of SEU cross section in separating the cross sections of sequential and combinational logic circuits.

Key Words and Phrases: single-event upset, SEU characterization, fault tolerance, SEU cross section, SEUs in combinational and sequential circuits.

Table of Contents

1. INTRODUCTION

Single-Event Upsets (SEUs) are transient faults in electronic systems caused by radiation such as alpha particles and cosmic rays. SEUs are a major cause of concern in a space environment. They have also been observed at ground level [Ziegler 96] and are being addressed by industry for sub-0.25μm CMOS technologies [Dai 99][Hareland 00][Seifert 01]. An example effect is a *bit-flip* — an undesired change of state in a bistable or storage element, also referred to as a *soft error*. In a combinational circuit, e.g., an arithmetic logic unit (ALU), the effect is a transient voltage pulse that can lead to incorrect output. The focus of this report is on techniques for characterizing the effects of SEUs during a radiation test that is conducted on ground or in space. We propose a technique that can be used for any digital circuit but describe it in the context of processor chips.

The error rate in an entire processor chip can be determined by building a system with two processors and irradiating one of them while both processors execute a program that extensively exercises the internal logic of the processors and generates outputs for comparison. A much more difficult task is to determine the relative rate at which various areas of logic within the chip are upset and the specific logical effects that occur. For a microprocessor, different areas of logic are: register file, ALU, multiply and divide unit and floating-point unit (FPU), on-chip caches, etc. The fault-tolerance technique that is chosen for protecting each unit against SEUs depends on many factors such as: type of the unit (memory, logic, etc.) and the sensitivity of the unit to SEUs in the target environment.

Estimating the SEU rate in a digital circuit is important in designing a fault-tolerant system and systems that are exposed to radiation. If the error rate in each functional unit is known, efforts can be focused on the units that are more vulnerable to SEUs. Once fault tolerance has been added to reduce the effects of SEUs in those units to an acceptable level, we can focus on less sensitive and critical units and continue this procedure until the overall system reliability requirements are met.

There is a second motive for estimating the error rate of individual functional units. The SEU sensitivity of a processor depends on the application program that is being executed on the processor [Elder 88][Kimbrough 94][Koga 85][Shaeffer 92][Thomlinson 87][Velazco 92]. This is due to the different utilization of the processor functional units by different programs. If the error rate in each functional unit is known, one can predict the error rate of different application programs without having to execute them during a radiation test.

SEU characterization can be done using hardware or software techniques. The duplication method described above is an example of a hardware technique. The duplication can be done at chip level where corresponding pins are compared for mismatch. One can also design a test chip in which each functional unit is duplicated and extra comparators are inserted for localizing errors. Hardware approaches have the advantage of providing good controllability and observability for testing. However, they are expensive and can be intrusive (the additional circuitry may have side effects). For example, a test processor with duplicated internal units may not be able to run at the same frequency as the original processor. Another hardware approach is using the scan chains available in many digital circuits for manufacturing test [Abramovici 90][McCluskey 86]. In this approach, patterns are continuously scanned in and out of the flip-flops and latches while the chip is irradiated. The scanned-out patterns are compared with the scanned-in patterns and checked for any bit-flips. Scan chains provide good controllability and observability for the scanned elements and have been used in radiation testing of processors [Brown 97].

In a software approach, no hardware is added or modified. The processor executes a set of test programs that exercise all the internal units and detect deviation from normal operation. The advantage of software approaches is that they are not intrusive, i.e., the operating conditions are the same as that of the real application. The biggest constraint of software approaches is the limited controllability and observability that they provide for error location.

In a direct software method, test programs target one functional unit at a time. However, direct location of errors for characterizing and estimating the error rate in various areas of logic is not always easy because:

- A program cannot be confined to one or two units. Many functional units – such as the instruction fetch unit – are involved in execution of any program regardless of the type of the program.
- Different faults can cause the same erroneous state; therefore the fault source cannot be uniquely identified by the error.
- The time between when an error occurs and when it is detected (detection latency) will allow errors to propagate, making it hard to trace the error back to its source.
- It may be difficult to read out the internal state after the error has occurred because not all the flip-flops are accessible through software, or the processor may be hung.

In general, a direct software method may be inaccurate and sometimes impossible. In this report, we propose an indirect software method. In this method, a set of carefully designed test programs exercise a set of functional units and the error rate in each of the functional units is derived from the overall chip error rates observed in execution of the programs. A system of linear equations is formed where the error rates in functional units are the unknowns. We solve the equations using linear regression and obtain estimates of the error rates within a certain confidence interval. This method may be particularly useful for space-based experiments where there is no artificial fault injection and the observability of the state is very limited. While we may never find out exactly where all errors come from, it is possible to gain insight as to where most of them occur using a set of carefully designed test programs.

Note that if a functional unit has good observability and controllability (such as a register file), a direct software method works well and we take full advantage of it. The technique proposed in this report works in parallel with direct methods and fills the gaps where direct methods fall short.

We continue this report by giving a background on SEU related terminology, reviewing the previous work and listing our contributions in Sec. 2. Section 3, presents our approach and how the test programs are written. We use the MIPS R3000 architecture for illustration. The technique can be similarly applied to other processors and digital circuits. We also discuss the difficulties and caveats in writing the test programs in Sec. 3. In Sec. 4, we show how errors in bistables can be distinguished from errors in combinational logic by operating a sequential circuit at different clock frequencies. Sources of measurement errors, solutions for reducing these errors, and limitations of our technique are discussed in Sec. 5. To test the proposed technique, we conducted fault injection simulations. The simulation setup and results are presented in Sec. 6. Sections 7 and 8 summarize and conclude the report.

2. BACKGROUND AND PREVIOUS WORK

A *Single Event Upset* (SEU) is a change of state or a transient error induced in a device by an ionizing particle such as a cosmic ray or proton[1]. Circuit parameters and particle energy determine whether a particle can cause an SEU or not. The minimum charge that a particle must deposit in a node to cause an SEU is denoted by Q_{crit} and depends on node parameters such as capacitance and voltage. *Linear Energy Transfer* (LET) is a measure of the energy transferred to the device per unit length of the material through which the ionizing particle travels[2]. The minimum LET that can deposit the Q_{crit} and cause an SEU is called *LET threshold* (LET_{th}).

To calculate the rate at which SEUs occur in the circuit, we start with the rate at which particles hit the circuit. Particle hit rate, or *flux*, is measured in (particles/cm^2)/sec. Depending on the particle energy and the sensitivity of the circuit, a particle may or may not cause an SEU in the circuit. For a fixed particle energy, we designate the sensitivity of the circuit with P_{err} = *Prob* (error | particle hit). This sensitivity is less than one because not all parts of a chip are sensitive to particles (e.g., empty areas or areas with only wires and no transistors underneath). If the chip area is A_{chip}, the total sensitive

[1] There are many other types of radiation effects, for example, *Total Ionizing Dose* (TID) and *Single Event Latchup* (SEL), but they are not the subject of this report.
[2] The common unit for LET is MeV/mg/cm^2 of material (Si for substrate and SiO_2 for transistor gate in MOS devices); MeV = Million electron-Volts, mg = milligram.

area of the chip will be $\sigma = P_{err} \times A_{chip}$. σ is known as the *cross section* and is the device upset response to ionizing radiation. For an experiment with a specific LET, σ = # errors / (particle fluence), where particle fluence is particles/cm^2. The units for cross section are cm^2 (number of errors and particles are unitless). If we know the cross section of the chip, then the error rate will be $\lambda = flux \times \sigma$. The goal of this report is to devise a test technique for estimating the cross section of each individual functional unit of a digital circuit. The corresponding error rates can be easily derived by multiplying the cross sections by the particle flux of the given environment.

A common method for estimating cross sections is artificial fault injection on the ground where a device is exposed to heavy ions or proton using radioactive material or particle accelerators. In a duplicated system, one processor unit can be exposed to radiation while another one is shielded and isolated from radiation (acting as a golden part) [Elder 88][Moran 96]. In some radiation methods, it is possible to focus the radiation to a specific part of the circuit under test (CUT) and isolate the rest of the system. For example, the beam can be focused only on the memory or on the processor. With this setup, the cross section of each component can be estimated separately. This method is used in [Hiemstra 99] for radiation testing of the Pentium II microprocessor where the three dies inside a Pentium II package (the processor core, the L2 tag SRAM and the L2 cache) are irradiated separately. Fine-grained fault injection (in regions inside a die) is also possible by using a laser beam [Buchner 96][Buchner 97][Melinger 94]. However, a major limit of a laser beam is that it cannot penetrate metal. Therefore, if a sensitive region is covered by metal, the laser will not be able to measure its SEU threshold [Melinger 94]. As another example, the test chip in [Liden 94] was designed and manufactured with its sequential and combinational logic circuits physically separated on the die. During one phase of irradiation, the sequential part was covered to observe the upset rate in the combinational part as a comparison to the total upset rate. Physical separation of units is not possible when characterizing a commercial chip. However, covering some units with an epoxy layer has been used in testing Alpha processors [Seifert 01]. Apart from cost related issues, these methods may not be feasible or applicable in an experiment, for example, when characterizing a commercial off-the-shelf (COTS) component in space. In that case, software approaches could be the better choice or even the only choice. This report presents a technique for SEU characterization that is implemented in software and can be used stand-alone or in combination with other software and hardware techniques.

