An Adaptive N-variant Software Architecture for Multi-Core Platforms: Models and Performance Analysis

Li Tan[1] and Axel Krings[2]

[1] School of Electrical Engineering and Computer Science
Washington State University
litan@wsu.edu

[2] Department of Computer Science
University of Idaho
krings@uidaho.edu

Abstract. This paper discusses the models and performance analysis for an adaptive software architecture, which supports multiple levels of fault detection, masking, and recovery through reconfiguration. The architecture starts with a formal requirement model defining multiple levels of functional capability and information assurance. The architecture includes a multi-layer design to implement the requirements using N-variant techniques. It also integrates a reconfiguration mechanism that uses lower layers to monitor higher layers, and if a fault is detected, it reconfigures a system to maintain essential services. We first provide a general reliability model (based on generalized stochastic Petri nets) for such a system with cross-monitoring for reconfiguration. Next, we define a probabilistic automaton-based model for behavioral modeling of the system. This model is especially suitable for modeling security problems induced by value faults. Whereas the Petri net allows for reliability modeling and reconfiguration, the performance analysis of the system is given via probabilistic model checking. The models are experimentally evaluated and compared. With the current widespread deployment of multi-core processors, one question in software engineering is how to effectively harness the parallel computing power provided by these processors. The architecture presented here allows us to explore the parallel computing power that otherwise may be wasted, and uses it to improve the dependability and survivability of a system, which is validated by our performance analysis.

1 Introduction

With the introduction of multi-core processors and their wide-spread deployment ranging from laptops to cloud computing, a question has been raised on how we can effectively exploit the parallel resources. It is projected that the level of application-exploitable parallelism will limb behind the number of cores in multi-core processors and GPU (graphics processing units) since to date most applications only allow limited parallelism. One way of harnessing these otherwise unused or underutilized resources is to use them for the purpose of increasing dependability, security, and survivability of a system, but doing so requires the support from adaptive software.

Two key features of an adaptive software are (1) the ability to monitor its own execution and (2) the ability to reconfigure itself based on the result of runtime monitoring [6]. While the proposed benefits of adaptive software is promising, developing adaptive software also raises some challenging questions. First, in many cases self-adaptation adds one more dimension of complexity to often already complicated dependable system designs. A question is how to specify requirements and implement them in a way that facilitates orderly and verifiable system reconfiguration. Second, a system may be subject to a variety of faults. So a challenge is how one could compartmentalize and diversify system design so the system can be resilient to different types of faults. This may be especially relevant in safety-critical applications. Finally, runtime monitoring requires additional computation power. Thus, it is important that our design can make efficient use of the underlying hardware architecture to minimize overhead.

To address the first challenge, we use a formal model to specify requirements for self-adaptation and then propose a multi-layered assured architecture to realize requirements expressed in the formal model. The Adaptive Functional Capability Model (AFCM) introduced in [10] defines levels of capabilities for each system functionality. The AFCM specifies how a system shall reconfigure itself and scale down its functional capabilities while still providing essential services and information assurance. Each level of functional capability in the AFCM will then be implemented as a layer in a *Multi-layered Assured Architecture Design*. The architecture design embeds a *Monitoring and Reconfiguration Module* (MRM) that uses lower-layer functionalities as reference to monitor high-layer functionalities. In case a fault is detected, the system reconfigures itself by disabling affected layers, while lower layers still maintain essential services.

To further improve system resilience, we use a diversified layered design based on N-variant techniques in each layer [3, 4]. The N-variant techniques use redundant executions to reduce system vulnerability to common-mode faults. The expectation is that redundant but dissimilar implementations reduce or eliminate common-mode faults. Dissimilarity is typically discussed in the context of N-version programming [1] dating back to the late 70s. In N-version programming it is assumed that several software development groups independently derive programs from the same specification. The concept of N-variant software is inspired by N-version software, but in N-variant software different variants are generated in a more automated fashion. In both cases a fault is detected if a difference is detected between outputs generated by two versions or variants.

Redundant executions exercised by multiple variants and extra work of runtime monitoring requires additional computational power. To reduce overhead, our N-variant-based implementation takes advantage of multi-core hardware. Most new general-purpose computers incorporate dual or quad-core processors and higher numbers of cores are already used widely in GPUs. Whereas in theory the computational capabilities increase with the number of cores, it becomes difficult to exploit sufficient parallelism to keep all cores utilized. Most common applications still allow little parallelism and it is likely that cores may be underutilized or running idle. In our approach, unused or underutilized cores are exploited to increase reliability, security, and survivability. Specifically, multiple variants execute on different cores, and if they can execute on idle cores, this overhead can be largely absorbed. This was also shown in [8] where multi-variants executed in multi-core systems. Our approach extends this by making extensive use of N-variant implementation at each layer of functional capability. In general, the lower a layer, the more variants it may have in order to provide a higher degree of resilience and information assurance. Nevertheless, the exact number of variants and their configuration required at each layer depends on the type and number of faults that are to be detected or masked.

