TU/e technische universiteit eindhoven

Deviation-Tolerant Computation in Concurrent Failure-Prone Hardware

Phillip Stanley-Marbell
Diana Marculescu

ES Reports
ISSN 1574-9517

ESR-2008-01
24 January 2008

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

People who are really serious about software should make their own hardware. — Alan Kay

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Deviation-Tolerant Computation in Concurrent Failure-Prone Hardware

PHILLIP STANLEY-MARBELL

Technische Universiteit Eindhoven, Den Dolech 2, 5612 AZ Eindhoven, NL.

DIANA MARCULESCU

Department of ECE, Carnegie Mellon, 5000 Forbes Ave, Pittsburgh PA 15213-3890, US.

In many applications of computing systems, particularly those which process data samples from real-world signals, it is possible to trade off accuracy of computation in the presence of hardware faults, for performance or energy efficiency. Such trade-offs may be even more pronounced in platforms which employ multiple processing elements, as one may then also consider trade-offs between the speeds of communication between processing elements (and hence the computation throughput), and the possibility of faults in such communications (and hence possible errors in a computation's result).

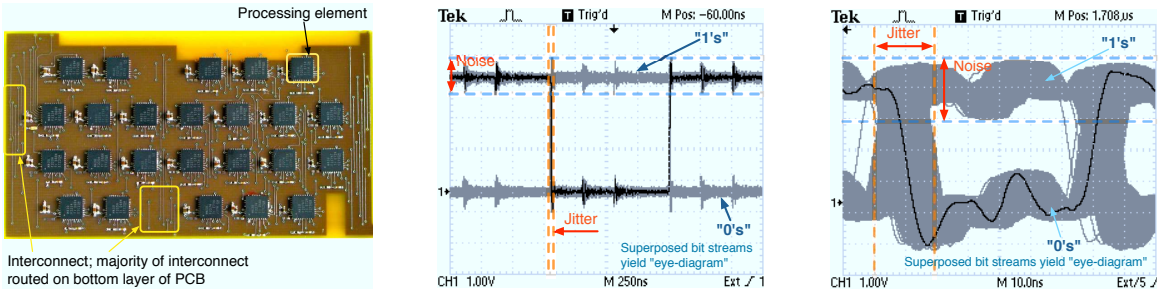
Presented are analysis on the relation between faults occurring in compute hardware or communicated program state (in a multiprocessor system) and the resulting *deviations in values* manifested in source-level program variables. These relations are dependent on the distributions of values taken on by program variables of different data types in the absence of faults, and we present detailed characterizations of these distributions for a large collection of programs. We show how the analytic derivations, in conjunction with the empirical characterizations, can enable the implementation of *deviation-tolerant* transformations in programs. The work is presented in the context of a hardware platform we have designed and implemented, containing 24 processing elements, that manifests tradeoffs between occurrences of faults in hardware, performance, and energy efficiency.

1. INTRODUCTION

Failures in hardware may be the result of a variety of phenomena, but are usually eventually manifested as undesirable program behaviors. These undesirable behaviors may take many forms, including deviation from correct flow control and deviations of values taken on by program variables. In some classes of applications, small magnitudes of such deviations of values may be acceptable. For example, small magnitudes of deviations in the variables representing pixel values in an image processing application may be tolerable. When such tolerance to deviations exists, it is desirable to quantify the relation between the occurrence of faults in hardware and the incurred deviations of variable values, in order to enable trade-offs between performance or energy efficiency, and such value deviations. Analyses capturing these relations, as well as the empirical program properties on which they depend, are the subject of this paper. These analyses enable a variety of new program-level transformations, such as compile-time transformations to trade off the distribution of deviation magnitudes, for computation and memory overheads, or for energy efficiency.

1.1 A motivating hardware platform

A concrete example of a hardware platform with such performance versus value deviation trade-offs is shown in Figure 1(a). The platform shown in the figure is employed as a computation resource for a low-power em-



(a) 24-processor system. (b) Eye diagram at 1.0 Mb/s (c) Eye diagram at 16.0 Mb/s

Fig. 1. A hardware platform containing 24 processing elements, used as a scalable embedded multicomputer system ((a), left). The oscilloscope-captured eye diagrams from hardware measurements performed on this platform, shown in (b) and (c), illustrate tradeoffs between speed and error probability (a function of noise) on the interconnect between processing elements.

bedded system. It contains 24 ultra-low-power microcontrollers, each running at 16 MHz, with 32 kB of internal flash memory and 1 kB of on-chip SRAM [Texas Instruments, Inc., 2006] interconnected in a low-diameter interconnect network, and provides *both* low idle power dissipation (less than 30 μW) and high peak computation throughput (scalable up to 384 MIPS at approximately 1 mW per MIPS).

Both the interconnect and processing elements exhibit performance/power versus reliability trade-offs. At high interconnect speeds (which can be configured by software), the likelihood of bit-errors in communicated data is increased, as the signal-to-noise ratio is decreased, and relative jitter increases (Figure 1(b) and (c)). High interconnect speeds however provide increased performance and reduce the energy per communicated bit. The system operates in a voltage range of 1.8 V to 3.6 V, and at a given voltage, the maximum frequency at which each of its constituent processing elements can be safely run is bounded within a window specified by the manufacturer of the component processors; operating at voltages that are close to the lower threshold of permissible voltage for a given frequency reduces dynamic power dissipation, but increases the likelihood of faults in computation and communication.

The hardware platform shown in Figure 1(a) is programmed using a programming model in which applications to be executed are partitioned over the collection of processors, with these individual partitions communicating over the interconnect to achieve the execution of a single application. Faults in the communication interconnect, due to either the chosen communication speed or operating voltage, manifest as bit errors in communicated data, and eventually as errors in the executing application. The communicated data corresponds to values of program variables and data structures, and when these variables or data structure elements are of arithmetic types (e.g., types `int` and `float` in the C programming language), we may consider the effects of communication faults as inducing *value deviations*. The precise nature of these value deviations are dependent on the distribution of bit-level faults incurred in the communication medium, on the types of the variables or data structures (e.g., `unsigned int` versus `signed int`), and on the values being communicated. Some variables, due to the nature of data they represent, may be tolerant of larger value deviations than others (e.g., variables representing color pixel values, versus pointers). It is thus possible, in combination with forward error correction for a restricted set of the data traversing the interconnect, to operate the system at a configuration that provides a desired trade-off between performance, energy dissipation, and correctness.

In this paper, we present the analytic relations for determining the distributions of value deviations, and empirical studies of the distributions of error-free values taken on by variables of different data types on which the value deviation relations depend, for a large collection of embedded and general purpose applications.

1.2 Other sources of temporary logic upsets in hardware and software

There are a variety of other physical processes leading to faults in semiconductor devices. For the purposes of this work, those of interest are temporary or intermittent failures, which cause temporary disturbances of circuit state; in digital systems, these disturbances of circuit state are manifested as disturbances of digital logic values, or *logic upsets*.

Temporary logic upsets have long been of concern in high availability systems such as servers [Horst et al., 1990; Slegel et al., 1999]. In PCs, workstations and server-class systems, the predominant causes of logic upsets are high energy particles such as α-particles [Baumann, 2005]. The α-particle flux, the number of particle strikes per m^2/s per second, varies with altitude (with a peak at approximately 60,000 feet), with time (varies with the 11 year solar cycle), with application domain (e.g., terrestrial versus space applications), and also varies with latitude [Heidergott, 2005]. Some of the natural sources of α-particles are illustrated in Figure 2. In embedded systems, which are often deployed in environments that differ drastically from climate-controlled office and server rooms which are often thought of as typical computing system deployments, additional sources of logic upsets include electrical noise and various sources of electromagnetic radiation.

The scaling of semiconductor process technologies requires the use of ever lower operating voltages (e.g., to maintain a constant electric field across generations in *constant field scaling*); these lower operating voltages reduce the noise margins of circuits, making logic upsets even more likely. Device scaling also reduces the minimum charge necessary to disturb circuit state, and as a result, it is easier for lower-energy disturbances to cause upsets. Even though shrinking device sizes reduces the probability of a given hardware structure incurring a high-energy particle hit (there is a smaller area to target), the increasing number of transistors being integrated into contemporary designs results in little decrease of the probability of logic upsets for the whole integrated circuit across technology generations [Cannon et al., 2004].

The effect of the aforementioned physical phenomena is often complex; for digital systems however, fluctuations in circuit state eventually manifest as changes in logic values — i.e., a binary digit in a hardware structure is forced to a logic `0` or a logic `1`. If the value forced at a bit is the same as the value already there, the logic upset is said to be *masked*. We will refer to bit upsets, if not masked by the underlying bit values, as *errors*. To contrast them from irreversible failures, temporary / intermittent failures are usually referred to as *soft errors*.