Since the observability of general-purpose registers is higher than other parts, register file testing is the most popular test in a software approach. Some studies used the cross section of latches to infer the cross section of the whole chip [Asenek 98][Brown 97][Sexton 90]. The scan chain approach has been used in [Brown 97] for estimating the cross section of latches[3]. However, the experiment in [Velazco 92] shows that a cross section derived from testing only the registers is not representative of cross section for an application program and is insufficient to characterize the circuit behavior.

Error classification is another issue in SEU characterization of digital circuits. Errors collected during a fault injection experiment are classified based on their behaviors or properties. For example, errors are classified as bit errors, word errors and complex errors in [Elder 88] and as data errors, sequencing errors, address errors and other errors in [Velazco 92]. Then each type may be attributed to one or more functional units. For example, catastrophic errors such as unexpected jumps are attributed to the program counter (PC) and instruction decoder. In [Cusick 85], where the data, address and control pins of a Z80 microprocessor are checked for errors, single-bit errors in data and address pins are associated with bit-flips in general purpose registers, and single-bit errors in control pins are associated with internal latches. These associations are clearly crude and not accurate.

Typically, in order to characterize a processor chip, a combination of test programs are used in which each program targets a specific unit and the combined set exercises the whole chip. For example, [Kouba 97] uses a set of ALU, I/O and FPU intensive programs in order to exercise the whole chip. This method could yield an average and a

[3] One should be careful in using the cross section of the register file for the internal latches. In a test of the Z80 microprocessor, the internal latches show a larger cross section than the register file [Cusick 85].

range for the cross section for the whole chip. However, since many of the units of a processor are always used, it cannot give a separate cross section for each unit. Another approach is to use the application that will eventually execute on the system. For example, the “system” routine, reflecting a worst-case space flight application, was used for most test runs of the Intel 80486 in [Moran 96]. However, in general, the final application may not be available for radiation tests. Moreover, it is desirable to be able to predict the cross section of a processor for different applications without having to run them in radiation experiments.

In [Shaeffer 92], different programs are used with different instruction mixes from the instruction set architecture. The percentage of instruction set used in each program (e.g., 40 out of 120 instructions available in an instruction set architecture) is used as an estimate of the active chip area. This method does not consider the dynamic activity of the units. Some instructions may repeat frequently and some may not. Therefore, the corresponding units may be active most of the time or just for a short time.

In summary, the limits of previous approaches are as follows:

- Hardware approaches are expensive, intrusive and not always possible.
- It is not always possible to target only one functional unit in a test program.
- Error classifications cannot accurately localize the source of errors.

In this report, we propose a method that attempts to go beyond these limits and enhance the SEU characterization process of digital circuits. The contributions of this work are:

- A new software method for SEU characterization of digital circuits using weighted test programs, system of linear equations and multiple regression, which yields more accurate cross sections of individual units than previous methods and does not depend on ambiguous error classifications.
- Analysis of weighted test programs and the issues and caveats in writing them.
- A new technique for separating the cross section of bistables and combinational logic in SEU characterization of sequential circuits using clock frequency dependency of SEU cross section.

3. THE WEIGHTED TEST PROGRAMS METHOD

The fact that the cross section of a processor depends on the program executed on the system is shown in many studies [Elder 88][Kimbrough 94][Koga 85][Shaeffer 92][Thomlinson 87][Velazco 92]. SEU vulnerability depends not only on processor technology and physical element cross section, but also on the duty factors (utilizations) imposed by the instruction stream [Elder 88][Koga 85][Thomlinson 87]. The *duty factor* of each unit is the percentage of time the unit is active. For a storage element, this is the percentage of time that the element holds a live value (a value that is going to be used sometime during program execution). Multiplying the average cross section per bit by the total number of bits in the device gives an overestimate of the device cross section because all bits are not used at all times. A better approach is to estimate the number of "live and relevant" registers (registers whose SEU will cause observable errors during the relevant program execution) and use that as the number of bits [Koga 85]. A software tool is presented in [Asenek 98] for estimating the duty factors of the registers in a program. In [Elder 88], it is suggested that by knowing the cross section of each unit, σ_i, and their associated duty factors, f_i, the total SEU cross section, σ_T, of the processor executing that specific program can be predicted by:

$$\sigma_T = \sum_i \sigma_i f_i$$

This predication can be done if the σ_i's and f_i's are known. The tool presented in [Asenek 98] can be extended to estimate the f_i's of different units. In the next section, we propose a method that uses this basic formula for estimating the σ_i's (cross section of each unit).

3.1 System of Linear Equations

Consider a test program that calculates a sum over the contents of 10 general-purpose registers (Fig. 3.1 (a)) and assume we want to estimate the cross sections of the register file and the ALU. We can break the cross section of the processor executing this program into three parts:

$$\sigma_T = \sigma_{reg} f_{reg} + \sigma_{ALU} f_{ALU} + \sigma_{remainder} f_{remainder} \quad (1),$$

where σ_{reg} and σ_{ALU} are the cross sections of the register file and the ALU, respectively, and $\sigma_{remainder}$ lumps the cross section of the remainder of the active units of the processor (fetch unit, decode unit, etc.). f_{reg}, f_{ALU} and $f_{remainder}$ are the duty factors of the corresponding units. 10 registers are live during almost the entire execution. If the processor has 32 general purpose registers, then f_{reg} is 10/32 for this program.

```
init r1, .., r10;
sum = 0;
outer loop {
  loop1 n times {
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r10
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r10
  }
  loop2 m times {
    add  r0,sum,r1
    add  r0,sum,r2
    ...
    add  r0,sum,r10
    add  r0,sum,r1
    add  r0,sum,r2
    ...
    add  r0,sum,r10
  }
}
check sum;
```

(a)

```
init r1, .., r20;
sum = 0;
outer loop {
  loop1 n times {
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r10
    add  sum,sum,r11
    add  sum,sum,r12
    ...
    add  sum,sum,r20
  }
  loop2 m times {
    add  r0,sum,r1
    add  r0,sum,r2
    ...
    add  r0,sum,r20
    add  r0,sum,r11
    add  r0,sum,r12
    ...
    add  r0,sum,r20
  }
}
check sum;
```

(b)

Figure 3.1 Pseudocode of a test program for ALU and register file: (a) using 10 registers, (b) using 20 registers.

During the execution of *loop2* in Fig. 3.1 (a), the destination register for the `add` instructions is *r0* which is a constant 0 in MIPS architecture. In other words, we are ignoring the results that are produced by the ALU in this loop. The ALU output is used in every cycle in *loop1* but is ignored in *loop2*. Therefore, f_{ALU} is adjustable and is equal to *n*/(*n*+*m*), where *n* and *m* are the loop counters in Fig. 3.1. Note that the ALU is also used for incrementing and checking the loop counter variable, and some registers are allocated for the loop counter and sum value. One can count these activities in f_{ALU} and f_{reg} or in $f_{remainder}$. For this example test, the cross section of the remainder of the processor (other than ALU and register file), $\sigma_{remainder}$, is not the target and $f_{remainder}$ is unknown in these programs. However, we write the test programs such that $f_{remainder}$

remains the same among the test programs and can be replaced by a constant number in equation (1). We assume, without any loss of generality, that $f_{remainder}$ is 1, knowing that the derived $\sigma_{remainder}$ will be a fraction of the actual $\sigma_{remainder}$.

Next, consider the test program in Fig. 3.1 (b) that uses 20 registers instead of 10. For this program, f_{reg} is 20/32. We can modify the *m* and *n* parameters in one of these test programs such that it has a different f_{ALU}. This will give us a third program for the third equation. Once we run these three programs while the processor is irradiated, and measure the error rate for each of them, we get three equations with three unknowns, namely σ_{reg}, σ_{ALU} and $\sigma_{remainder}$. Solving these equations will give us the three unknown cross sections.

The basis of our proposed method is as follows. We write a set of programs that exercise a subset of the functional units of the chip, each with known and different duty factors. We call these *weighted test programs*. These programs are run while the system is irradiated and the total cross sections are measured by counting the total number of errors that occur in the execution and using the formula $\lambda = flux \times \sigma$. Finally, the resulting system of linear equations is solved to get the individual cross sections. We call this the weighted test program (WTP) method.

For example, if we break the cross sections into 3 parts (*x*, *y* and *z*) and write 3 programs that exercise these parts with different duty factors (a_i's, b_i's and c_i's), we get:

$$\begin{aligned}\sigma_1 &= a_1 x + b_1 y + c_1 z\\ \sigma_2 &= a_2 x + b_2 y + c_2 z\\ \sigma_3 &= a_3 x + b_3 y + c_3 z\end{aligned}$$

This can be solved for the unknowns assuming that there are no linear dependency between the equations. In the Appendix, we discuss how the estimates can be enhanced by having more equations than unknowns.

Duty factors take real values between 0, for no activity, and 1, for active all the time ($0 \le f_i \le 1$). Controlling the duty factors in a test program can only be done with careful programming and, in many cases, is not trivial. In the rest of this section, we present the generic structure of a test program and discuss issues that a programmer should be careful about and how they can be solved. We will also discuss the limits, some caveats and how test programs can be enhanced.

