Quantitative Analysis of Software- and Hardware-Fault Tolerance

Joanne Bechta Dugan
Department of Electrical Engineering
University of Virginia
Charlottesville, VA 22903-2442

The decision to develop redundant software versions as part of an approach to achieving software and fault tolerance is based on several criteria. These criteria include the consideration of the effects of potential failures (loss of life, severe environmental or economic consequences, etc), cost of developing and maintaining multiple software versions, and the expected dependability improvement that redundancy can achieve. Once a decision has been made to implement software fault tolerance through redundancy, there are several different architectural approaches that the designer can consider. The choice of whether to use fault tolerance, and how to achieve it, is facilitated by a quantitative comparative analysis of the dependability of several alternatives.

Dependability analysis of hardware and software fault tolerant architectures is complicated by the need to consider many types of faults which can occur. Hardware faults can be transient or permanent, software faults can be unrelated (occurring in a single version) or related (causing different versions to fail similarly). Certain combinations of hardware and software faults can collocate to defeat fault tolerance mechanisms. Comparative analysis is simplified by the use of a common modeling framework, using models that are simple to develop and understand.

We present such a modeling framework for the quantitative analysis of the reliability and safety of hardware and software fault tolerant architectures. We use fault trees and simple Markov models whose solutions are combined to produce the (time-dependent) probability that a proposed architecture produces an unacceptable result. The model includes transient and permanent hardware faults, imperfect fault recovery, reconfiguration, unrelated and related software faults and imperfect deciders.

In particular, we present example models for distributed recovery blocks, N-version programming and N self-checking programming. These models help elucidate the differences and similarities between the different architectures, and thus aid in understanding the different approaches.

The recovery block approach to software fault tolerance is the software analogy of "standby-sparing," and utilizes two or more alternate software modules and an acceptance test. The acceptability of a computation performed by the primary alternate is determined by an acceptance test. If the results are deemed unacceptable, the state of the system is rolled back and the computation is attempted by the secondary alternate.

In the NVP method, N independently developed software versions are used to perform the same tasks. They are executed concurrently using identical inputs. Their outputs are collected and evaluated by a decider. If the outputs do not all match, the output produced by the majority of the versions is taken to be correct.

The NSCP architecture considered in this paper is comprised of four software versions and four hardware components, each grouped in two pairs, essentially dividing the system into two halves. The hardware pairs operate in hot standby redundancy with each hardware component supporting one software version. The version pairs form self-checking software components. A self-checking software component consists of either two versions and a comparison algorithm or a version and an acceptance test. In this case, error detection is done by comparison. The four software versions are executed and the results of V1 and V2 are compared against each other, as are the results of V3 and V4. If either pair of results do not match, they are discarded and only the remaining two are used. If the results do match, the results of the two pairs are then compared. A hardware fault causes the software version running on it to produce incorrect results, as would a fault in the software version itself. This results in a discrepancy in the output of the two versions, causing that pair to be ignored.

We determine parameter values from an experimental implementation of N-version programming, in which multiple versions of a real-world "autopilot" function were developed. A quantitative comparison of these three architectures reveals several interesting features of the architectures, and demonstrates that a significant reduction in failure probability can be achieved through the use of software fault tolerance.