The contributions of this paper are three-fold. First the hierarchical formal framework for adaptive N-variant programs outlined in [10] is applied to a system with multiple functionalities. Second, a reliability model is derived for such systems using Generalized Stochastic Petri Nets (GSPN). This model specifies how N-variant executions with different levels of capabilities are managed, and it models cross-monitoring as a mechanism to initiate survivability measures. Third, we develop a stochastic behavior model and apply probabilistic model checking to it for analyzing the performance of our adaptive software architecture. Whereas the GSPN model allows to determine the reliability of the system, the stochastic behavior model incorporates the consequences of data-dependent false negatives, and it aids in the formal verification towards an implementation. We will show how one can use the reliability model to derive the theoretically achievable reliability that is independent of the data, and how one can use the independently derived behavioral model to verify the results of the reliability analysis, quantify the impact of data dependent coincidental faults, and aid in the derivation of implementations.

The rest of the paper will be organized as follows: Section 2 introduces a formal model for specifying levels of functional capabilities, a layered adaptive architecture design to realize the adaptive functional capability model, and an N-variant-based approach to implement the architecture design. Section 3 discusses the features and benefits of our adaptive software architecture in terms of its adaptivity and survivability. Section 4 gives a Generalized Stochastic Petri Net (GSPN) model that we use for reliability analysis. Section 5 develops a stochastic behavior model that is used in conjunction with probabilistic model checking for analyzing performance of our software architecture and also validating the result of reliability analysis. Section 6 discusses the outcome of our computational experiments that are designed for assessing the reliability and performance of our software architecture. Finally Section 7 concludes the paper.

2 Specification Model & Architecture Design

2.1 Adaptive Functional Capability Model

The Adaptive Functional Capability Model (AFCM) specifies multiple functionalities with adjustable levels of capability. The model attaches each functionality to layers of capability. During requirement elicitation, a development team works with stake holders of a project to identify not just functionalities, but also

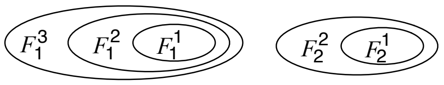

Fig. 1. AFCM for functionality F_1 and F_2

capability levels for each functionality. These capability levels specify graceful degradation in case of faults or when under attack.

Assume that the system is comprised of functionalities $F_1 \cdots F_m$. Figure 1 shows the AFCM for two sample functionalities F_1 and F_2. The requirements for F_1 define three levels of capabilities: F_1^1 defines the set of core operations that are mission-critical, F_1^2 includes F_1^1 and some non-critical but value-added operations, and F_1^3 adds some more value-added operations. We write $F_1^1 \preceq F_1^2 \preceq F_1^3$, where $\preceq$ is a preorder on the capability levels. The exact semantics of $\preceq$ is defined and interpreted based on application context. In [10] we described a transaction-based asynchronous system that contained mission-critical data and non-mission-critical but value-added data to demonstrate the AFCM.

The purpose of the AFCM is to specify not only functional requirements, but also requirements for adaptiveness. It has two features to serve its purpose: First, the model associates each functionality with capability levels, which specify reconfiguration requirements for the functionality. It states that, in the event of a fault, e.g., the system has been compromised, a system shall scale back its services in an orderly manner by following the capability levels defined in the AFCM, e.g., recovering to a lower level implemented in the next layer down; Second, the definition of capability levels also facilitates reconfigurable design. It requires that the system behavior at a higher capability level shall be an extension of the behavior at the lower level. Hence, we can use the behavior at a lower capability level as a reference for monitoring the behavior at a higher level, and an implementation for capability levels provides a path for a system to scale itself back.

It shall be noted that each functionality in an AFCM may have its own hierarchy of capability levels reflecting its requirement for adaptiveness. For example, F_1 and F_2 in Figure 1 have different levels of capabilities.