3.2 A Generic Test Program Structure

The structure used in the test programs in Fig. 3.1 can be expanded to a generic structure for testing multiple functional units. This structure is shown in Fig. 3.2.

```
initialize x registers;
R1 = R2 = .. = Rk = 0;
outer loop {
  loop1 n1 times {  // test FU1
    ...
    ...
    ...
  }  // accumulate result in R1
  loop2 n2 times {  // test FU2
    ...
    ...
    ...
  }  // accumulate result in R2

  ...

  loopk nk times {  // test FUk
    ...
    ...
    ...
  }  // accumulate result in Rk
}
check R1, R2, .., Rk;
```

Figure 3.2 Pseudocode of a generic test program for SEU characterization using the WTP method.

Each inner loop targets one functional unit (FU) and maximizes the duty factor of that unit. However, this does not necessarily mean that a unit that is exercised in one loop should not be active in any other loop (an example can be seen in Fig. 3.7). The cross section equation for this generic format is:

$$\sigma_T = \sigma_{reg} f_{reg} + \sigma_{FU1} f_{FU1} + \sigma_{FU2} f_{FU2} + ... + \sigma_{FUk} f_{FUk} + \sigma_{remainder} f_{remainder} \cdot$$

The duty factor of the register file, f_{reg}, depends on the number of registers used in the whole program and the percentage of time (total execution time) each register holds a live value. For combinational circuits, the duty factor can be estimated as follows. Denote the execution time of each inner loop *i* with T_i, and the duty factor of functional unit *j* during the execution of loop *i* with $f_{j,i}$. Then the overall duty factor of functional unit *j*, f_{FUj}, will be:

$$f_{FUj} = \sum_{i=1}^{k} \frac{T_i}{T_{outer\_loop}} f_{j,i}\text{, where } T_{outer\_loop} = \sum_{i=1}^{k} T_i\ .$$

We use this structure for the examples used in the following sections.

3.3 Matching Duty Factors

The two test programs in Fig. 3.1 are designed to have different register file and ALU duty factors (f_{reg} and f_{ALU}) but the same $f_{remainder}$. Two things were considered in writing these programs to fix the duty factors that are not supposed to change ($f_{remainder}$ in this case).

First, the bodies of the inner loops in Fig. 3.1 (a) are repeated twice so that it matches with that in Fig. 3.1 (b) in terms of number of instructions and execution clock cycles. This will make the branch frequency and program sizes the same.

Second, the structure of *loop1* is used for *loop2* with minimum changes so that f_{reg} and $f_{remainder}$ are almost the same between these loops. In *loop2*, a `nop` operation could be used instead of the `add` instructions. However, with `nop`, a different instruction is decoded and the register file is not accessed. Hence, an `add` instruction that reads the register file is a better choice.

We can create test programs with a desired f_{ALU} by changing the *n* and *m* parameters. f_{reg} can also be changed by using the same program structure but using a different number of registers. For example, for f_{reg} = 5/32, the test program is shown in Fig. 3.3.

```
init r1, .., r5;
sum = 0;
outer loop {
  loop1 n times {
    add  sum,sum,r1
    add  sum,sum,r2
    add  sum,sum,r3
    add  sum,sum,r4
    add  sum,sum,r5
    add  sum,sum,r1
    ...
    add  sum,sum,r5
  }
  loop2 m times {
    add  r0,sum,r1
    add  r0,sum,r2
    add  r0,sum,r3
    add  r0,sum,r4
    add  r0,sum,r5
    add  r0,sum,r1
    ...
    add  r0,sum,r5
  }
}
check sum;
```

Figure 3.3 Pseudocode of a test program for ALU and register file using 5 registers.

3.4 Instruction Latencies and Stall Cycles

In this section, we look at an example that demonstrates the importance of micro-architecture details. Clearly, these details vary among different processors. Test programs for one processor may not be directly portable to another processor due to these differences. We use the multiplication unit in a MIPS 3000 processor. Similar attention should be paid when writing a test for other units and other architectures.

```
outer loop {
  loop1 n times {
    mult  R1,R1,r1
    mult  R1,R1,r2
    ...
    mult  R1,R1,r20
  }
  loop2 m times {
    mult  r0,R1,r1
    mult  r0,R1,r2
    ...
    mult  r0,R1,r20
  }
}
check R1;
```

(a)

```
mult  R1,r1
mflo  R1
nop
nop
```

(b)

```
outer loop {
  loop1 n times {
    add  R1,R1,r1
    add  R1,R1,r2
    ...
    add  R1,R1,r20
  }
  loop2 m times {
    mult  R2,R2,r1
    mult  R2,R2,r2
    ...
    mult  R2,R2,r20
  }
}
check R1 & R2;
```

(c)

Figure 3.4 (a) Pseudocode of a test program with adjustable f_{Mult}, (b) how each ‘`mult`’ instruction in (a) is translated into MIPS 3000 instructions, (c) pseudocode for testing ALU and multiplier together.

To test the multiplier, a test program similar to the program in Fig. 3.1 can be written that has adjustable f_{Mult} (Fig. 3.4(a)). The equation for this program is:

$$\sigma_T = \sigma_{reg} f_{reg} + \sigma_{Mult} f_{Mult} + \sigma_{remainder} f_{remainder} .$$

In the MIPS R3000 architecture, each of the multiplication instructions will be translated to 4 instructions as shown in Fig. 3.4(b). In this architecture, the result of a multiplication operation is written in two special registers, *hi* and *lo*. The '`mflo`' instruction moves the content of the *lo* register into the register assigned to sum. According to the restrictions of the architecture, the results will not be available until two cycles later, and since the next multiplication is dependent on the result, two '`nop`' instructions are also inserted between each two multiplications. This shows that a programmer should look at a test program at machine language level to see exactly what instructions are generated from a high level language source code. Moreover, the multiplication causes 10 stall cycles during its execution. During the stall cycles, the pipeline stages are frozen (the flow of instructions down the pipeline is stopped) and depending on the microarchitecture, no instructions may be fetched or decoded. In other words, the activities of many functional units may be zero and only state holding registers and some control logic are active during stall cycles. Overall, $f_{remainder}$ will be lower than 1 or what we had for the ALU test programs. This is very important in estimating duty factors for each test program.

As shown in Fig. 3.4(c), we may combine the test programs in Fig. 3.1(b) and Fig. 3.4(a) to test the ALU and multiplier together. However, due to extra move and '`nop`' instructions, and the stall cycles, it is clear that the simple $f_{ALU} = n/(n+m)$ relation is not true for this test. Similar details have to be considered when testing other long latency operations such as division or floating-point operations.

As we can see, careful attention to the details in low level source code and microarchitecture design is very important for deriving accurate duty factors of functional units in each test program. The process becomes more complicated in processors that have features such as register renaming, superscalar execution and out-of-order execution. An example is analyzed in the next section. Most processor vendors have a suite of assembly level verification tests that target particular functional units. These tests can be used as a starting point for writing SEU characterization tests.

3.5 A Superscalar Microprocessor Example

Assume we run the test in Fig. 3.4(c) on the PowerPC 750. This microprocessor is a 2-way issue machine with 2 fixed-point (integer) units, FXU1 and FXU2 [Motorola 97]. FXU1 can do arithmetic, shift, rotate, and logical operations as well as multiplication and division. FXU2 can do only the arithmetic, shift, rotate, and logical operations. Figure 3.5(a) shows how the program in Fig. 3.4(c) would execute on this processor. In *loop1*, the first two `add` instructions are issued to the two integer units but the second `add` has to wait for the result of the first add. In the next cycle, the second `add` will execute in FXU2 while the third `add` is issued to FXU1. This pattern happens for the following `add`'s and the instructions get split between the two integer units in this loop. Therefore, the measured cross section for ALU will be the average of the cross sections of the ALU1 and ALU2 (subunits of FXU1 and FXU2). In *loop2*, since FXU2 does not do multiplications, all the instructions execute on FXU1, separated with *c* cycles, were *c* depends on the throughput of the multiplier. If the integer units were identical, the same split would occur for the `mult` instructions, too.

```
outer loop {
  loop1 n times {
    add  R1,R1,r1   // FXU1, cycle x
    add  R1,R1,r2   // FXU2, cycle x+1
    add  R1,R1,r3   // FXU1, cycle x+2
    add  R1,R1,r4   // FXU2, cycle x+3
    ...
    add  R1,R1,r19  // FXU1, cycle x+18
    add  R1,R1,r20  // FXU2, cycle x+19
  }
  loop2 m times {
    mult  R2,R2,r1  // FXU1, cycle y
    mult  R2,R2,r2  // FXU1, cycle y+c
    ...
    mult  R2,R2,r20 // FXU1
  }
}
check R1 & R2;
```

(a)

```
outer loop {
  loop1 n times {
    add  R1,R1,r1   // FXU1, cycle x
    add  R3,R3,r1   // FXU2, cycle x
    add  R1,R1,r2   // FXU1, cycle x+1
    add  R3,R3,r2   // FXU2, cycle x+1
    ...
    add  R1,R1,r20  // FXU1, cycle x+19
    add  R3,R3,r20  // FXU2, cycle x+19
  }
  loop2 m times {
    mult  R2,R2,r1  // FXU1, cycle y
    add   R3,R3,r1  // FXU2, cycle y
    mult  R2,R2,r2  // FXU1, cycle y+c
    add   R3,R3,r2  // FXU2, cycle y+c
    ...
    mult  R2,R2,r20 // FXU1
  }
}
check R1 & R2 & R3;
```

(b)

Figure 3.5 Execution of test programs on PowerPC 750: (a) alternating use of the integer units, (b) simultaneous use of the integer unit.