One measure of the rate of occurrence of logic upsets is the metric of *failures in time (FIT)*, with 1 FIT corresponding to one failure every 10^9 device operation hours. A current-generation 8 MB (64 Mb) static random-access memory (SRAM), has a FIT rate of approximately 100,000 at sea level, and may thus witness approximately one such upset a year.

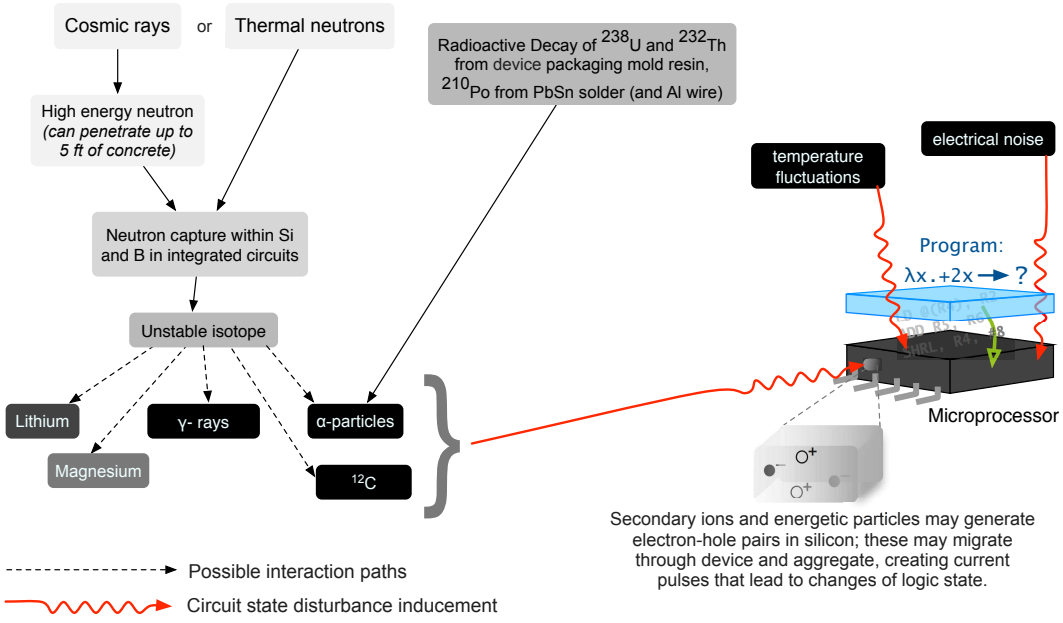


Fig. 2. Some sources of temporary logic upsets in hardware.

1.3 Contributions and paper outline

This paper presents the mathematical underpinnings and quantitative evaluations necessary to enable *deviation-tolerant computation and program-level transformations*. The work is presented in the context of a concrete hardware platform in which such transformations are of interest. Following an overview of relevant related research in Section 2, Section 3 introduces the terminology employed in the remainder of the paper. The derivation of the analytic expressions relating empirical program properties and hardware fault properties, to the resulting program-level value deviations are presented in Section 4. The empirical program properties on which these derived relations depend are presented in Section 5, followed by an example application of the ideas presented in the paper in Section 6. Section 7 concludes the paper with a summary and directions for future research.

2. RELATED RESEARCH

A recent attempt to formalize the effects of soft-errors on the behavior of programs is presented in [Walker et al., 2006]. The model addressed therein is one in which the goal is to *attempt to nullify the effect of soft errors* by redundant computation, as opposed to being tolerant of some distribution of value deviations. Other techniques have previously been presented in the research literature targeting corruption of code and control-flow deviations resulting from logic upsets [Saxena and McCluskey, 1990]. The techniques presented in this paper are complementary, as we do not directly address code corruption or control-flow disruption due to logic upsets, but rather address the potential for reducing the overhead of required redundancy (in time or space) when some deviation in values of computations resulting from logic upsets may be tolerable.

The observation that different portions of programs, or of hardware, may require different amounts of fault-protection, has previously been made for hardware systems [Mukherjee et al., 2003], phases of programs [Reis et al., 2005], and the computations performed by special classes of applica-

tions [Wong and Horowitz, 2006]. In contrast, the program-level deviation-tolerance analysis presented in this paper facilitates the application of varying amounts of fault protection to individual program variables, and we present concrete compile-time analysis for facilitating such per-variable transformation.

The analyses presented in this paper employ as input the distribution of error-free values taken on by program variables at runtime. While such distributions have not been studied in detail in the research literature, they are related to the ideas of spatially-frequent values [Yang and Gupta, 2002], minimum bit-widths [Brooks and Martonosi, 1999; Budiu et al., 2000; Mahlke et al., 2001; Stephenson et al., 2000], value profiling [Calder et al., 1997], and value locality [Lipasti et al., 1996].

3. TERMINOLOGY, DEFINITIONS, AND ASSUMPTIONS

When the values taken on by a phenomenon are not predictable, they may intuitively be considered in terms of their individual likelihoods of occurrence. The possible manifestations of such phenomena are referred to in probability theory as *events*. In the context of this paper, events will correspond to *source-level program variables* of a given type (e.g., `int` or `double`) taking on a specific value. Each event can be interpreted as a value taken on by a *random variable*. Associated with each event is a *probability*, a real number between 0 and 1, that indicates the likelihood of the event, with value 0 for the impossible event, and value 1 for the certain event. In the case of a discrete space of events, the function mapping events to their probabilities is called a *probability mass function (PMF)*. The PMF for a random variable X defines the probability that the random variable X (which might represent the values taken on by a *program source variable*,) takes on the specific value x, written as $\Pr\{X = x\}$ or $f_X(x)$.

Given two random variables A and B, the *joint probability mass function (joint PMF)*, is the probability that A takes on the value a at the same time as B takes on the value b, written as $\Pr\{A = a, B = b\}$, or $f_{A,B}(a,b)$. The distributions $f_A(a)$ and $f_B(b)$ are referred to as the *marginals* of the joint PMF $f_{A,B}(a,b)$. If the random variables A and B are *independent*, the joint PMF is identically the product of the marginals. The specific notation used in the remainder of the paper is summarized in Figure 3.

4. PER-TYPE DISTRIBUTIONS OF VALUE DEVIATIONS IN PROGRAMS

Deviations in values of variables due to logic upsets are determined by three factors — (1) the logic upsets occurring within machine words representing variables of a given type, (2) the nature of the bit-level layout of different data types, and (3) the values taken on by variables in the absence of errors.

The data type of a variable (e.g., `int` versus `float`) determines how individual bits affect the numeric value the variable holds. For example, in an unsigned 8-bit data type, the most significant bit (bit 7) has a greater contribution to its value than the least significant bit (bit 0). The values taken on by variables in the absence of upsets determine the likelihood that a logic upset at a given bit position will be masked. For example, if a variable always takes on the numeric value zero, and if logic upsets are always such that they force the affected bit position to a 0, then upsets will always be masked and the value deviation will always be zero. As a more concrete example, Figure 4 shows the empirical runtime value and bit-state probability distributions

$\Pr\{E\} \stackrel{\text{def}}{=}$ Probability of the event E

$f_X(x) \stackrel{\text{def}}{=} \Pr\{X = x\}, \qquad F_X(x) \stackrel{\text{def}}{=} \Pr\{X \leq x\}, \qquad \overline{F}_X(x) \stackrel{\text{def}}{=} \Pr\{X > x\}$

Parameters related to physical processes

$t \stackrel{\text{def}}{=}$ **transient logic upset vector**

$t_{\langle i \rangle} \stackrel{\text{def}}{=} i^{th}$ bit of transient logic upset vector

$f_{t_{\langle i \rangle}}(k) \stackrel{\text{def}}{=}$ Probability mass function (PMF) for $t_{\langle i \rangle}$

$=$ probability that position i of bit upset vector, t, takes on value $k \in \{0, 1\}$

Empirical program properties

$V \stackrel{\text{def}}{=}$ Random variable, **error-free variable value**; assumed independent of t

$V_{\langle i \rangle} \stackrel{\text{def}}{=}$ Random variable, i^{th} bit of variable value

$f_V(v) \stackrel{\text{def}}{=} \Pr\{V = v\}$, *e.g.,* $= \dfrac{1}{2^n}$ *for an* n*-bit* V *with all values equally likely*

$f_{V_{\langle i \rangle}}(k) \stackrel{\text{def}}{=} \Pr\{V_{\langle i \rangle} = k\}, \quad k \in \{0, 1\}$, *e.g.,* $= 0.5$ *when all values are equally likely*

Derived quantities

$W \stackrel{\text{def}}{=}$ Random variable, **error-containing variable value** —

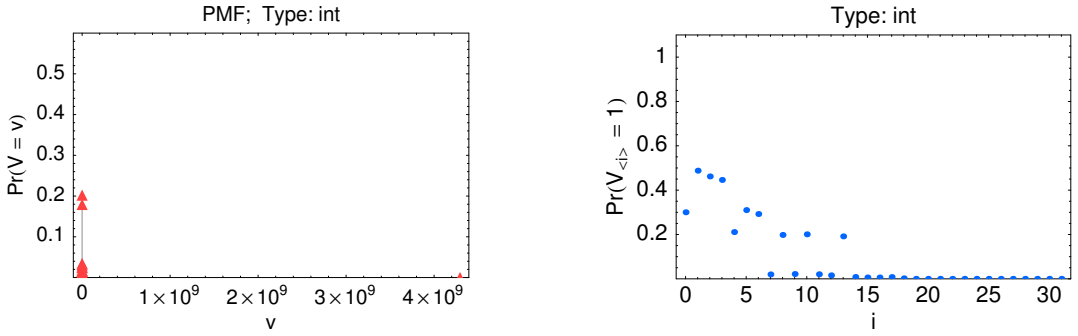
a value that is incorrect due to the incidence of logic upsets in hardware

$f_W(w) \stackrel{\text{def}}{=} \Pr\{W = w\}$

$M \stackrel{\text{def}}{=}$ Random variable, numeric **value deviation**

$f_M(m) \stackrel{\text{def}}{=} \Pr\{\mid W - V \mid = m\}$

Fig. 3. Summary of terminology and definitions.