2.2 Layered N-variant Architecture

To implement the reconfiguration requirements specified in the AFCM using different levels, a layered adaptive architecture design is used. Each layer realizes its corresponding AFCM level using N-variant techniques such as shown in [3, 8]. Figure 2 shows an example of the adaptive N-variant architecture for two functionalities F_1 and F_2. The architecture has a layered structure. Each vertical layer realizes its respective capability level, i.e., it implements sequences of operations specified for its capability level. A layer may be disabled if it is found not functioning correctly, i.e., if a fault is detected. Fault detection is the result of redundancy management at the specific layer, or the layer beneath it, which has the monitoring abilities of its capabilities at the layer above it. The capability level of the entire functionality is decided by its highest enabled layer. Each layer itself is a collection of variants implementing its capability level. Variants are systematically diversified so that it is unlikely that a common-mode fault can occur [3, 8], e.g., for a given fault model, an attacker can not compromise all the variants without being detected and/or the fault being masked. One could argue that a lower layer should have more variants to improve resilience of the functionality. Each functionality may have a different layered structure that reflects its adaptability requirement. For example, in Figure 2 F_1 and F_2 are implemented by (L_1^1, L_1^2, L_1^3) and (L_2^1, L_2^2), respectively.

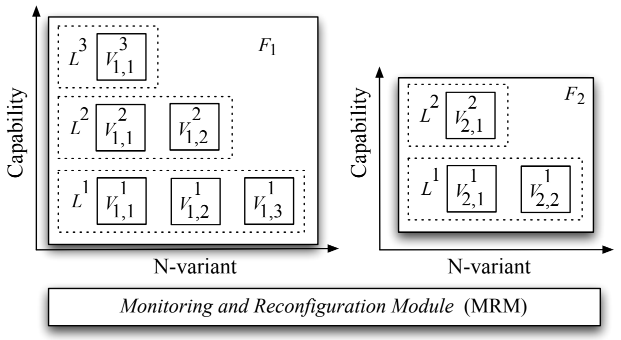

Fig. 2. A layered adaptive N-variant architecture design for the AFCM of Figure 1

3 Adaptability and Survivability

The layered adaptive N-variant architecture improves system resilience by supporting 1) real-time fault detection though redundancy management and cross-layer monitoring, 2) fault masking, and 3) system reconfiguration. The architecture design in Figure 2 includes the Monitoring and Reconfiguration Module (MRM). Critical or sensitive functionalities are implemented using the layered N-variant architecture and the MRM acts as a sentry for layered N-variant components. The MRM monitors and sanctions the communication in and out of the N-variant components. Together, the N-variant-based layers and the MRM provide runtime monitoring and real-time fault tolerance with reconfiguration, essential for an adaptive system.

Runtime monitoring by the MRM: The MRM uses observable behavior of a lower layer to decide whether the layer above is compromised or not. If a fault is detected, it reconfigures the system by disabling affected layers while essential functional capabilities are still provided by the lower layer(s). In section 2.1

capability levels in the AFCM are defined in such a way that a sequence of operations specified at a higher level is an extension of some sequence of operations specified at a lower level. In our layered adaptive N-variant architecture, all the layers process incoming requests concurrently. Since a layer L^i is an implementation of an AFCM capability level F^i, a sequence of operations executed by layer L^i shall be *included* in a sequence of operations executed by layer L^{i+1}. Should this not be the case it indicates problems (i.e., a fault) in L^{i+1}. The lower layer L^i is realized using N-variants of simpler implementations and potentially a higher degree of redundancy. It is argued that lower complexity implementations together with more stringent analysis/testing at L^i is assumed to make variants in L^i more reliable than in L^{i+1}. A larger degree of N-variants also increases reliability, as it implements a *k-of-N* configuration.

Real-time reconfiguration: The AFCM provides a reconfiguration plan in which a functionality can scale back its services in an orderly manner, and thus provide graceful degradation. A layer L^i serves as the backup for layer L^{i+1} above it. A lower layer forgoes some functional capability in lieu of improved dependability. If the MRM detects a fault in layer L^{i+1}, it disables L^{i+1} and the system automatically scales its capability to the level implemented by L^i. For the sake of completeness it should be noted that capabilities can not only be decreased, but also extended should the need arise, e.g., after recovery or repair.

The layered adaptive N-variant architecture is designed for improving system survivability for mission-critical applications: the capability of shifting to a lower layer (upon detection of a fault) provides a contingency plan that allows a system to scale back its services towards essential services as the result of the fault.

4 Reliability Analysis

In this section we discuss the impact of the layered model from a reliability analysis point of view. Before establishing a formal model for the analysis we need to establish the link to previous research. Multi-variant approaches have been previously described in [3, 4, 8, 12]. However, these models are described within a single layer, and their N-variant models constitute special cases of the layered architecture presented here. Thus, the cited approaches can be adopted at any layer within our architecture. It should be noted that they all have specifications and implementation at the same level and layer respectively. This means that the approaches deal with fault detection and possible treatment dependent on the degree of redundancy. However, adaptability and graceful degradation as described above is not supported. For example, the multi-variant scheme described in [8] uses two variants of memory referencing. Both variants implement the same functional capability, e.g., at layer L_1 with variants $V^1_{1,1}$ and $V^1_{1,2}$. The model in [3] has the similar limitation.