If the two ALUs are not identical, their cross sections can be separated using the test program in Fig. 3.5 (b). In this test, a set of `add` instructions are interleaved with the original ones with no dependency between the two sets of additions. In this case, two add instructions will execute in each cycle keeping both ALUs active all the time in *loop1*. By ignoring the result of the second set of additions (R3) at the end of the program, the cross section of ALU2 will be excluded from the measured total cross section (similar to *loop2* in Fig. 3.1(b)) and the estimated σ_{ALU} will be only for the ALU1. As will be explained in Sec. 5, the cost of ignoring R3 is some loss in error detection coverage. To avoid this cost, R3 is checked, too. Moreover, some `add` instructions are also added to *loop2*. If we use the body of *loop2* in Fig. 3.5(a) for the test in Fig. 3.5(b), then the duty factors of the two ALUs will always be exactly the same. The result is linear dependency in the cross section equations that will make the corresponding cross sections inseparable. To make the duty factors of the two ALUs different, `add` instructions are inserted between the multiplications of *loop2* in Fig. 3.5(b) to increase the duty factor of ALU2 and make it linearly independent of the duty factor of ALU1. The number of `add` instructions added to this loop can be changed to get different programs with different duty factors.

3.6 Cache Effects

On-chip caches usually contain most of the transistors in a processor chip and dominate the overall cross section of the chip [Hiemstra 99][Moran 96][Velazco 92]. In order to separate the cross sections of CPU and caches, the system may be tested in two configurations during radiation tests: with caches enabled and with caches disabled [Hiemstra 99][Moran 96][Velazco 92] (note that there will be more configurations if there is more than one level of cache on chip). However, disabling the caches has a major effect on CPU activity. With no caches, every memory access (instruction and data) is followed by several stall cycles. As discussed in Sec. 3.4, this results in lower activity in CPU functional units (smaller duty factors). Therefore, the cross section measured for CPU with disabled caches is a fraction of the cross section during normal operation (with enabled caches)[4]. This may be a minor factor with level 2 cache disabled and level 1 cache enabled because the cache miss rate may be low (especially if the test programs fit in level 1 cache), but is clearly a major factor with all caches disabled. The cache effects can be considered as changes in duty factors. Alternatively, if the caches have parity bits or error correcting code (ECC), the processor can be tested with these protection features enabled and disabled ([Hiemstra 99] uses this method as well as disabling the caches). When the ECC is disabled, the cache SEUs will contribute to the overall cross section and when ECC is disabled, the cache SEUs are masked. The difference of the cross sections estimated in the two modes of operation is the cross section of the cache. The advantage of this method is that duty factors in CPU remain the same as in normal operation.

In general, when testing the functional units of a CPU with caches enabled, it is better to design the code and data size of the test programs such that they fit in the caches for most of the execution time. This way, cache misses will not affect the duty factors. However, if the caches do not have ECC or parity, we need to minimize the cache utilization so that the overall cross section is not dominated by cache SEUs. This may be achieved by writing test programs that have small code and data sizes. We can vary the duty factor of the caches by changing the code and data size. For example, in the test programs of Fig. 3.6 the code size of the program in part (b) is about twice the code size of the program in part (a). The cross section equation for these tests is:

$$\sigma_T = \sigma_{reg} f_{reg} + \sigma_{ALU} f_{ALU} + \sigma_{I-cache} f_{I-cache} + \sigma_{remainder} f_{remainder},$$

where $\sigma_{I-cache}$ is the instruction cache cross section and $f_{I-cache}$ is the corresponding duty factor.

[4] It is not clear whether this effect is taken into account in papers reporting cross section numbers for processors with caches.

```
Sub1()
{
  init r1, .., r20;
  sum = 0;
  outer loop {
    loop1 n times {
      add  sum,sum,r1
      add  sum,sum,r2
      ...
      add  sum,sum,r20
    }
    loop2 m times {
      add  r0,sum,r1
      add  r0,sum,r2
      ...
      add  r0,sum,r20
    }
  }
  check sum;
}

main()
{
  loop 20 {
    Sub1();
  }
}
```

(a)

```
Sub1()
{
  init r1, .., r20;
  sum = 0;
  outer loop {
    loop1 n times {
      add  sum,sum,r1
      add  sum,sum,r2
      ...
      add  sum,sum,r20
    }
    loop2 m times {
      add  r0,sum,r1
      add  r0,sum,r2
      ...
      add  r0,sum,r20
    }
  }
  check sum;
}

Sub2()  // exact copy of Sub1
{
  init r1, .., r20;
  sum = 0;
  outer loop {
    loop1 n times {
      add  sum,sum,r1
      add  sum,sum,r2
      ...
      add  sum,sum,r20
    }
    loop2 m times {
      add  r0,sum,r1
      add  r0,sum,r2
      ...
      add  r0,sum,r20
    }
  }
  check sum;
}

main()
{
  loop 10 {
    Sub1();
    Sub2();
  }
}
```

(b)

Figure 3.6 Pseudocode of a test program with different instruction cache duty factor.

3.7 Cross Section Granularity

One of the issues with software methods for SEU characterization is their limited cross section granularity. Perhaps $\sigma_{remainder}$ is the best example because it lumps many different smaller cross sections into one. Some cross sections that are lumped into one cross section in one test program may be separated in another test program. However, many of these smaller cross sections correspond to units that are involved in the execution of any program and their activities are tightly coupled. For example, the duty factor of the decoder and the fetch unit may increase or decrease by the same amount. Therefore, it may be hard or even impossible to separately estimate these cross sections using software.

As an example, let us consider the SEUs in the address decoder of a register file. The resulting transients can lead to reading or writing an incorrect register during a register file access. The register address decoder may be considered part of σ_{reg} or $\sigma_{remainder}$. In the test programs in Fig. 3.1, if all registers are initialized with the same value, reading an incorrect register will not cause an error and therefore, the SEU will not be detected. If the registers are initialized with different values, reading an incorrect register will result in an incorrect sum and will be detected. However, in this test program, such an error is indistinguishable from an ALU error. Therefore, a different test program has to be written for estimating the cross section of the register address decoder. One such test is shown in Fig. 3.7. We assume a pipelined processor architecture with operand bypassing as shown in Fig. 3.8 (this does not apply to all architectures). In *loop1* of this program, the register file is read in every cycle, but in *loop2*, for every other add instruction, all the operands are provided by pipeline bypasses and the register file is not read. In other words, the duty factor of the register file address decoder (of one read port) is 1 in *loop1* and ½ in *loop2*.

```
init r1, .., r10;
sum = 0;
outer loop {
  loop1 n times {
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r10
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r10
  }
  loop2 m times {
    add  sum,sum,r1
    add  sum,sum,sum
    add  sum,sum,r2
    add  sum,sum,sum
    ...
    add  sum,sum,r10
    add  sum,sum,sum
  }
}
check sum;
```

Figure 3.7 Pseudocode of a test program for measuring the cross section of register file address decoder**.**

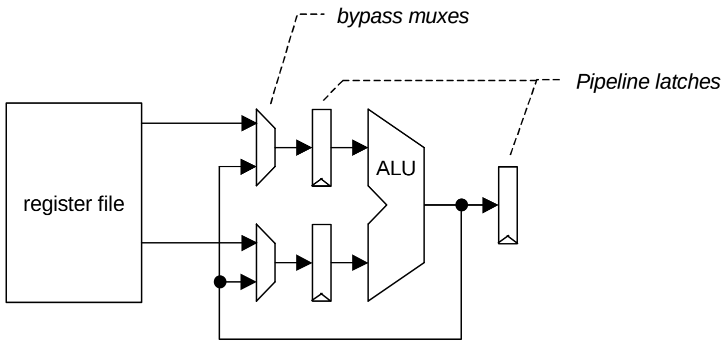

Figure 3.8 Block diagram of a simple pipelined architecture with operand bypassing.

As another example, consider the register file cross section, σ_{reg}. There are two types of test for a register file: static and dynamic. In a static test, registers are initialized with fixed values, and after a period of no register file access, they are checked for correct values. In a dynamic test, registers are continuously read and written throughout the test and the contents are checked. A register file has a similar structure as an SRAM. Simulations and experiments have shown that the cross section of an SRAM depends on the frequency of read and write operations [Buchner 96]. Therefore, a cross section derived from a static test is different from a cross section derived from a dynamic test. A heuristic way to capture this behavior is to write:

$$\sigma_{reg} f_{reg} = \sigma_{reg\_static} f_{reg\_static} + \sigma_{reg\_dynamic} f_{reg\_dynamic} ,$$

where σ_{reg_static} represents the cross section of the register file when it is not being read or written, and $\sigma_{reg\_dynamic}$ represents the additional cross section that it exhibits when it is read or written. A test program that has adjustable f_{reg_static} and $f_{reg\_dynamic}$ is shown in Fig. 3.9. In loop1, 20 registers are being read and the register that holds the sum value is written back in every cycle. No read or write is performed in loop2. Note that, as shown in Fig. 3.9, the register SEUs can be separated from ALU errors or other errors by printing the register values when the final sum is incorrect. This is an example of a case (mentioned in Sec. 1) where a direct method is used in parallel with the method proposed in this report.

```
init r1, .., r20;
sum = 0;
outer loop {
  loop1 n times {
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r20
  }
  loop2 m times {
    nop
    nop
    ...
    nop  // 20 nop's
  }
}
check sum;
if sum!=CORRECT_SUM
  print r1, .., r20;
```

Figure 3.9 Pseudocode of a test program for static and dynamic behavior of register file.