(a) Empirical distribution of values.

(b) Empirical bit-level probabilities.

Fig. 4. Example of empirically measured distributions of variable values and bit-level probabilities, illustrating varying likelihood of logic upset masking across bits in variables of data type `int` for the SPEC2000 `ammp` benchmark.

aggregated across variables of type `int` in the `ammp` benchmark from the SPEC CPU 2000 benchmark suite[1]. The data in the figure was collected by monitoring the values of variables in the benchmark each time they

[1] We have observed other applications from different application domains to exhibit similar properties. The presentation of those detailed empirical characterizations of the value and bit-state distributions for different data types, across multiple applications is presented in Section 5.

were read or written at runtime, and logging the observed values, over the benchmark's lifetime. From the figure, it can be seen that the most significant 17 bits of variables of C language data type `int` in the `ammp` application, take on the value `1` with almost zero probability, and are thus very unlikely to mask logic upsets which force them to a `1`.

In what follows, we present analytic expressions for the relation between value deviation, probability of logic upsets, and the values taken on by variables, in the absence of faults. We proceed by first deriving expressions for the distribution of possibly-erroneous values taken on by variables, which we refer to as the *error-containing value*, W. This distribution of values for W, $f_W(w)$, will then be used in Section 4.4 to obtain closed-form analytic expressions for the distribution of value deviations M, $f_M(m)$.

4.1 Error-containing value PMF, $f_W(w)$, unsigned n-bit values, single logic upsets

For upset-free values, $f_V(v)$ defines the probability that an n-bit value, V, takes on the specific instance value of v. Similarly, the *error-containing value PMF*, $f_W(w)$ defines the probability that a value, W, which might have incurred single or multiple-bit upsets, defined by the upset distribution $f_{t_{\langle i \rangle}}(k)$, has the specific instance value w. For the case of a single logic upset in one of the n bit positions,

$$
\begin{aligned}
f_W(w) &= \sum_{i=0}^{n-1} \Big(\underbrace{\Pr\left\{t_{\langle i \rangle} = 0, V = w + 2^i\right\}}_{\text{❶}} + \underbrace{\Pr\left\{t_{\langle i \rangle} = 1, V = w - 2^i\right\}}_{\text{❷}} \Big) + \underbrace{\Pr\left\{ \bigcup_{0<i<n-1} t_{\langle i \rangle} = V_{\langle i \rangle}, V = w \right\}}_{\text{❸}} \\
&= \sum_{i=0}^{n-1} f_V(w + 2^i) f_{t_{\langle i \rangle}}(0) + \sum_{i=0}^{n-1} f_V(w - 2^i) f_{t_{\langle i \rangle}}(1) + \sum_{i=0}^{n-1} f_{t_{\langle i \rangle}}(V_{\langle i \rangle}) f_V(w), \qquad (1)
\end{aligned}
$$

where, for the term ❶, the error-containing value is smaller than upset-free value, due to bit i forced from a `1` to a `0`; for term ❷, the error-containing value is larger than upset-free value, due to bit i forced from a `0` to a `1`; for term ❸, the logic upset at bit i is masked by the pre-existing value in the variable. In the last step above, the joint PMFs are re-written as the product of the marginal PMFs due to the assumed independence between upset-free values and the occurrence of logic upsets. The overall structure of Equation 1 is governed by the digital arithmetic properties of unsigned integer values, wherein bit i of an n-bit word contributes 2^i to its numeric value.

4.2 Error-containing value PMF, $f_W(w)$, unsigned n-bit values, multiple independent logic upsets

The analytic expression for the error-containing value in Equation 1 was derived based on the assumption of the possibility of occurrence of only a single logic upset at a time. In practice however, it is possible that multiple logic upsets may affect a single machine word representing a program variable, and such multi-bit upsets may be independent or correlated.

In considering multiple independent upsets, it is easiest to begin by looking at the conditional PMF, $f_W(w \mid V = a)$, and once that has been determined, un-conditioning by summing over all possible values for a. For a particular instance $V = a$ of the upset-free value V, the corresponding value deviation (m) and

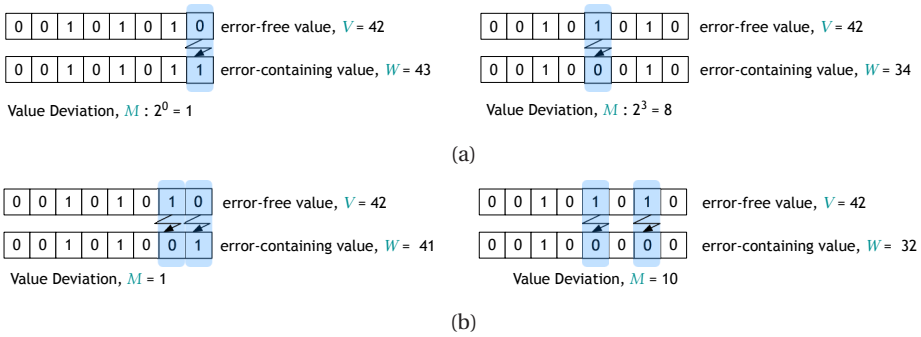


Fig. 5. Example and intuition behind the relation between *error-free values*, V, the distribution of (a) single and (b) multiple independent logic upsets, $f_{t_{\langle i \rangle}}(k)$, the resulting *error-containing values*, W, and the value deviation M, for unsigned data types.

error-containing values (w) are related by $m = \mid w - a \mid$. Then,

$$\begin{aligned} f_W(w \mid V = a) &= \frac{\Pr\{W = w, V = a\}}{f_V(a)} \\ &= \frac{\Pr\left\{V = a, \sum_{i=0}^{n-1} M_i = \mid w - a \mid\right\}}{f_V(a)}, \\ \text{where} \quad M_i &= \left(-\left(a_{\langle i \rangle} 2^i\right) + \left(t_{\langle i \rangle} 2^i\right)\right)\left(t_{\langle i \rangle} \oplus a_{\langle i \rangle}\right). \end{aligned} \tag{2}$$

The intuition behind the above expression for M_i, is to obtain an expression for the idea: "If bit i in a is flipped from 0 to 1, the contributed error is the addition of 2^i, and if flipped from 1 to 0, the contributed error is the subtraction of 2^i". When the logic upset leads to a bit value in the upset-vector, t, to be the same as that in the value, a, then $t_{\langle i \rangle} \oplus a_{\langle i \rangle} = 0$. Un-conditioning by summing over all possible values of a, we have

$$\begin{aligned} f_W(w) &= \sum_{a=0}^{2^n-1} f_W(w \mid V = a) \\ &= \sum_{a=0}^{2^n-1} \frac{\Pr\left\{V = a, \sum_{i=0}^{n-1} M_i = \mid w - a \mid\right\}}{f_V(a)}, \\ \text{where} \quad M_i &= \left(-\left(a_{\langle i \rangle} 2^i\right) + \left(t_{\langle i \rangle} 2^i\right)\right)\left(t_{\langle i \rangle} \oplus a_{\langle i \rangle}\right). \end{aligned} \tag{3}$$

4.3 Another illustrative example

Figure 5 further illustrates by example, the intuition behind the relation between single and multiple logic upsets and the attendant error-containing values and value deviations. In the case of single logic upsets affecting a variable (Figure 5(a)), the value deviation can only be a power of two, corresponding to one of each of the bits representing the variable incurring a logic upset that is not masked. This was captured in Equation 1. When multiple independent logic upsets may occur, all integer deviations in value of a variable are possible (Figure 5(b)), and this case was captured by Equation 3.