The application of N-variant approaches is not simply another way of generating dissimilarity similarly to N-version programming, but the specific derivation of variants that are designed to minimize common-mode faults, to the point of predictable elimination. For example, the memory management in [8] practically eliminates the potential for buffer overflows, since the two variants use reverse memory allocation (one uses forward and the other reversed allocation). In order for a buffer overflow attack to succeed not only would both variants have to be attacked at the same time, but, more importantly, the buffer overflow would have to have meaning in two directions. The latter implies that the overflow would have to "flow" into reverse memory in the two variants, and only a buffer overflow that acts as a palindrome could have the potential to succeed. The approach in [12] is similar in nature in that memory is partitioned in such a way that a valid access in one variant is invalid in the other. Thus, given the schemes described in [8] and [12] it is statistically very unlikely that two modules produce the same fault as the result of code injection.

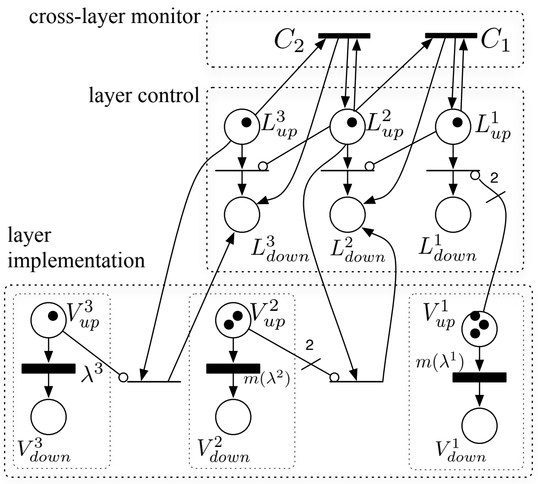

Fig. 3. Complete GSPN Model

The analysis of the multi-level and multi-layer approach is described using the example of functionality F_1 shown in Figure 2. F_1 has three levels, F_1^1, F_1^2 and F_1^3, and their respective layers L^1, L^2 and L^3 use 3-variant, 2-variant, and simplex implementations respectively. If we consider each layer individually then, given the levels of redundancy, one can mask one value fault at L^1, one can only detect (but not mask) one value fault at L^2, and there is neither detection nor correction potential at L^3.

The reliability and survivability of functionality F^1 is modeled in the generalized stochastic Petri net shown in Figure 3. The net is drawn as three subnets: the cross-layer monitor, the layer control and the layer implementation. The three N-variant layers are modeled in the three subnets of the layer implementation subnet and represent TMR, duplex, and simplex models of layers L^1, L^2 and L^3 respectively. Each of these three simple nets only model their associated layer in isolation, i.e., they constitute 2-of-3, 2-of-2, and 1-of-1 systems. Note that only the timed transitions of the TMR and duplex depend on the markings of their input places, reflected by the marking functions $m(\lambda^1)$ and $m(\lambda^2)$ respectively. The simplex at layer 3 has a fail rate of simply λ^3.

The layer control subnet models the interaction between layers, which are assumed to be either up or down. Initially all three layers are up as indicated by the single tokens in L^1_{up}, L^2_{up} and L^3_{up}. If a layer fails, i.e., if the threshold of tokens is reached in one of the subnets of the layer implementation subnet, the layer control subnet automatically disables the corresponding layer and all higher layers, i.e., failure at layer i will shut down all layers $\geq i$. For example, a failure of one of the variants in layer 2 results in a token being absorbed in V^2_{up}. This in turn disables the inhibitory arc at V^2_{up} (which needs two tokes to inhibit) and the associated transition will "move" the token from L^2_{up} to L^2_{down} in the layer control subnet. The lack of a token in L^2_{up} will cause the transition between L^3_{up} and L^3_{down} to fire. The result is that both L^2_{down} and L^3_{down} have a single token. The probability of a token in L^i_{down} is thus the unreliability of layer L^i. Alternatively, the probability of a token in L^i_{up} is the reliability of L^i.