Breaking the cross section is not always possible in software. Consider the ALU tests in Sec. 3.1. Let us assume a pipelined processor architecture. In such architecture, there are latches between pipeline stages (Fig. 3.8). In particular, there are two latches at the input of the ALU to hold the operand values. An error in one of these latches can have the same effect as an error in the combinational circuit of the ALU (the output latch can also be considered if the output is not bypassed). In fact, the cross section σ_{ALU}, used in the equations of the ALU test in Sec. 3.1, lumps the ALU cross section and the cross section of the relevant latches. A more precise equation that separates the two cross sections is:

$$\sigma_T = \sigma_{reg} f_{reg} + \sigma_{ALU} f_{ALU} + \sigma_{latches} f_{latches} + \sigma_{remainder} f_{remainder} \cdot$$

We would like to separate these two cross sections to see how much each contributes to the overall cross section. The result can determine if there is a need to add error detection for each part, e.g., parity for the registers and residue code ([Avizienis 71]) for the ALU.

To estimate the ALU cross section, we have to go through the latches to get values in and out of the ALU (in some architectures it may be possible to test the latches separately, e.g., see [Koga 85]). Consequently, the duty factors of ALU and latches (f_{ALU} and $f_{latches}$) are linearly dependent, that is, they increase or decrease by the same amount. Due to this linear dependency, the system of linear equations will not yield individual values for σ_{ALU} and $\sigma_{latches}$, just a total of the two. In general, other techniques have to be used when this type of linear dependency exists between functional units in a circuit. In the next section, we propose one such technique that may be used for the above example. Using scan chains may be another possible approach.

4. SEUs IN SEQUENTIAL AND COMBINATIONAL CIRCUITS

Most of the previous studies of SEUs in digital circuits focused mainly on sequential (storage) elements because the cross section of combinational circuits were considered insignificant compared to sequential circuits. In a study by Liden [Liden 94], combinational circuits contributed only 2% to the total number of errors. In contrast, the results in [Koga 85] show that the errors in arithmetic logic unit (ALU) constitute a statistically significant number of errors in one of the studied processors.

In this section, we first look at how SEU manifestations differ in sequential and combinational logic circuits and why SEUs in combinational logic circuits are becoming important. Then we describe how the differences can be used to separate the cross sections of these circuits.

A sequential circuit is composed of memory elements and combinational logic. In this report, we use the term “*sequential logic*” to refer to only the memory elements in the circuit, such as flip-flops, latches, registers and embedded RAM, and not the combinational logic parts. SEUs in sequential logic manifest themselves as bit-flips (also called soft errors). A *bit-flip* is an undesired change in the state of a storage element –

from 0 to 1 or from 1 to 0. Some previous studies show that for transients (glitches in voltage caused by an SEU) that are much shorter than a clock cycle, the cross section of sequential logic is independent of clock frequency and remains constant [Buchner 97][Shoga 94]. Other studies have found different behaviors, from nonlinear ([Reed 96]) to linear ([Gardic 96][Reed 96]) dependencies.

SEUs in combinational circuits cause transients at their outputs. The energy deposited in a node by an ionizing particle can be modeled as a voltage (or current) pulse. The duration of this pulse depends on the particle LET and the affected circuit, and can vary from hundreds of picoseconds for Si transistors [Wagner 88] to microseconds for certain GaAs transistors [Buchner 91]. If this pulse appears at the input of a following flip-flop (or latch) close to the latching edge of the clock, the transient may cause an error in system behavior. This window of vulnerability (Fig. 4.1) is referred to as the *latching window*[5] ([Cha 93]) and is the sum of the flip-flop (or latch) setup and hold times. If the transient pulse arrives at the flip-flop outside this window, the pulse will have no effect and will not cause any error. If the clock frequency is increased, the outputs of combinational circuits are sampled at a higher rate. Therefore, the probability of latching the transient pulse increases. In fact, many studies have shown a linear dependency between cross section of combinational circuits and clock frequency [Buchner 96][Buchner 97][Liden 94][Reed 96][Schneiderwind 92].

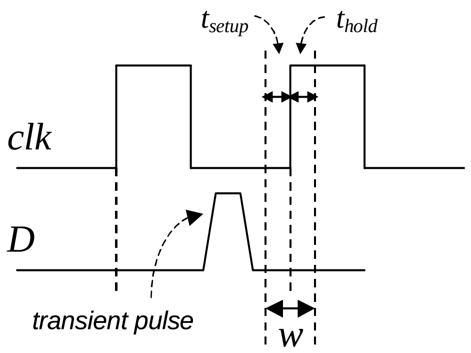

Figure 4.1 Latching window (*w*) of a transient pulse on the data input of a *D* flip-flop.

Many things can prevent an SEU from causing an error in a combinational logic circuit. In other words, a transient pulse generated at the input or inside a combinational

[5] Also known as the *metastability window*, or the *decision window* [Wakerly 00].

circuit may or may not appear on its output. Clearly, a pulse that is below the noise margin of the circuit has no effect (the particle energy is less than LET_{th}). Also, depending on input values, the output of a gate may or may not be sensitized to the input on which the transient pulse appears[6]. For example, a pulse on the upper input of AND gate *A* in Fig. 4.2 will be *logically masked* if there is a 0 on its lower input. If a pulse with a height greater than the noise margin appears at a sensitized input of a logic gate, it will propagate to the gate output. If the pulse width is greater than the logic transition time of a gate, the pulse passes through the gate with no change. However, if its width is shorter than the logic transition time its height will decrease. Consequently, such a transient pulse attenuates as it travels through the logic levels of a combinational circuit [Baze 97][Cha 93][Friedman 85]. This is shown in Fig. 4.2. Eventually, the pulse may be *electrically masked* ([Liden 94]) if it attenuates to a low level that can cause no error. Due to this effect, transients that are closer (in terms of logic levels) to the primary outputs of a combinational circuit are more probable to cause an error than those that are farther from the outputs.

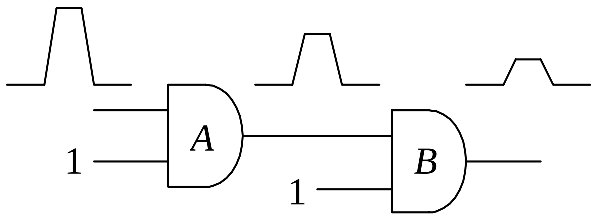

Figure 4.2 Attenuation of a transient pulse as it travels through a combinational circuit**.**

The trend in digital circuits is towards higher clock frequencies, lower noise margins (resulting in smaller LET_{th}) and fewer logic levels in each clock cycle. This means that combinational circuits may have a large contribution on the total error rate in future digital circuits. Therefore, it is important to measure the cross section of combinational circuits. Researchers have recently addressed the importance of combinational circuits in SEU behavior of digital circuits and have proposed circuit designs to reduce the effects [Nicolaidis 99].

[6] This is considered in test programs of Fig. 3.1 by initializing the registers with different patterns such that the ALU is exercised with many different input values. The initial values can also be changed between different test runs.

Let us address the problem of separating the cross section of an ALU (combinational logic) and pipeline latches (sequential logic) described at the end of Sec. 3.7. As explained in paragraph 5 if this section, the cross section of combinational circuits (σ_{comb}) increases linearly with clock frequency. Cross sections of sequential circuits (σ_{seq}) have shown different behaviors with clock frequency. If we know how σ_{seq} changes for the sequential elements used in the circuit under test, we can use this property to break the linear dependencies in our equations by running the test programs at different clock frequencies (assuming that the circuit can work properly at those clock frequencies). For example, let us assume that σ_{seq} does not change with clock frequency. We can operate the processor at its nominal frequency (ω_1) and half its nominal frequency ($\omega_2 = ½\,\omega_1$). If we know that $\sigma_{ALU@\omega 2} = ½\ \sigma_{ALU@\omega 1}$ and $\sigma_{latches@\omega 2} = \sigma_{latches@\omega 1}$, the resulting equations will be:

$$\sigma_{T@\omega 1} = \sigma_{reg@\omega 1} f_{reg} + \sigma_{ALU@\omega 1} f_{ALU} + \sigma_{latches@\omega 1} f_{latches} + \sigma_{remainder@\omega 1} f_{remainder}$$
$$\sigma_{T@\omega 2} = \sigma_{reg@\omega 1} f_{reg} + 1/2\, \sigma_{ALU@\omega 1} f_{ALU} + \sigma_{latches@\omega 1} f_{latches} + \sigma_{remainder@\omega 2} f_{remainder}$$

which can be solved to get separate σ_{ALU} and $\sigma_{latches}$. Note that $\sigma_{remainder}$ may include some combinational circuit and change with frequency. Therefore, we use $\sigma_{remainder@\omega 1}$ and $\sigma_{remainder@\omega 2}$ in the equations and estimate them separately.