Equations 1 and 3 represent the probability distributions of *error-containing values* that may be taken on by unsigned n-bit data types in the presence of logic upsets in the underlying hardware. From these expressions, and given a known distribution of values taken on in the absence of upsets, such as the example illustrated in Figure 4 for the SPEC 2000 application `ammp`, it is possible to obtain expressions for the *deviation* in value that

will be incurred by variables in the presence of logic upsets. Such distributions of value deviation, represented by the value deviation PMF $f_M(m)$, can serve as a basis for a variety of program transformations to improve resilience to soft-errors; an example of one such transformation is presented in Section 6.

4.4 Analytic Expressions for Variable Value Deviation

The error-containing value, W was derived in Sections 4 to enable the generation of a closed-form expression for $f_M(m)$. The PMFs $f_M(m)$, $f_W(w)$ and $f_V(v)$ are related by

$$\begin{aligned} f_M(m) &= \Pr\{M = m\} \\ &= \Pr\{\mid W - V \mid = m\} \\ &= \Pr\{W = V + m\} + \Pr\{W = V - m\} \\ &= \sum_a f_W(a) f_V(a-m) + \sum_a f_W(a) f_V(a+m). \end{aligned} \tag{4}$$

The intuition behind the above is that the value deviation is m whenever the error-free random variable V differs from the error-containing random variable, W, by m, for all possible cases of V and W.

Based on Equation 4 and on the prior derivation of $f_W(w)$ in Equation 1, we can now obtain a closed form expressions for $f_M(m)$. For the case of unsigned n-bit values and singly-occurring logic upsets, substituting Equation 1 into Equation 4, we obtain

$$\begin{aligned} f_M(m) = \sum_{a=0}^{2^n-1} \Bigg(\sum_{i=0}^{n-1} f_V(a+2^i) f_{t\langle i\rangle}(0) + \sum_{i=0}^{n-1} f_V(a-2^i) f_{t\langle i\rangle}(1) \\ + \sum_{i=0}^{n-1} f_{t\langle i\rangle}(V_{\langle i\rangle}) f_V(a) \Bigg) f_V(a-m) + \sum_{a=0}^{2^n-1} \Bigg(\sum_{i=0}^{n-1} f_V(a+2^i) f_{t\langle i\rangle}(0) \\ + \sum_{i=0}^{n-1} f_V(a-2^i) f_{t\langle i\rangle}(1) + \sum_{i=0}^{n-1} f_{t\langle i\rangle}(V_{\langle i\rangle}) f_V(a) \Bigg) f_V(a+m). \end{aligned} \tag{5}$$

A similar expression of $f_M(m)$ for the case of multiple concurrent upsets can be obtained by substituting Equation 3 into Equation 4; we omit it here for brevity.

In the derivations thus far, there have been two missing components: the PMF for error-free values, $f_V(v)$ and the logic upset PMF, $f_{t\langle i\rangle}(k)$. The distribution of logic upsets, $f_{t\langle i\rangle}(k)$, is dependent on the hardware and environment in which applications are deployed. For example, from the eye diagram in Figure 1(c), it can be observed that the noise in high logic levels is larger in magnitude (up to 2 V below the nominal logic1), than the noise in low logic levels (which is approximately a maximum of +/- 1 V). It is thus more likely for a 1 to be misinterpreted as a 0 , than vice versa. In the absence of specific information for a given platform however, a reasonable approximation would be to assume 0→1 and 1→0 upsets are equally probable, and are uniformly distributed over the hardware state. The analyses presented in this paper are however not dependent on any such assumption.

The distribution of values taken on by program variables of different type ascriptions, $f_V(v)$, is an empirical property. Obtaining $f_V(v)$ would involve, for example, profiling large suites of applications, while monitoring the values taken on by all the variables of a given data type, within the individual programs. This empirical estimate of $f_V(v)$ would be meaningful if the collection of programs studied was sufficiently large. The results

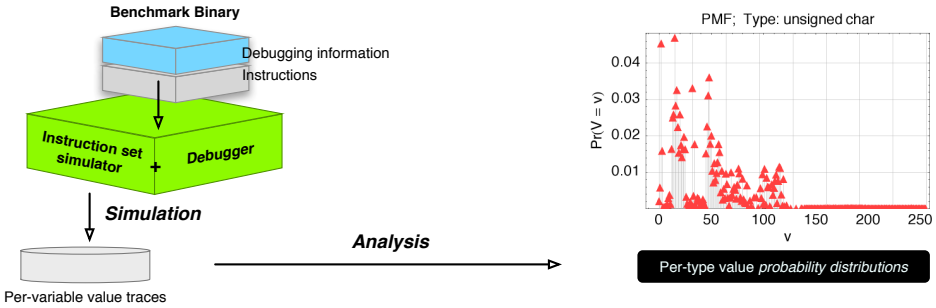


Fig. 6. Infrastructure for performing automated variable-level value tracing to construct per-type PMFs.

of such detailed empirical characterizations are presented in the next section.

5. EMPIRICAL DISTRIBUTIONS

To investigate the per-type statistical value distributions of variables, we extended an instruction-set simulator [Stanley-Marbell and Marculescu, 2007] with many of the capabilities traditionally found in a debugger such as GDB [Stallman and Pesch, 1993], to process the debugging information embedded in binaries loaded for execution on the simulator, as illustrated in Figure 6. This enables the instruction-set simulation environment, given any compiled binary, to automatically determine the list of all source-level program variables and their associated types, as well as the mapping between these variables and machine registers, static data sections, heap and stack memory addresses. During cycle-level simulation of the programs, memory and register accesses are correlated with the variables known to be mapped to them.

The SPEC CPU 2000 integer and floating point benchmark suites, as well as the MiBench embedded benchmark suite, were simulated using the aforementioned infrastructure, to characterize the statistical distributions of the values taken on by their source-level variables. Programs in the SPEC benchmark suite are meant to be representative of common PC and workstation applications, while the MiBench benchmark suite includes programs that are representative of a spectrum of embedded, mobile and desktop applications.

In what follows, we present the empirically measured probability distributions for free-standing variables with the C language data types `char`, `unsigned char` (`uchar`), `short int`, `int`, `unsigned int` (`uint`), `long int` and `double`, and pointers. For *each* benchmark, over a hundred variables of these different types are represented. The results presented *for each data type*, being aggregates over each benchmark suite, thus represent thousands of program variables, with over 10 million sample points, aggregated over time. For variables of type `double`, we show data for the upper 32 bits of the floating-point word, to keep the analysis tractable — the complete probability distribution for the full 64-bit floating point representations contain over 10^{19} data points.

The goal in presenting the distributions discussed in this section, is to present concrete empirical evidence of the nature of per-type variable value distributions, which are the last essential component in the value deviation distribution derivations.

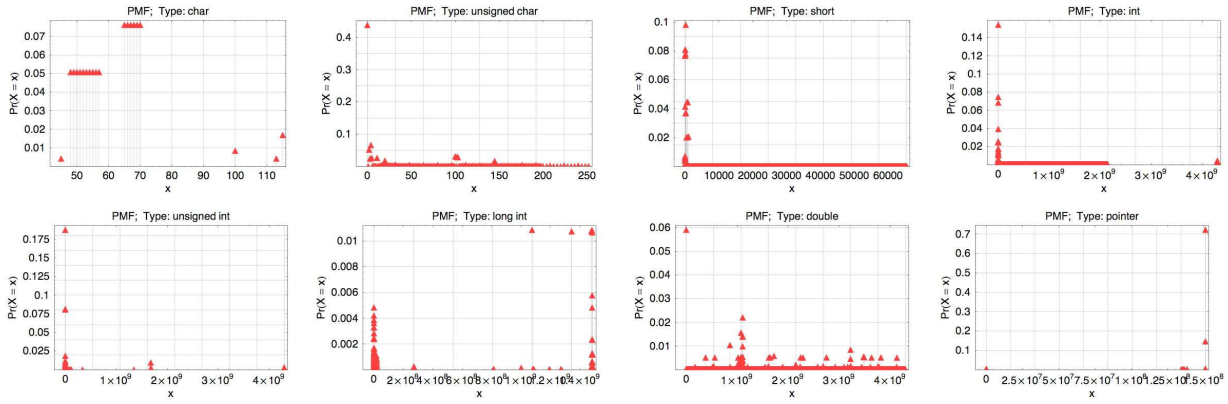


Fig. 7. PMFs for variable values of several basic data types in C language benchmarks from the MiBench benchmark suite (`basicmath`, `bitcount`, `qsort`, `susan`, `jpeg`, `lame`, `typeset`, `dijkstra`, `ghostscript`, `stringsearch`, `blowfish`, `rijndael`, `sha`, CRC32, FFT).