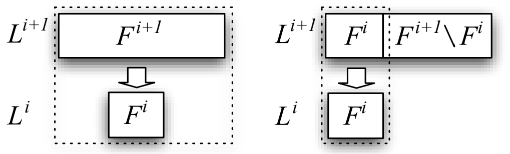

Fig. 4. Cross-layer monitoring scope

Monitoring across layers is modeled with the cross-layer monitoring subnet of Figure 3. To describe the exact approach for cross-layer monitoring consider Figure 4, which shows the relationship between two levels of requirements for functionality F, i.e., between F^i and F^{i+1}. Cross-monitoring of layer L^i on L^{i+1} is achieved by comparing results from L^{i+1} to those computed by the more reliable layer L^i. However, the functional capabilities of the layers are not the same, e.g., as was shown in Figure 1. This makes realistic cross-layer monitoring dependent on the computations of F^{i+1} and F^i that can be effectively compared. The resulting extreme cases are shown in Figure 4. To the left we have the scenario in which layer L^i can monitor all functionalities of layer L^{i+1}. This is different in the right scenario in which layer L^i can only monitor a subset of the functionality at layer L^{i+1}. Note that $F^i \prec F^{i+1}$. The respective implementations are in layers L^i and L^{i+1}. Note that layer L^{i+1} consists of the implementations of the operations of the lower layer based on F^i as well as the value-added operations specified by $F^{i+1} \setminus F^i$. Thus cross-layer monitoring is limited to operations specified by F^i. Realistic cross-layer monitoring is anywhere in-between the two extreme scenarios and is application dependent.

The Stochastic Activity Network (SAN) of cross-layer monitoring is shown in Figure 5. The transition is activated when operations specified by F^i differ in

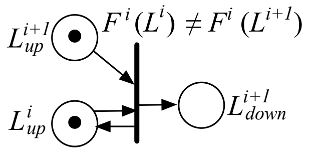

Fig. 5. SAN for cross-layer monitoring

layer L^i and L^{i+1}, i.e., if $F^i(L^i) \neq F^i(L^{i+1})$, where notation $F^i(L^j)$ indicates the functional specification with respect to layer L^j. Since F^{i+1} includes F^i the MRM indicates fault-free behavior if $F^i(L^i) = F^i(L^{i+1})$.

Cross-layer monitoring is modeled in the cross-layer monitoring subnet of Figure 3 by translating the condition $F^i(L^i) \neq F^i(L^{i+1})$ of the SAN in Figure 5 into cross-monitoring detection rates C_i in the GSPN.

It is simple to establish the exact reliability of functionality F_1 when the fail-rates are known. However, in the presence of malicious faults, e.g., hacking attacks or exploits, the assumption of constant fail-rates does not hold anymore. In that case, the GSPN stays the same, whereas the formal analysis of the net becomes much more complicated. The reason is that the constant fail rates of the timed transitions have to be replaced by time-dependent hazard functions. The GSPN of Figure 3 will be the basis for the reliability simulations in Section 6.

5 Stochastic Models

To evaluate the performance of our architecture, we model its stochastic behavior using probabilistic automaton. We then apply probabilistic model checking to study the performance of the model. We are particularly interested in two metrics: service availability and information security. To help illustrate our approach, we will use an example based on layer L^1 of functionality F_1 in Figure 2.

To model the component-based design in Section 2.2, our stochastic model is a parallel composition of layers and the Monitoring and Reconfiguration Module (MRM), which in turn is a parallel composition of Monitoring and Reconfiguration Sub-Modules (MRSM). Each MRSM is associated with a layer. The MRSM monitors and reconfigures its associated layer by using the output from the layer below. This is consistent with the cross-layer monitoring SAN shown in Figure 5.

Preliminary: We model our software architecture and its components as probabilistic finite automata [7]. A probabilistic finite automaton extends transitions in a finite automaton with probabilities. Formally a probabilistic automata is a tuple $\langle Q, \Theta, \delta, Q_0, F, P_\delta, P_0 \rangle$, where Q is a set of states, Θ is a set of input symbols, $\delta \subseteq Q \times \Theta \times Q$ is a set of transitions, $Q_0 \subseteq Q$ is a set of start states, $F \subseteq Q$ is a set of accepting states, $P_\delta : \delta \rightarrow (0, 1]$ assigns each transition a probability, and $P_0 : Q_0 \rightarrow (0, 1]$ assigns each start state a probability. In addition, $\Sigma_{q \in Q_0} P_0(q) = 1$ and for each $(p, a, p') \in \delta$, $\Sigma_{q \in \{q \mid (p,a,q) \in \delta\}} P_\delta((p, a, q)) = 1$.

A successful run of the probabilistic automaton $B = \langle Q, \Theta, \delta, Q_0, F, P_\delta, P_0 \rangle$ is a sequence $\rho = q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_n} q_n$ such that $q_0 \in Q_0$, $(q_i, a_{i+1}, q_{i+1}) \in \delta$ for $0 \le i < n$, and $q_n \in F$. Given a sequence of inputs $a_1 \cdots a_n$, the probability that the automaton B runs ρ is $P_0(q_0) \times \Pi_{0 \le i < n} P_\delta((q_i, a_{i+1}, q_{i+1}))$.