In summary, clock frequency dependency can be used just like duty factors for creating linearly independent equations. Note that this method will not work if σ_{comb} and σ_{seq} have exactly the same frequency dependency.

5. MEASUREMENT ERRORS

As explained in Sec. 3.1, we write a system of linear equations based on the measured total cross sections and the duty factors designed in the weighted test programs. There are several sources of error in these equations that lead to inaccuracy in the solutions (the estimated σ_i's). We look at these errors in this section.

One source of error in the equations is the uncertainty in the duty factors associated with each functional unit. We saw this in the example of Sec. 3.1 where we tried to match the duty factors in two test programs.

The second source of error in the equations is the fact that error detection coverage in test programs is not 100 percent, which means, some errors are not detected during test. For example, an error in instruction fetch, decode, or control logic can cause an erroneous jump within *loop1* of the test programs in Fig. 3.1. This error results in an incorrect sum and therefore, is detected. However, an erroneous jump within *loop2* of this program is not detected (skipping one or more add instructions in this loop will not change the sum). For this test program, the errors that are cause by SEUs in instruction fetch unit, decode unit, or control logic are lumped in $\sigma_{remainder}$. The undetected errors will cause uncertainty in the measured σ_T and consequently a low estimation for $\sigma_{remainder}$ and a high estimation for σ_{ALU}. For the example of Fig. 3.1, some calculation needs to be done in *loop2* such that if instructions are skipped, the error is detected. This may be done by using an instruction that takes the same number of cycles as addition but does not use the ALU. One such instruction may be the shift operation. Figure 5.1 shows this enhancement for the test program of Fig. 3.1(a). In the test program of Fig. 5.1, the shift unit is also being tested and we have:

$$\sigma_T = \sigma_{reg} f_{reg} + \sigma_{ALU} f_{ALU} + \sigma_{Shifter} f_{Shifter} + \sigma_{remainder} f_{remainder} .$$

Since the register values are shifted in loop2, the most significant bits and least significant bits of the shifted registers are not live at all times (some of the SEUs that occur in these bits will have no effect on program output). This change must be reflected by adjusting f_{reg}. Another option is to use the rotate operation which does not affect the liveness of the registers[7]. The caveat in using the rotate operation is that the register values should be selected such that the rotated values are not the same as the original values otherwise some errors, for example, the control flow errors discussed in the third paragraph of this section, will not be detected.

[7] Not available in the MIPS instruction set architecture.

```
init r1, .., r10;
sum = 0;
outer loop {
  loop1 n times {
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r10
    add  sum,sum,r1
    add  sum,sum,r2
    ...
    add  sum,sum,r10
  }
  loop2 m times {
    sll  r1, r1, 1  // sll = shift left logical
    sll  r2, r2, 1
    ...
    sll  r10,r10,1
    srl  r1, r1, 1  // srl = shift right logical
    srl  r2, r2, 1
    ...
    srl  r10,r10,1
  }
}
check sum;
```

Figure 5.1 Enhanced version of the test program of Fig. 3.1(a).

Imperfect error detection coverage introduces some error in the measured overall cross section for each program (the σ_T 's). In the above example, the probability of an erroneous jump within the loop is very small. Therefore, in this case, these undetectable errors have low probability of occurence. In other cases, where undetected errors may cause more measurement errors, enhancing the error detection coverage is very important.

Unforeseen interrupts during the execution of test programs are another source of error in the equations. For example, if the computer system under test has a multitasking operating system, there will be task switches during radiation tests. A task switch takes several clock cycles to save and restore processor state to memory. This not only introduces errors in the duty factors but also provides opportunity for errors that might not have been considered in the design of the test program. Therefore, it may be better to disable task switching during certain radiation tests[8].

SEUs in units external to the chip under test may also skew the estimates. When radiation is not localized and the whole system is vulnerable to SEUs, we need to

[8] In some tests, we may want to consider task-switching effects or we may want to test a unit that facilitates task switching. Clearly, task switching is enabled in these cases.

consider all the discrete logic on the boards. For example, SEUs can occur in data and address buffers that connect the processor and main memory. It is difficult to distinguish the errors in these buffers from errors inside the processor. In the examples in this report, they are considered in $\sigma_{remainder}$. If these buffers are designed with parity or ECC, their errors are easily separated. Furthermore, if the parity or ECC protection can be disabled, test programs can be executed in disabled and enabled configurations to verify the estimated cross section for the buffers.

In summary, the key points in using the weighted test programs method are the following. First, the duty factors need to be controlled independently and as accurately as possible (Sec. 3.3-3.6). Second, the error detection coverage should be as high as possible. This may include detecting errors that do not affect the final program outputs (as explained in paragraph 3 of this section). Third, all the external effects should be considered and either isolated or counted in the experiments. Fourth, in the weighted test programs method there is no need to classify the errors and associate them with different functional units when there is ambiguity. The separation will be done by solving the system of linear equations. However, if there is only one possible source for an error, we can clearly use this fact for estimating the cross section of the corresponding functional unit.

The enumerated measurement errors cause inaccuracy in the estimated cross sections. We cannot eliminate all the measurement errors. However, the estimation errors can be reduced by having more equations than the number of unknowns in the linear equations. The statistical analysis of the derived linear equations is discussed in the Appendix. This method was used in the fault injection simulations that are explained in the next section.

6. FAULT INJECTION SIMULATIONS

To test the proposed technique, we conducted a series of fault injection simulations using a Verilog model of a simple pipelined MIPS processor (the MIPS-Lite used in the Stanford computer architecture course). Three sites were chosen for fault injection: (1)

the output of the ALU (before the bypass path forks off as shown in Fig. 6.1), (2) the register file, and (3) the output of the decoder that designates the destination register of the instructions. In each fault injection simulation, a single bit error is injected. If the fault site is the ALU, one random bit of the 32-bit output is inverted for one full clock cycle. For the register file, one of the 32 registers is chosen at random and a random bit in that register is flipped (this will persist until the register is overwritten or the simulation is reset). For the decoder, one random bit of the 4-bit output that designates the destination register is inverted for a full clock cycle.

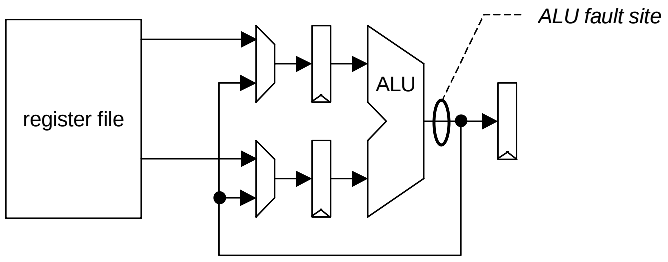

Figure 6.1 Block diagram showing the fault injection site for the ALU.

The programs in Fig. 3.1 were used as the test programs[9]. The C programs were compiled on a MIPS system with optimization level 2. We also counted the registers that were used for loop counter, etc. in the assembly code and ended with one test program with 14 registers and one with 20 registers. The outer loop was set to 10 iterations[10]. Three sets of values were used for the two inner loop counters: (*loop1*, *loop2*) = [(30, 0), (20, 10), (10, 20)], which gives three duty factors for ALU: f_{ALU} = 1, 2/3 and 1/3. The 2 values for f_{reg} (14/32 and 20/32) and 3 values for f_{ALU} means 6 different programs (assuming $f_{remainder}$ is 1 and constant), which gives us 6 equations with 3 variables: σ_{reg}, σ_{ALU} and $\sigma_{remainder}$. Notice that these tests are written for estimating σ_{reg} and σ_{ALU}, not $\sigma_{remainder}$.

Each test program takes about 6000 cycles to execute. In each fault injection simulation, a random cycle was selected as the time that the transient fault occurs (single

[9] Instead of 10 and 20 registers, the test programs were changed to use 9 and 18 registers so that there is no register spilling to memory for the test with 20 registers.
[10] This low number was chosen to keep the simulation time reasonable. In actual experiment, it should be in the orders of 10^6.

fault assumption) and one of the three fault injection sites was selected randomly based on a chosen distribution. This fault distribution simulates possible numbers for σ_{reg}, σ_{ALU} and $\sigma_{remainder}$ that could happen in a real chip. Each test program was simulated 10,000 times (a total of 10,000 faults for each program).

The result of calculations (summations) is stored at a fixed address in memory. When the test program finishes, the content of this address is recorded in the simulation output file. Since an injected fault may result in an infinite loop or hang up of the processor, a watchdog timer was simulated by limiting the number of execution cycles. We stopped the simulation if the cycle count exceeded 3 times the correct number of cycles. As the last instruction in the test programs, a flag is set to indicate if the program was executed to completion. The value of this flag is also recorded in the simulation output file. If this flag indicates completion and the calculation result matches the expected value, then the fault did not cause an error. Otherwise, the fault is counted as an error in the overall cross section measured for that test program.

Table 6.1 shows the simulation results for two different fault distributions. The numbers of faults (second column) represent the simulated cross sections. The numbers in bold type are the total number of errors counted for each test program and are the only numbers that are available in an actual radiation experiment. The rest of the numbers in columns 3 to 8 are known because in simulations we know where the fault is injected and if it caused an error. These numbers are not known in an actual experiment but are given here to make some observations.

Table 6.1 Fault injection simulation results.