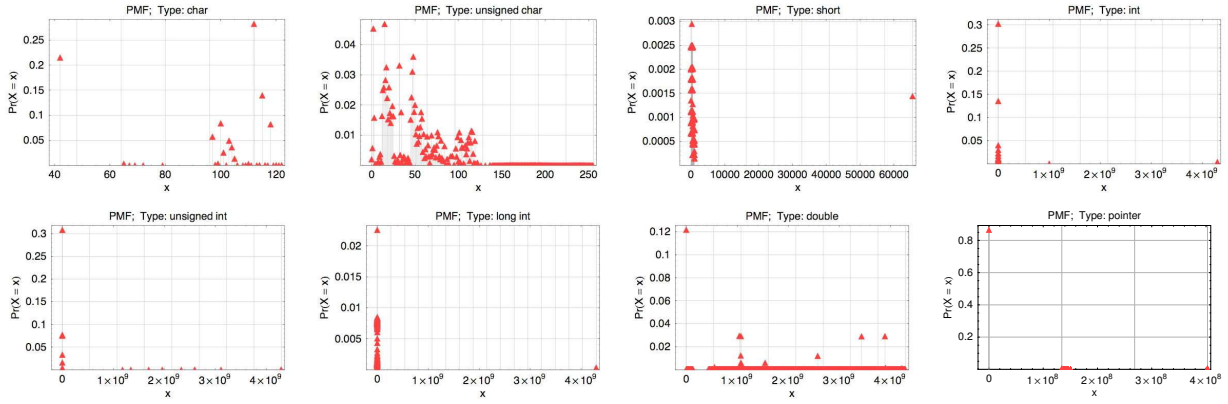


Fig. 8. PMFs for variable values of several basic data types in C language benchmarks from the SPEC CPU 2000 integer and floating point benchmark suites (`ammp`, `art`, `bzip2`, `cc1`, `equake`, `gzip`, `mcf`, `parser`, `vortex`, `vpr`).

5.1 Value distributions

Figure 7 and Figure 8 plot the per-type value distribution PMFs aggregated across each benchmark suite. In the figures, the horizontal axis represents values taken on by variables (the *support set* or *sample space*) and the vertical axis represents their probabilities. The distributions observed are consistent with common intuition. For example, across both benchmark suites, the distribution of values of type `char` are clustered around values 0 and the ASCII encoding value range for alphanumeric characters, 48 – 127. It is observed that for many data types, a small fraction of values carry a disproportionately large fraction of the probability density. For example, values close to zero are in many cases the values with the highest probability, often occurring up to a third of the time. This is an important fact to keep in mind when considering the effects of faults at runtime. It implies, for example, that applications partitioned over the hardware platform of Figure 1(a), are likely to mask the typical expected faults deduced from the eye diagram in Figure 1(c). The observations are also put to use when we develop compact representations of distributions later in the paper.

In general, the distributions of values of different types show similar trends across the different benchmark suites (and also, as will be seen in Section 5.2, across individual programs within a suite). The distributions showing the greatest similarity are those for the integer types (`short int`, `int`, `unsigned int` and `long`

Table 1. Summary statistics for value distributions.

(a) MiBench

	Mean	Stdev.	Median	Mode	Skewness
char	61.05	11.69	57	65 – 70	2.17
uchar	37.94	59.33	3	0	1.46
short	1640.11	7879.38	64	192	5.88
int	1.92×10^8	8.66×10^8	13	0	4.46
uint	1.05×10^8	6.20×10^8	63	0	6.23
long	5.23×10^8	6.59×10^8	2.07×10^6	1.07×10^9	0.56
double	1.54×10^9	1.04×10^9	1.10×10^9	0	0.98
pointer	1.35×10^8	4.60×10^7	1.51×10^8	1.51×10^8	-2.59

(b) SPEC CPU 2000

	Mean	Stdev.	Median	Mode	Skewness
char	94.63	28.30	112	112	-1.22
uchar	46.44	34.58	46	15	1.01
short	481.47	2487.04	345	255	25.82
int	2.79×10^7	3.43×10^8	3	0	12.32
uint	42244.2	1.03×10^7	4	0	341.49
long	2.59×10^6	1.05×10^8	63	1	40.70
double	1.70×10^9	1.23×10^9	1.07×10^9	0	0.48
pointer	2.22×10^7	6.37×10^7	0	0	3.62

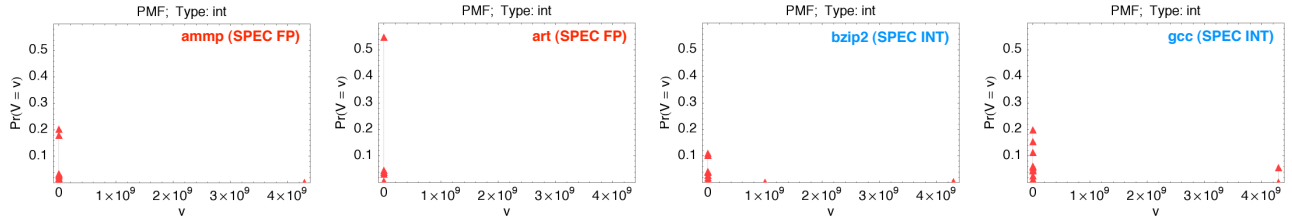


Fig. 9. Per-benchmark value distributions, for variables of type `int` in the SPEC 2000 benchmark suite.

`int`); this is encouraging, since these data types dominate the runtime accessed variables in most applications. The distributions for data type `double` both exhibit peaks at approximately 1×10^9 and 3×10^9. This is a result of the bit-level structure imposed by the IEEE 754 floating point format, as will be seen in Section 5.3. The distributions showing the least similarity are those for the character types. This is due to the fact that variables of type `char` and `unsigned char` are typically very dependent on program inputs, as they are often used to hold values such as strings. Despite this, it can still be seen that some general similarity exists, as the values taken on, across both application domains, are clustered around the values for the ASCII encoding of letters and numbers.

Tables 1(a) and 1(b) present relevant summary statistics for the value distributions. They reinforce the observations that (1) the most likely values for integer types are closer to zero (low means and medians, generally large positive skewness[2]), and (2) the most frequently occurring value is often zero (mode is zero for many per-type value distributions).

5.2 Per-program distributions

The analyses presented thus far have been aggregates across multiple programs, over time. It was previously observed that, even across collections of programs targeted at drastically different application domains (SPEC: desktop/workstation, versus MiBench: embedded), the most frequently accessed data type, `int`, showed significant similarity across domains. For the SPEC 2000 benchmark suite, Figure 9 illustrates the value distributions for variables of type `int` for several individual programs. While there are some differences in how much probability is attached to *individual* values (the stems with triangle tops in the graph), the general trend observed for the aggregate distribution across multiple programs holds. It is therefore acceptable to employ the aggregate distributions observed for the benchmark suite as a general case.

[2]The *skewness* of a distribution is a measure of its asymmetry. For example, uniform and Gaussian distributions both have a skewness of 0, while an exponential distribution has a skewness of 2.

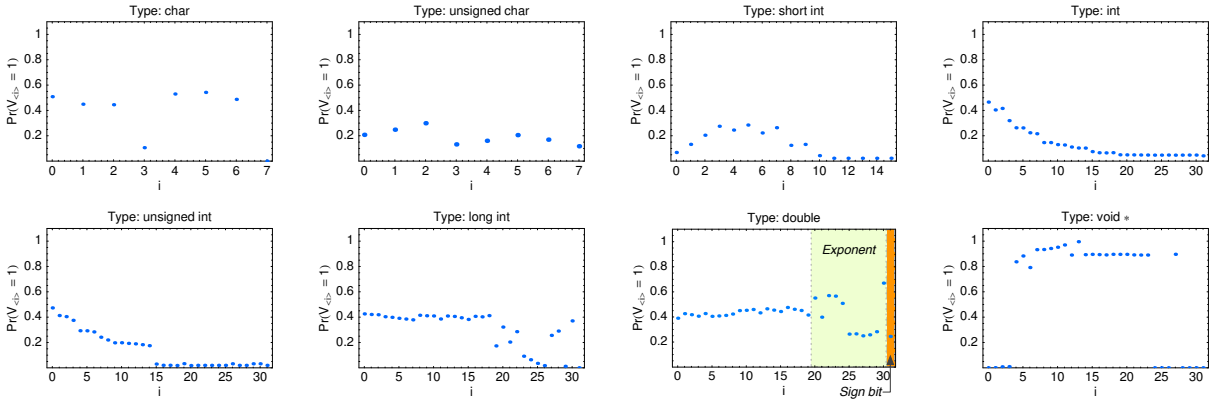


Fig. 10. PMFs for variable bit values (coordinate values) of several basic data types in C language benchmarks from the MiBench benchmark suite.