In this paper, we use an extension of probabilistic automata that supports state variables and input variables. A state is designated by a predicate over state variables. The predicate must be true when the state is active. Input variables describe environmental inputs. Instead of input symbols, transitions are guarded by a predicate over input and state variables. A transition is enabled if its source state is active and its guard is true.

Stochastic Models of N-variant layers: Figure 6 shows a probabilistic automaton that models L^1 of F_1 in Figure 2. It describes the behaviors of 3 variants in L^1 and also a built-in voting mechanism that decides which variant(s) are *working.* Models of N-variant layers use three state variables:

1. v is the number of working variants. The built-in voting mechanism decides the status of variants by applying a threshold voting function. For example, if value faults are considered and only 2 of 3 working variants produce the same result, then simple majority vote will update the status of the minority variant as *not* working.
2. w is the status of a layer. Initially all layers are working. If all the variants in a layer are marked as *not* working, the layer will be marked as *not* working. It should be noted that a variant marked working may still produce incorrect results, as indicated by its e flag.
3. e is an error flag. $e = true$ indicates that an erroneous output is produced by the layer. This could happen when, for example, all the working variants produce the exact same erroneous output, although its probability is small due to N-variant implementation.

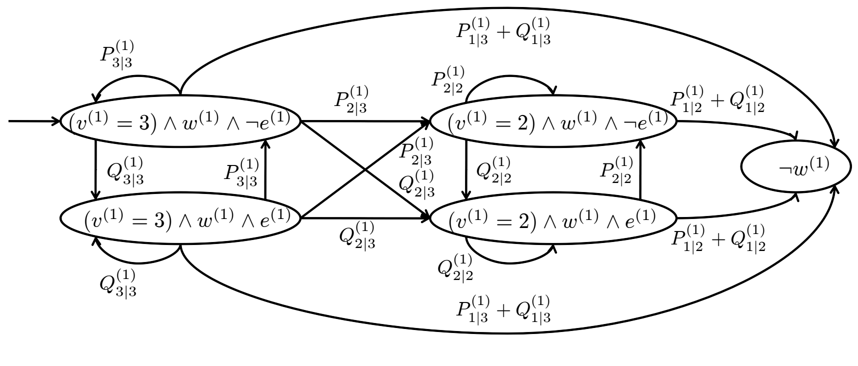
$(v^{(1)}=3) \wedge w^{(1)} \wedge \neg e^{(1)}$
$(v^{(1)}=3) \wedge w^{(1)} \wedge e^{(1)}$
$(v^{(1)}=2) \wedge w^{(1)} \wedge \neg e^{(1)}$
$(v^{(1)}=2) \wedge w^{(1)} \wedge e^{(1)}$
$\neg w^{(1)}$
$P^{(1)}_{3|3}$
$Q^{(1)}_{3|3}$
$P^{(1)}_{2|3}$
$Q^{(1)}_{2|3}$
$P^{(1)}_{2|2}$
$Q^{(1)}_{2|2}$
$P^{(1)}_{1|2} + Q^{(1)}_{1|2}$
$P^{(1)}_{1|3} + Q^{(1)}_{1|3}$

Fig. 6. Probabilistic automaton for L^1 of F_1

The superscript (i) on variables indicate which layer they are associated to. For example, $v^{(1)}$ indicates the number of *working* variants in layer L^1. We also use the following notations: given n variants, $P_{k|n}$ is the probability that the maximal number of variants producing the same result is k and there are k variants producing the same *correct* result; $Q_{k|n}$ is the probability that the maximal number of variants producing the same result is k and there are *no* k variants producing the same *correct* result.

Stochastic Models of MRMs: Since the functional capability of the higher layer subsumes that of the lower layer, the output from a layer can be used to monitor the layer above for the services provided by both layers. This is implemented by the Monitoring and Reconfiguration Module (MRM), as described in Section 3. An MRM also decides whether the higher layer shall be disabled when there is a discrepancy in the common services provided by two adjacent layers.

The MRM is a collection of Monitoring and Reconfiguration Sub-Module (MRSM), each of which monitors a particular layer using the output from the layer below. Figure 7 shows a probabilistic automaton that models the MRSM for layer L^2. Models of MRSMs and MRMs use the following two variables: d, a Boolean variable indicating whether the output of a layer is disabled by the MRM, and; c, a Boolean variable indicating whether a layer is compromised. $Q^{(1,2)}$ in Figure 7 is the probability that, when layer L^1 is working and not disabled, layer L^2 is working, and both layers produce erroneous results (i.e. $\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge e^{(1)} \wedge e^{(2)})$), the results produced by both layers are same.