Fault Distribution	Num of Faults	Errors counted for test program with f_{reg}, f_{ALU}					
		14/32, 1	14/32, 2/3	14/32, 1/3	20/32, 1	20/32, 2/3	20/32, 1/3
30% Reg. File	3,113	1,385	1,362	1,354	1,987	1,925	1,909
50% ALU	4,846	4,486	3,096	1,613	4,515	3,102	1,659
20% Remainder	2,041	1,917	1,802	1,656	1,961	1,823	1,668
Total	10,000	**7,788**	**6,260**	**4,623**	**8,463**	**6,850**	**5,236**
90% Reg. file	8,980	3,806	3,729	3,727	5,546	5,380	5,334
5% ALU	509	468	325	168	473	324	175
5% Remainder	511	477	440	407	489	451	409
Total	10,000	**4,751**	**4,494**	**4,302**	**6,508**	**6,155**	**5,918**

A few observations can be made from the numbers in Table 6.1. First, the number of errors caused by injecting faults in the decoder output is not the same for the different test programs for a particular fault distribution. Whether a fault on this output causes an error depends on the duty factors of the ALU and the register file. In other words, the activity in the other units (ALU and register file) affects the duty factor of the decoder, which is contrary to the assumption that $f_{remainder}$ is constant. This is an example that shows the difficulty in controling the duty factors and keeping them separate. In this case the effect is small but can be noticed as a source of error in the estimations. In general, it is very hard to find the exact duty factors. Simulation tools such as the one presented in [Asenek 98] can be used for estimating the duty factors.

Second, the number of errors caused by injecting faults in the register file (ALU) is not the same for test programs that have the same f_{reg} but different f_{ALU} (or the same f_{ALU} but different f_{reg}). The numbers show a small interdependency that introduces error in the estimations.

Using the numbers in Table 6.1, the linear equations from the first fault distribution are:

$$\begin{aligned}
7788 &= 14/23\,\sigma_{reg} + \sigma_{ALU} + \sigma_{remainder}\\
6260 &= 14/23\,\sigma_{reg} + 2/3\,\sigma_{ALU} + \sigma_{remainder}\\
4623 &= 14/23\,\sigma_{reg} + 1/3\,\sigma_{ALU} + \sigma_{remainder}\\
8463 &= 20/23\,\sigma_{reg} + \sigma_{ALU} + \sigma_{remainder}\\
6850 &= 20/23\,\sigma_{reg} + 2/3\,\sigma_{ALU} + \sigma_{remainder}\\
5236 &= 20/23\,\sigma_{reg} + 1/3\,\sigma_{ALU} + \sigma_{remainder}\,.
\end{aligned}$$

Solving these equations using the multiple linear regression method explained in Appendix will yield: σ_{reg} = 3,339, σ_{ALU} = 4,794 and $\sigma_{remainder}$ = 1,567. We can also calculate the confidence intervals for each case. Table 6.2 summaries the results of multiple linear regression calculations. Column 2 and 3 are the simulated and estimated cross sections, respectively. The last column is the 95% confidence interval for the estimations. The numbers show that, despite all the sources of error in estimations that were enumerated in Sec. 5 and observed in this section, the targeted cross sections,

σ_{reg} and σ_{ALU}, can be estimated by the proposed method within ±5% of the actual value. However, the estimate for σ_{ALU} in the second fault distribution shows that the method may not very accurate when a cross section is a very small portion of the overall cross section. This inaccuracy is partially due to the inaccuracy in $f_{remainder}$ as explained three pargraphs before. In an actual radiation test of a chip, the remainder is a mix of different units and $f_{remainder}$ may be more or less sensitive to f_{reg} and f_{ALU}. The accuracy of estimations are enhanced as the remainder portion of the system is broken into smaller units and more test programs are written to estimate their cross sections. Our method may not yield accurate estimates for small contributors to the overall cross section but it can determine the major contributors, which is the primary objective of the method.

Table 6.2 Simulated cross sections and the corresponding estimations from multiple linear regression analysis.

Fault Distribution	Functional Unit	Cross Sections (σ)		95% Confidence Interval
		Simulated	Estimated	
(30%, 50%, 20%)	Reg. File	3,113	3,339	[2,905 3,773]
	ALU	4,846	4,794	[4,645 4,944]
	Remainder	2,041	1,567	[1,313 1,821]
(90%, 5%, 5%)	Reg. File	8,980	8,949	[8,238 9,661]
	ALU	509	779	[534 1,024]
	Remainder	511	81	[-337 498]

7. SUMMARY

SEU characterization of digital circuits is important in designing and predicting the behavior of fault tolerant systems that are exposed to radiation. Hardware methods for this characterization may be expensive and intrusive in some applications. Direct software methods are good for some units such as the register file. However, as explained before, cross section of registers is not sufficient for estimating the cross section of the whole chip. Furthermore, the error manifestation of SEUs is different in combination and sequential logic circuits and SEUs in combinational logic are gaining importance in future circuits (Sec. 4). Therefore, methods for estimating the SEU cross section of individual units of a chip are beneficial to building fault tolerant systems.

We present a new method for SEU characterization of digital circuits and estimating the cross section of its constituent units during a radiation test. We use the fact that cross section of a chip changes under different conditions. One such condition is the utilization (duty factor) of the units. The other condition is the clock frequency. These two parameters are used to set up a system of linear equations based on a set of weighted test programs. Then, linear regression is used to solve the equations and obtain estimates for the cross section of each unit. The confidence interval of the estimated cross sections is shrunk by having more equations than unknowns.

8. CONCLUSIONS

The proposed method alleviates some of the limits of previous methods. Cross section granularity may be the primary limit of our technique. We used two parameters that affect cross sections: duty factor and clock frequency. If other parameters are found to affect cross sections, they can be used in a similar way to create linearly independent equations and achieve smaller granularity for cross sections.

Simulation results show that our method can determine the major contributors to the overall cross section and can give good estimation for cross sections of the corresponding units. However, due to measurement errors that are enumerated in Sec. 5, our method may not yield accurate estimates for small contributors of the overall cross section.

One advantage of our method is that, unlike previous methods, it does not need any error location or error classification. This is because, for each test program, all the errors are collected and considered for the total cross section in the corresponding equation. The errors are separated indirectly (as opposed to direct tracing of the error to its source) when the individual cross sections are derived by solving the equations.

Our method can be combined with previous methods such as the scan chain approach. It may also be extended for SEU characterization of analog circuits.

ACKNOWLEGMENTS

This work was supported in part by the National Aeronautics and Space Administration (NASA) and administered through the Jet Propulsion Laboratory (JPL), California Institute of Technology under Contract No. 1216777. The authors wish to thank Scott Davidson, Nirmal Saxena, Subhasish Mitra, Nahmsuk Oh and Nur Touba for their reviews and valuable comments.

REFERENCES

[Abramovici 90] Abramovici, M. et al., *Digital Systems Testing And Testable Design*, IEEE Press, 1990.

[Asenek 98] Asenek, V., et al., "SEU Induced Errors Observed in Microprocessor Systems," *IEEE Trans. on Nuclear Science*, Vol. 45, No. 6, pp. 2876-83, Dec. 1998.

[Avizienis 71] Avizienis, A., "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Trans. Comp.*, C-20, pp. 1322-1331, Nov. 1971.

[Baze 97] Baze, M.P., and S.P. Buchner, "Attenuation of Single Event Induced Pulses in CMOS Combinational Logic," *IEEE Trans. on Nuclear Science*, Vol. 44, No. 6, pp. 2217-23, Dec. 1997.

[Brown 97] Brown, G.R., "Honeywell Radiation Hardened 32-bit Processor Central Processing Unit, Floating Point Processor, and Cache Memory Dose Rate and Single Event Effects Test Results," *IEEE Radiation Effects Data Workshop*, pp. 110-5, 1997.

[Buchner 91] Buchner, S., et al., "Charge Collection in GaAs MESFETs and MOSFETs," *IEEE Trans. on Nuclear Science*, Vol. 38, No. 6, pp. 1370-6, Dec. 1991.

[Buchner 96] Buchner, S., et al., "Modification of Single Event Upset Cross Section of an SRAM at High Frequencies," *Radiation Effects on Components and Systems Conf.*, pp. 326-32, 1996.

[Buchner 97] Buchner, S., et al., "Comparison of Error Rates in Combinational and Sequential Logic," *IEEE Trans. on Nuclear Science*, Vol. 44, No. 6, pp. 2209-16, Dec. 1997.

[Cha 93] Cha, H., et al., "A Fast and Accurate Gate-Level Transient Fault Simulations Environment," *IEEE Intr'l Symp. on Fault-Tolerant Computing (FTCS-23)*, pp. 310-9, June 1993.

[Cusick 85] Cusick, J., et al., "SEU Vulnerability of the Zilog Z-80 and NSC-800 Microprocessors," *IEEE Trans. on Nuclear Science*, Vol. NS-32, No. 6, pp. 4206-11, Dec. 1985.

[Dai 99] Dai, C., et al., "Alpha-SER Modeling & Simulation for Sub-0.25μm CMOS Technology," *Proc. Symp. On VLSI Tech.*, pp. 81-2, 1999.