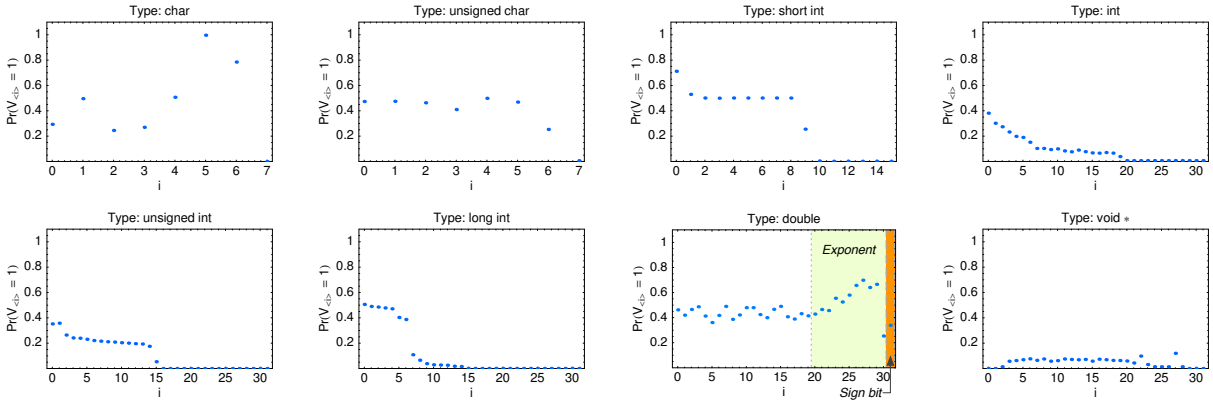


Fig. 11. PMFs for variable bit values (coordinate values) of several basic data types in C language benchmarks from the SPEC CPU 2000 integer and floating point benchmark suites.

5.3 Bit-level distributions

Bit-level distributions provide an alternate view of the distribution of values taken on by variables of different types. They indicate, for each bit in the layout of a data type, the probability of the bit taking on the logic value `1` (versus `0`). Under conditions of *statistical independence* between bit-level values, they can provide a compact representation of the value distributions shown previously.

Figure 10 and Figure 11 show the bit-level PMFs for several C language data types, across the dynamic executions of the MiBench and SPEC CPU 2000 benchmark suites. In the figures, the horizontal axes represent bit positions in the variables, and the vertical axes represent the probability of the given bit position taking on the logic value `1`. Across both suites of programs, the bit-level probability distributions for types `int` and `unsigned int` show the greatest similarity, both exhibiting bit probabilities for a `1` that asymptotically approach zero with increasing bit position, and in the case of `unsigned int`, exhibiting a sharp drop at bit 15.

For the most-significant 32 bits of double precision floating point values, the bit-level probabilities reflect the structure imposed by the IEEE 754 floating point format. They can be broken up into three groups: the *sign bit*, *exponent* and *significand* (overlaid in Figures 10 and 11). For the upper 32 bits of the double precision

floating-point values, the sign bit has a markedly different probability from the rest of the bits, of being a logic 1, while the upper 20 bits of the significand are distributed almost uniformly with probability of approximately 0.4, in both benchmark suites.

In order for empirical probability distributions to be used in any practical analysis, they must be representable in a compact form. From the previous section, it is obvious that using the complete empirical probability mass function is impractical, as it contains too many points — it is therefore desirable to somehow "compress" the distribution with a minimal loss of important information. While the bit-level probabilities may seem to provide a compact representation for value distributions, they do not. This is due to the fact that the random variables representing each bit position are not independent. To construct the value PMF from the bit-level PMFs would require the *joint PMFs* between all bits; the specification of this joint PMF is as large as the value distribution PMF. Approaches to abbreviating the distributions, such as curve-fitting, turn out to be inappropriate due to the lack of smoothness in the distributions, with often drastic differences in probability between adjacent values in the support set.

5.4 Compact representation of distributions

One approach for obtaining compact representations of value distributions is motivated by the observation that a small number of values in the support sets (the sets of possible values) of the PMFs for most data types, constitute a large fraction of the probability density. We will refer to these sets, for the n most-probable values, as the set ϕ_n. These most-probable values are different from *frequent values* distributed within memory [Yang and Gupta, 2002]. While frequent values were studied in the context of their frequency of occurrence within the general spatial distribution of memory words, ϕ_n represents the most frequent n values and their associated probabilities, for a given programming language data type, over the lifetime of a program.

Figures 12 and 13 show the amount of probability density in the set ϕ_n of n most-probable values versus n, for several of the primitive data types in the C programming language, for the MiBench and SPEC CPU 2000 benchmark suites, respectively. In the figures, the horizontal axes represent the size of the collection of most probable values, n, and the vertical axes represent the probability of a value being from this set. In many cases, a set of 100 support values carry upwards of 60% of the probability density. Looking at the amount of density in the sets ϕ_n however enables a *precise tradeoff* in the size of the fit (number of points) versus accuracy (amount of density covered).

The set ϕ_n can therefore be used as a compact representation for value distributions. In this form, the values in ϕ_n will be assigned probabilities according to the empirical measurements, with the remaining probability density spread across values outside ϕ_n. For example, for $n = 5$, the representation for the distribution of

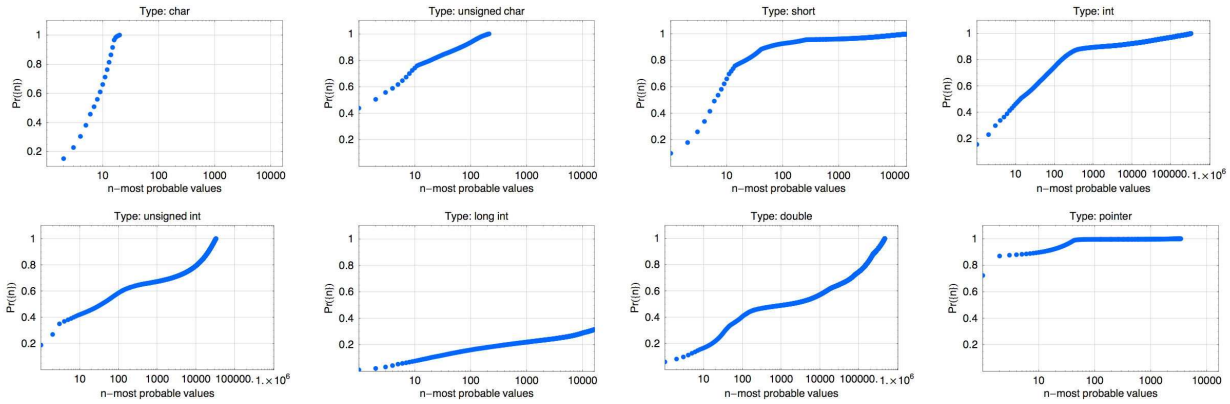


Fig. 12. The sets of n most probable values and their probabilities, for basic data types in C language benchmarks from the MiBench benchmark suite.

variable values of type `int`, based on empirical measurements from the SPEC CPU 2000 benchmark suite is

$$f_V(v) = \begin{cases} 0.3028 & : \quad v = 0 \\ 0.1358 & : \quad v = 1 \\ 0.0403 & : \quad v = 2 \\ 0.0300 & : \quad v = 4 \\ 0.0228 & : \quad v = 5 \\ 1.09 \times 10^{-10} & : \quad \text{otherwise} \end{cases},$$

and the corresponding compact value distribution for variable values of type `int` for the MiBench suite is

$$f_V(v) = \begin{cases} 0.1543 & : \quad v = 0 \\ 0.0749 & : \quad v = 1 \\ 0.0685 & : \quad v = 7 \\ 0.0394 & : \quad v = 2 \\ 0.0258 & : \quad v = 3 \\ 1.48 \times 10^{-10} & : \quad \text{otherwise} \end{cases}.$$

In the above, we have used the set ϕ_5 for clarity of presentation; in practice, it will be more accurate to use sets of the order of ϕ_{100}. The choice as to the size of the set can be made precise by looking at the data presented in Figures 12 and 13, and picking n based on the desired accuracy.

6. EXAMPLE APPLICATION OF $F_M(M)$: PROGRAM TRANSFORMATIONS TO BOUND VALUE DEVIATIONS IN THE PRESENCE OF LOGIC UPSETS

The PMF $f_M(m)$ derived in previous sections, characterizes the distribution of value deviations, for a given distribution of values taken on by program variables, $f_V(v)$, and a distribution of logic upsets that may occur in a computing system, $f_{t\langle i\rangle}(k)$.