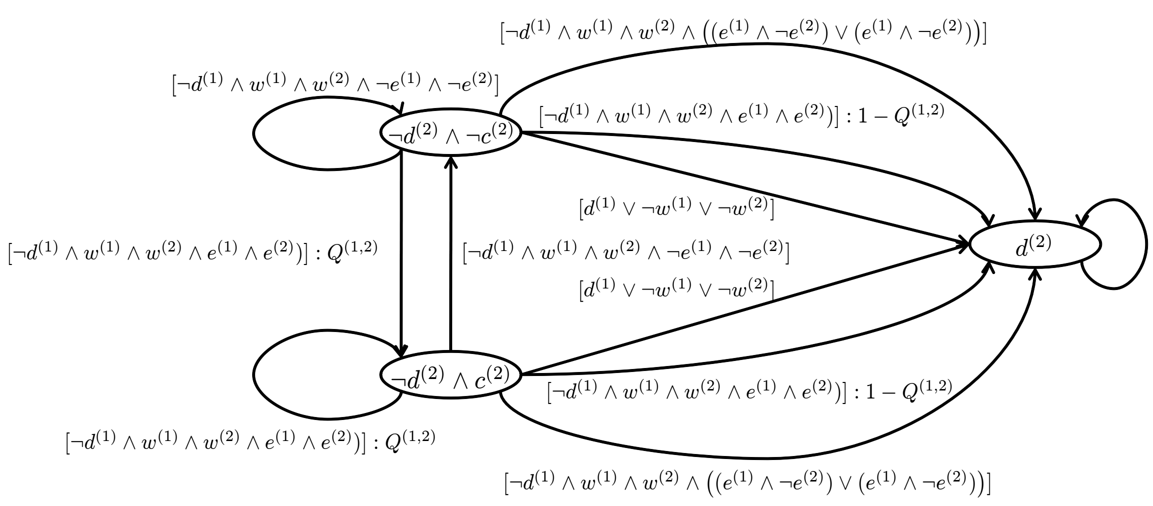
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge \big((e^{(1)} \wedge \neg e^{(2)}) \vee (e^{(1)} \wedge \neg e^{(2)})\big)]$
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge \neg e^{(1)} \wedge \neg e^{(2)}]$
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge e^{(1)} \wedge e^{(2)})] : 1 - Q^{(1,2)}$
$\neg d^{(2)} \wedge \neg c^{(2)}$
$[d^{(1)} \vee \neg w^{(1)} \vee \neg w^{(2)}]$
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge e^{(1)} \wedge e^{(2)})] : Q^{(1,2)}$
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge \neg e^{(1)} \wedge \neg e^{(2)}]$
$d^{(2)}$
$[d^{(1)} \vee \neg w^{(1)} \vee \neg w^{(2)}]$
$\neg d^{(2)} \wedge c^{(2)}$
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge e^{(1)} \wedge e^{(2)})] : 1 - Q^{(1,2)}$
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge e^{(1)} \wedge e^{(2)})] : Q^{(1,2)}$
$[\neg d^{(1)} \wedge w^{(1)} \wedge w^{(2)} \wedge \big((e^{(1)} \wedge \neg e^{(2)}) \vee (e^{(1)} \wedge \neg e^{(2)})\big)]$

Fig. 7. Probabilistic automaton for the Monitoring and Reconfiguration Sub-Module in layer 2 (MRSM2).

MRSM2 uses the output from L^1 to monitor L^2. If both layers are deemed as working by their voting mechanism, then MRSM2 compares the outputs from both layers. Since L^2 subsumes L^1 in terms of capability, the output from L^2 used for comparison is a subset of the output from L^2 with respect to the capability of L^1. There are several scenarios:

1. If either L^1 is not functioning (disabled by MRSM1 or deemed as *not* working by its voting mechanism) or L^2 is deemed as *not* working, L^2 is disabled. Note that we are not considering the recovery of a layer in this paper. Instead, recovery is implemented by shifting to a lower layer. Thus, once L^2 is disabled, it is never reinstated;
2. Otherwise, MRSM2 compares the outputs of L^1 and L^2 with respect to the functional capability level of L^1;
   (a) If two outputs are different, then MRSM2 disables L^2. Since MRSM2 cannot tell which output is right, it assumes L^2 as incorrect and hence disables it. The causes for two different outputs could be (1) one of the layers produces an incorrect output but the other produces the correct one (i.e., $(\neg e^{(1)} \wedge e^{(2)}) \vee (e^{(1)} \wedge \neg e^{(2)})$), or (2) both layers produce incorrect results and the results are different (with probability $1 - Q^{(1,2)}$).
   (b) If two outputs are same, then L^2 continues to function. The causes for the same outputs could be (1) both layers produce the correct output, or (2) both layers produce the same incorrect results (with probability $Q^{(1,2)}$). In the latter case the output of L^2 is compromised ($c^{(2)} = true$).