[Elder 88] Elder, J.H., et al., "A Method for Characterizing a Microprocessor's Vulnerability to SEU," *IEEE Trans. on Nuclear Science*, Vol. 35, No. 6, pp. 1678-81, Dec. 1988.

[Friedman 85] Friedman, A.L., et al., "Single Event Upsets in Combinatorial and Sequential Current Mode Logic," *IEEE Trans. on Nuclear Science*, Vol. NS-32, No. 6, pp. 4216-18, Dec. 1985.

[Gardic 96] Gardic, F., et al., "Dynamic Single Event Effects in a CMOS/Thick SOI Shift Register," *Radiation Effects on Components and Systems Conf.*, pp. 333-9, 1996.

[Hareland 00] Hareland, S., et al., "Intel Perspectives on SER," *Topical Research Conf. On Reliability*, pp. 28, Oct. 30 – Nov. 1, 2000.

[Hiemstra 99] Hiemstra, D.M., and A. Baril, "Single Event Upset Characterization of the Pentium MMX and Pentium II Microprocessors Using Proton Irradiation," *IEEE Trans. on Nuclear Science*, Vol. 46, No. 6, pp. 1453-60, Dec. 1999.

[Hines 90] Hines, W.H., and D.C. Montgomery, *Probability and Statistics in Engineering and Management Science*, 3rd ed., (Chapter 15), John Wiley & Sons, Inc., 1990.

[Kimbrough 94] Kimbrough, J.R., et al., "Single Event Effects and Performance Prediction for Space Applications of RISC Processors," *IEEE Trans. on Nuclear Science*, Vol. 41, No. 6, pp. 2706-14, Dec. 1994.

[Koga 85] Koga R., et al., "Techniques of Microprocessor Testing and SEU-Rate Prediction," *IEEE Trans. on Nuclear Science*, Vol. NS-32, No. 6, pp. 4219-24, Dec. 1985.

[Kouba 97] Kouba, C.K., and Gwan Choi, "The Single Event Upset Characterization of the 486-DX4 Microprocessor," *IEEE Radiation Effects Data Workshop*, pp. 48-52, 1997.

[Liden 94] Liden, P., et al., "On Latching Probability of Particle Induced Transients in Combinational Networks," *IEEE Intr'l Symp. on Fault-Tolerant Computing (FTCS-24)*, pp. 340-9, June 1994.

[McCluskey 86] McCluskey, E.J., *Logic Design Principles*, Prentice-Hall, Inc., 1986.

[Melinger 94] Melinger, J.S., et al., "Critical Evaluation of the Pulsed Laser Method for Single Event Effects Testing and Fundamental Studies," *IEEE Trans. on Nuclear Science*, Vol. 41, No. 6, pp. 2574-84, Dec. 1994.

[Moran 96] Moran, A., "Single Event Effect Testing of the Intel 80386 Family and the 80486 Microprocessor," *IEEE Trans. on Nuclear Science*, Vol. 43, No. 3, pt. 1, pp. 879-85, June 1996.

[Motorola 97] *MPC750 RISC Microprocessor Technical Summary*, http://www.motorola.com, 1997.

[Nicolaidis 99] Nicolaidis, M., "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies," *IEEE VLSI Test Symp.*, pp. 86-94, April 1999.

[Reed 96] Reed, R.A., et al., "Single Event Upset Cross Sections at Various Data Rates," *IEEE Trans. on Nuclear Science*, Vol. 43, No. 6, pp. 2862-7, Dec. 1996.

[Schneiderwind 92] Schneiderwind, R., et al., "Laser Confirmation of SEU Experiments in GaAs MESFET Combinational Logic," *IEEE Trans. on Nuclear Science*, Vol. 39, No. 6, pp. 1665-70, Dec. 1992.

[Seifert 01] Seifert, N., et al., "Historical Trend in Alpha-Particle Induced Soft Error Rates of the Alpha Microprocessor," *IEEE Intr'l Reliability Physics Symp.*, April 30 – May 3, pp. 259-265, 2001.

[Sexton 90] Sexton, F.W., et al., "SEU Characterization and Design Dependence of the SA3300 Microprocessor," *IEEE Trans. on Nuclear Science*, Vol. 37, No. 6, pp. 1861-8, Dec. 1990.

[Shaeffer 92] Shaeffer, D.L., et al., "Proton-Induced SEU, Dose Effects, and LEO Performance Predictions for R3000 Microprocessors," *IEEE Trans. on Nuclear Science*, Vol. 39, No. 6, pp. 2309-15, Dec. 1992.

[Shoga 94] Shoga, M., et al., “Single Event Upset at Gigahertz Frequencies,” *IEEE Trans. on Nuclear Science*, Vol. 41, No. 6, pp. 2252-8, Dec. 1994.

[Thomlinson 87] Thomlinson, J., et al., “The SEU and Total Dose Response of the INMOS Transputer,” *IEEE Trans. on Nuclear Science*, Vol. 34, No. 6, pp. 1803-7, Dec. 1987.

[Velazco 92] Velazco, R., et al., “SEU Testing of 32-Bit Microprocessors,” *IEEE Radiation Effects Workshop*, pp. 16-20, 1992.

[Wagner 88] Wagner, et al., “Alpha-, Boron-, Silicon-, and Iron-Ion-Induced Current Transients in Low-Capacitance Silicon and GaAs Diodes,” *IEEE Trans. on Nuclear Science,* Vol. 35, No. 6, pp. 1578-84, Dec. 1988.

[Wakerly 00] Wakerly, J.F., *Digital Design Principles & Practices*, 3rd ed., Prentice Hall, 2000.

[Ziegler 96] Ziegler, J.F., et al., *IBM J. Res. Develop.*, Vol. 40, No. 1, (all articles), Jan. 1996.

APPENDIX

STATISTICAL ANALYSIS

The general form of a linear equation derived from running a test program in a radiation test can be written as:

$$y = a_0 + a_1 x_1 + a_2 x_2 + ... + a_k x_k + \varepsilon \text{ (2)},$$

where *y* is the measured cross section, a_i's are the cross sections with a_0 being the *baseline* cross section that is independent of the program being executed on the processor, and x_i's are the duty factors. ε represents all the measurement errors and is, of course, unknown. By writing *n* different programs and doing *n* different measurements, we get *n* equations similar to Eq. (2). A multiple regression model can be used for solving these equations (for a detailed explanation of the formulas in this section please see [Hines 90]). The equations may be written in a combined matrix format:

$$Y_{n\times 1} = X_{n\times p} A_{p\times 1} + \varepsilon_{n\times 1}.$$

Matrix *A* contains the *p* (*p=k+1*) unknown cross sections. Using the method of least squares, this matrix can be estimated by the following formula:

$$\hat{A} = (X^T X)^{-1} X^T Y \text{ (minimizing } L = \sum_{i=1}^{n} \varepsilon_i^2 \text{).}$$

Once we have the estimates for the cross section, we need to know how good the estimates are by calculating their confidence interval. In general, if

$$Prob\{L_i \le a_i \le U_i\} = 1 - \alpha,$$

the interval $L_i \le a_i \le U_i$ is called a $100(1-\alpha)$ percent *confidence interval* for a_i. Each of the statistics

$$\frac{\hat{a}_i - a_i}{\sqrt{\hat{\sigma}^2 C_{ii}}} \qquad i = 0, 1, \ldots, k$$

is distributed as *t* with *n - p* degrees of freedom, where C_{ii} is the *ii*th element of the $(X^T X)^{-1}$ matrix, and $\hat{\sigma}^2$ is the estimate for the error variance. A $100(1-\alpha)$ confidence interval for the regression coeffient a_i, *i* = 0, 1, …, *k*, is:

$$\hat{a}_i - t_{\alpha/2,n-p}\sqrt{\hat{\sigma}^2 C_{ii}} \le a_i \le \hat{a}_i + t_{\alpha/2,n-p}\sqrt{\hat{\sigma}^2 C_{ii}}.$$

The $t_{\alpha/2,n-p}$ factors can be looked up from a percentage points table of a *t* distribution. A small part of this table is shown in Table A.1 for illustration. When the number of measurements is equal to the number of unknowns (*n* = *p*), there is zero degree of freedom and we cannot define a confidence interval. By making more measurements, we can use linear regression and define a confidence interval for our estimates. As shown in Table A.1, the $t_{\alpha/2,n-p}$ factors become smaller as the number of measurements increases (higher *n* - *p*). However, the change in $t_{\alpha/2,n-p}$'s is not significant for higher degrees of freedom. Therefore, after a certain number of measurements, more measurements will have small effect in shrinking the confidence interval. For example, for 95% confidence interval ($\alpha = .05$) and *n* – *p* = 5, $t_{\alpha/2,n-p}$ is equal to 2.571 and decreases to 2.447 *n* – *p* = 6. Therefore, *n* – *p* = 5 may be considered sufficient for our method.

Table A.1 Percentage points of the *t* distribution [Hines 90].

n - *p*	$\alpha/2$		
	.05	.025	.01
1	6.314	12.706	31.821
2	2.920	4.303	6.965
3	2.353	3.182	4.541
4	2.132	2.776	3.747
5	2.015	2.571	3.365
6	1.943	2.447	3.143
7	1.895	2.365	2.998

The C_{ii} parameters are the other factors that influence the confidence interval and are preferably minimized by choosing proper duty factors during the design of test programs. In general, it is better to have large differences between duty factors in different tests.