Traditional approaches to fault tolerance, (e.g., triple modular redundancy [von Neumann, 1956], redun-

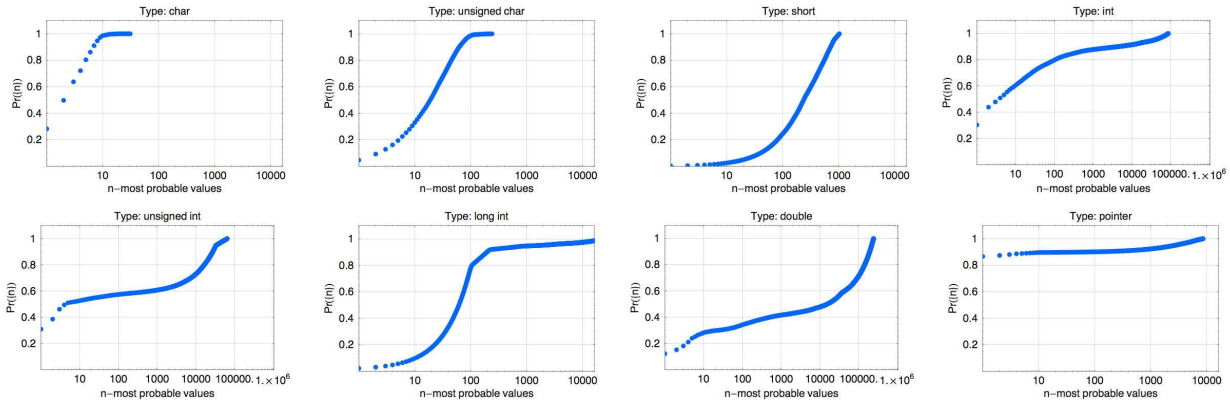


Fig. 13. The sets of n most probable values and their probabilities, for several basic data types in C language benchmarks from the SPEC CPU 2000 integer and floating point benchmark suites.

dant computations [Oh et al., 2002; Reis et al., 2005] or the various forms of forward error correction [Baylis, 1997]), attempt to *nullify* the effects of logic upsets, or to guarantee recovery from some fixed number of logic upsets, without particular concern for the *semantic disturbance* caused by upsets — no consideration of the role that different bit positions play in the digital arithmetic encoding of affected data values is made. In many applications however, it may be acceptable for the values taken on by variables to deviate within some bounds (i.e., different magnitudes of value deviation, m, may have different permissible probability). The analytic framework developed in this paper enables the tolerable deviations to be stated precisely, as constraints on $f_M(m)$ (derived in Section 4) — a constraint on $f_M(m)$ may be expressed as an upper bound, e.g., as $\Pr\{M > m\} \leq g(m)$. This reads as "the probability that the value deviation exceeds m, should always be less than or equal to $g(m)$."

The amount of value deviation tolerable, will naturally vary between applications (e.g., a datapath-dominated signal processing application versus a control-dominated application). It will also vary *within* an application. Ideally, therefore, one would like to be able to specify tolerable value deviations at the level of individual variables, alongside their type annotation. In practice, tolerable deviations need only be specified for a few variables, e.g., for those representing quantities in which some amount of value deviation *is* tolerable, such as variables representing pixel values. The tolerable deviations that can be permitted on other variables in the program may then be determined by dataflow analysis. In the context of the hardware platform introduced previously in Figure 1(a), this means that forward-error-correcting codes which provide differing amounts of protection for different bit positions, may be used in conjunction with increasing the inter-processor communication speed (and hence the probability of bit upsets). In particular, these tradeoffs may be exercised for particular variables whose contents are transferred over the interconnect, as a result of application partitioning for the platform's multiple processing elements.

With the permissible per-variable deviation distributions in hand, and with the analyses presented in this paper, it is then possible to determine new bit-level representations (selectively adding bit-level redundancy) that should be employed for variables when they reside in hardware structures, or are communicated across interconnect media, that may be susceptible to logic upsets.

```
1   f() {
2           e : const 2.71828;
3           a : int epsilon(1, 0.1);
4           b : int epsilon(x, e^(-3x));
5           c : int;
6           v : int;
7           ...

99          v = c;
100         a = 2;
101         b = a + v;
102 }
```

Fig. 14. Example to illustrate program-level error tolerance annotations: `a : type` represents variable definition.

6.1 Specifying tolerable deviation distributions in programs

One example of the manner in which language-level tolerable deviations may be specified in a programming language is presented in Figure 14. Line 3 in the program fragment defines a variable a, of type `int`, with a tolerable value deviation constraint (henceforth referred to simply as a *deviation constraint*)

```
epsilon (1, 0.1)
```

This constraint indicates that the programmer is willing to tolerate a value deviation of greater than 1, with probability 0.1. An example scenario in which this might be relevant would be if the variable represents a pixel color in an image processing application. In such a situation, small deviations in value might result in imperceptible changes in color, and the programmer-specified constraint states precisely how much value deviation is tolerable.

Following the analysis developed in this paper, the deviation constraint on variable a is identically the constraint $\Pr\{M_a > 1\} \leq 0.1$ (recall the summary of notation in Figure 3), i.e., $\bar{F}_{M_a}(1) \leq 0.1$, where M_a is the *value deviation random variable* for variable a. Similarly, line 4 in the program fragment defines a variable b with constraint $\bar{F}_{M_b}(x) \leq e^{-3x}$. The value read into the variable v on line 99, propagates through the program to the variable b (line 101), on which there exists a deviation-tolerance constraint. If this is the only use of the value stored in variable v, it can also inherit the relaxation of required correctness that the deviation tolerance constraint on b implies.

6.2 Deriving an encoding to constrain value deviation

Given tolerable deviations explicitly provided (or inferred) for program variables, as in the foregoing discussion, it is possible to formulate program transformations to take advantage of these permissible value deviations, in the presence of a some known worst-case distribution of logic upsets. In what follows, we use the distribution of tolerable value deviation to determine necessary *bit-level layout and redundancy* (encoding) of variables required to satisfy the programmer specified and inferred constraints.

For singly-occurring logic upsets, from Equation 5, we have an expression for the PMF, $f_M(m)$, in terms of the PMFs $f_V(v)$ and $f_{t\langle i\rangle}(k)$. These can be substituted into the expressions resulting from the language level constraints, to obtain an inequality in which the only unknown is the $f_{t\langle i\rangle}(k)$ which would be required to ensure the inequality holds. For singly-occurring upsets, from Equation 5 and language-level constraints of the form

$\Pr\{M > m\} \leq g(m)$, we obtain:

$$1 - \sum_{k=0}^{m} \left(\sum_{a=0}^{2^n-1} \left(\sum_{i=0}^{n-1} f_V(a+2^i) f_{t\langle i \rangle}(0) + \sum_{i=0}^{n-1} f_V(a-2^i) f_{t\langle i \rangle}(1) + \sum_{i=0}^{n-1} f_{t\langle i \rangle}(V_{\langle i \rangle}) f_V(a) \right) f_V(a-m) + \sum_{a=0}^{2^n-1} \left(\sum_{i=0}^{n-1} f_V(a+2^i) f_{t\langle i \rangle}(0) + \sum_{i=0}^{n-1} f_V(a-2^i) f_{t\langle i \rangle}(1) + \sum_{i=0}^{n-1} f_{t\langle i \rangle}(V_{\langle i \rangle}) f_V(a) \right) f_V(a+m) \right) \leq g(m). \quad (6)$$

Given values of m and $g(m)$, chosen by a programmer as part of the type ascription of a variable (Section 6.1), and given an $f_V(v)$ (determined empirically in Section 5), we can determine the particular $f_{t\langle i \rangle}(k)$, which we shall call $f^{req}_{t\langle i \rangle}(k)$, necessary to provide a solution to Equation 6. Knowing this *required* $f^{req}_{t\langle i \rangle}(k)$, then, given an $f^{BER}_{t\langle i \rangle}(k)$, resulting from a particular observed *bit error rate (BER)* for a particular hardware platform and operating environment, our goal is then to introduce redundancy at the level of bits, to reduce the effect of logic-upset-inducing events, from $f^{BER}_{t\langle i \rangle}(k)$ to $f^{req}_{t\langle i \rangle}(k)$.

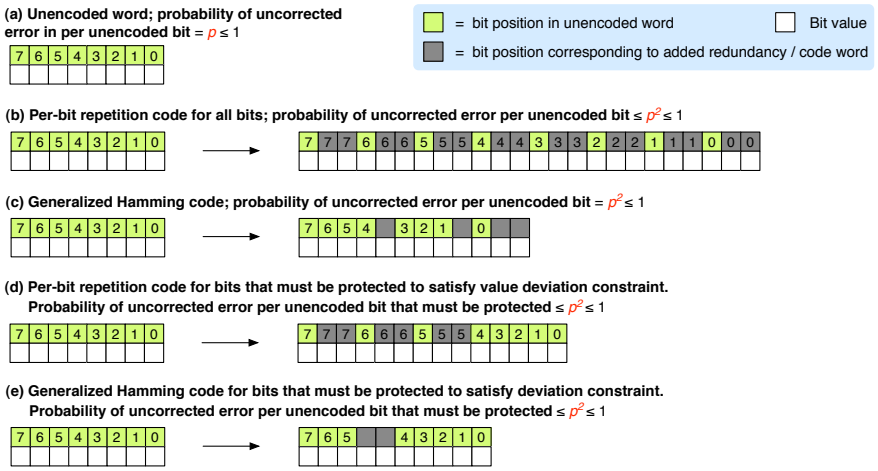


Fig. 15. Encoding for deviation tolerance.

As a concrete example, Figure 15 illustrates different methods for encoding unsigned 8-bit words to enable forward error correction of single logic upsets, reducing the probability of an uncorrected error from p to p^2. For example, let logic upsets occur with probability p, e.g., $p = 10^{-14}$ per hour, where it is desired to have a probability of uncorrected faults of at most $p = 10^{-28}$. The naive approach (Figure 15(b)) is to replicate each bit of the word (or the entire word) five times, and to take a majority vote for each bit position. Figure 15(c) shows the reduction in overhead using a simple but more intelligent generalized Hamming code [Baylis, 1997]. If on the other hand it was desired, not to reduce the probability of error to $p = 10^{-28}$, but rather to reduce the probability of *value deviation* caused by an upset, of magnitude greater than 32 (say), to 10^{-28}, this would mean (in the case of the single upset assumption), that only the most significant three bits would need to be replicated (Figure 15(d)) or encoded (Figure 15(e)). Similarly, a requirement of the probability of deviations greater than x, of $\frac{126.765}{x^{2.0}}$, sets the same constraint on deviations of magnitude 32 being 10^{-28}, but also permits deviations of magnitude 128, as long as they only occur with probability of 9.095×10^{-41}.

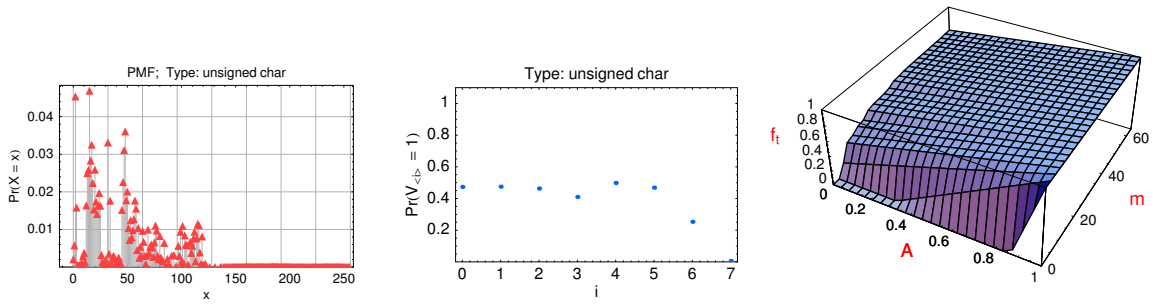


Fig. 16. Example empirical value distributions, $f_V(v)$, empirical bit state probabilities, and required bit upset probability given program-level constraints of $\Pr\{M > m\} \leq A$ as a function of m and A.

To illustrate, consider a scenario in which the logic upset phenomenon affects all bits with equal probability, thus $f_{t\langle i\rangle}(k) = f_t$. Examples of the variation of the required f_t as a function of both the tolerable value deviation, m, and the probability bound on the tolerable error, $g(m) = \text{constant} = A$, is shown in Figure 16. The empirical distributions in the figure are aggregates for variables of data type `unsigned char`, across applications from the MiBench benchmark suite. Each point on the surface in the figure represents an upper bound on the required f_t that will satisfy the corresponding constraints m and A. In the plot, for small magnitudes of permissible value deviation, m, and for small associated likelihood of such deviation being exceeded, A, the required worst-case bit upset probability approaches zero. As the amount (m) and probability (A) of permissible value deviation increase, so also does the permissible worst-case bit upset probability.

One simplistic approach to encoding is to employ bit-level replication, and to perform a majority vote over the replicated bits. From the required f_t shown in Figure 16, the amount of replication required, given a hardware substrate with logic upset PMF $f^{BER}_{t\langle i\rangle}(k)$, is given by:

$$2\left\lceil \frac{\log(f_t)}{\log(f^{BER}_{t\langle i\rangle}(k))} \right\rceil + 1 \tag{7}$$

6.3 Other applications of the analyses

The analyses presented thus far could be employed for purposes other than bounding value deviation under tolerance constraints. For example, when the bit upset probability required to satisfy a program's constraints are weaker than the bit upset rate of the hardware on which the said programs execute, probabilistic computations, which proceed with some probability of error, can be employed. Such computations have the potential to be performed at lower energy cost, and are an area of active research [Palem, 2003]. Another example application of the analysis is aggressive voltage scaling for application phases for which all the computed results can be proven to be tolerable of some amount of value deviation. By aggressively reducing operating voltage below thresholds required for guaranteed safe operation of circuits (permitting a minimum noise margin to be maintained), significant power consumption savings can be attained. Yet other applications include probabilistic analysis of programs [Rinard, 2006] and probabilistic techniques for ensuring program safety [Berger and Zorn, 2006].

7. SUMMARY AND FUTURE WORK

Single- and multi-bit logic upsets, when occurring in machine state representing integer- or real-valued program variables, may lead to errors that can be expressed in terms of a *value deviation* or *error-containing value* distribution. This paper presented analyses characterizing the probability mass functions for the value deviation, $f_M(m)$, and error-containing values, $f_W(w)$, in terms of the distribution of error-free values, $f_V(v)$, an empirical property, and the spatio-temporal distribution of logic upsets, $f_{t(i)}(k)$, a function of the operating environment, hardware and fault model. The analyses have applications in program transformations for trading-off program reliability for error-correction overhead, or reliability for power consumption. Detailed empirical characterizations of the error-free value distributions for a comprehensive suite of applications was presented, as well as techniques for compactly representing these distributions to facilitate their use in the analyses, and the theoretical and quantitative contributions of the paper were presented in the context of a concrete hardware platform that exhibits the properties assumed by the analyses. Current directions include extending the analysis to floating point data types, where their approximate real-valued nature permits interesting new directions.

References

R. C. Baumann. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. 5(3):305–316, september 2005.

J. Baylis. *Error Correcting Codes: A Mathematical Introduction*. Chapman & Hall/CRC, 1997.

E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. *SIGPLAN Not.*, 41(6):158–168, 2006. ISSN 0362-1340.

D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *HPCA*, pages 13–22, 1999.

M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 969–979, London, UK, 2000. Springer-Verlag. ISBN 3-540-67956-1.

B. Calder, P. Feller, and A. Eustace. Value profiling. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 259–269, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7977-8.

E. H. Cannon, D. D. Reinhardt, M. S. Gordon, and P. S. Makowenskyj. Sram ser in 90, 130 and 180 nm bulk and soi technologies. In *ICS '06: Proceedings of the 42nd Reliability Physics Symposium Proceedings*, pages 300–304, New York, NY, USA, April 2004. IEEE.

W. Heidergott. SEU Tolerant Device, Circuit and Processor Design. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 5–10, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-058-2.

R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop cyclone processor. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 216–226, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-366-3.

M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 138–147, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-767-7.

S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Synthesis*, 20(11):1355–1371, November 2001.

S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, 2003. ISSN 0272-1732.

N. Oh, S. Mitra, and E. J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Computers*, 51(2):180–199, 2002.

K. V. Palem. Energy aware algorithm design via probabilistic computing: from algorithms and models to moore's law and novel (semiconductor) devices. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 113–116, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-676-5.

G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4): 366–396, 2005. ISSN 1544-3566.

M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-282-8.

N. R. Saxena and W. K. McCluskey. Control-flow checking using watchdog assists and extended-precision checksums. *IEEE Trans. Comput.*, 39(4):554–559, 1990. ISSN 0018-9340.

T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor design. *IEEE Micro*, 19:12–23, Mar. 1999.

R. Stallman and R. H. Pesch. *Debugging with GDB: the GNU source-level debugger*. Free Software Foundation, Inc., pub-FSF:adr, 4.09 for GDB version 4.9 edition, 1993. Previous edition published under title: The GDB manual. August 1993.

P. Stanley-Marbell and D. Marculescu. Sunflower: Full-System, Embedded Microarchitecture Evaluation. *2nd European conference on High Performance Embedded Architectures and Computers (HiPEAC 2007) / Lecture Notes on Computer Science*, 4367:168–182, 2007.

M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 108–120, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-199-2.

Texas Instruments, Inc. Datasheet, MSP430x22x2, MSP430x22x4 Mixed Signal Microcontroller. 2006.

J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, pages 43–98, 1956.

D. Walker, L. Mackey, J. Ligatti, G. Reis, and D. August. Static typing for a faulty lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, September 2006. ACM Press.

V. Wong and M. Horowitz. Soft Error Resilience of Probabilistic Inference Applications. In *Proceedings of the Workshop on System Effects of Logic Soft Errors*, March 2006.

J. Yang and R. Gupta. Frequent value locality and its applications. *Trans. on Embedded Computing Sys.*, 1(1):79–105, 2002. ISSN 1539-9087.