5.1 Error distribution and model parameters

In our performance analysis, we consider different cases of error distributions. We let N denote the size of the sample space for outputs from layers. That is, N is the number of data points for an output. We also assume that there is only one data point deemed to be correct at a given time for a given output. As stated before, $P^{(j)}_{1|k}$ denotes the probability that only a single variant among k working variants on layer j produces the correct result. In order to address dissimilarity of variants and the impact of their independence of faults (or lack thereof) we will introduce $s^{(j)}_k$ and $d^{(j)}_k$ as the *correct* and *error* similarity coefficients below.

The possibility that all the k variants on layer j produce the correct result, denoted as $P^{(j)}_{k|k}$, is $P^{(j)}_{k|k} = (P^{(j)}_{1|k})^k \cdot s^{(j)}_k$, where $1 \le s^{(j)}_k \le (P^{(j)}_{1|k})^{1-k}$. The possibility that all the k working variants on layer j produce the exactly same erroneous result, denoted as $Q^{(j)}_{k|k}$, is $Q^{(j)}_{k|k} = \frac{(1-P^{(j)}_{1|k})^k}{(N-1)^{k-1}} \cdot d^{(j)}_k$, where $1 \le d^{(j)}_k \le \frac{(N-1)^{k-1} \cdot \left(1-(P^{(j)}_{1|k})^k \cdot s^{(j)}_k\right)}{(1-P^{(j)}_{1|k})^k}$. To see the meaning of $s^{(j)}_k$ and $d^{(j)}_k$, let's consider the following scenarios:

I. We consider the case in which the k variants are not real "variants" but exact duplicates of each other and thus behave identically. This constitutes a total lack of independence of faults, i.e., it is classic common-mode fault behavior. For this case we may define $s^{(j)}_k = (P^{(j)}_{1|k})^{1-k}$ and,

$$d^{(j)}_k = \frac{(N-1)^{k-1} \cdot \left(1 - (P^{(j)}_{1|k})^k \cdot s^{(j)}_k\right)}{(1 - P^{(j)}_{1|k})^k} = \frac{(N-1)^{k-1}}{(1 - P^{(j)}_{1|k})^{k-1}}$$

Therefore, we have,

$$P^{(j)}_{k|k} = (P^{(j)}_{1|k})^k \cdot s^{(j)}_k = P^{(j)}_{1|k}$$

and,

$$Q^{(j)}_{k|k} = \frac{(1-P^{(j)}_{1|k})^k}{(N-1)^{k-1}} \cdot d_{k,j} = \frac{(1-P^{(j)}_{1|k})^k}{(N-1)^{k-1}} \cdot \frac{(N-1)^{k-1}}{(1-P^{(j)}_{1|k})^{k-1}} = 1 - P^{(j)}_{1|k} = Q^{(j)}_{1|k}$$

That is, the probability of k variants producing the same result is reduced to the probability of a single variant producing the exactly same result.

II. Now we consider the other extreme, i.e., true independence of faults for the variants. Consider $s^{(j)}_k = 1$ and $d^{(j)}_k = 1$. Therefore we have,

$$P^{(j)}_{k|k} = (P^{(j)}_{1|k})^k \cdot s^{(j)}_k = (P^{(j)}_{1|k})^k$$

and,

$$Q^{(j)}_{k|k} = \frac{(1-P^{(j)}_{1|k})^k}{(N-1)^{k-1}} \cdot d^{(j)}_k = \frac{(1-P^{(j)}_{1|k})^k}{(N-1)^{k-1}}$$

The equation for $P^{(j)}_{k|k}$ indicates that k variants behave independently, so that the probability that all k variants produce the correct result is $P^{(j)}_{k|k} = (P^{(j)}_{1|k})^k$. The equation for $Q^{(j)}_{k|k}$ further assumes that errors are distributed uniformly among the sample space (i.e., data points), so the probability that a particular erroneous data point will be produced by a variant is $\frac{1-P^{(j)}_{1|k}}{N-1}$. Since there are $N-1$ erroneous data points and k variants, the probability that all the k variants produce the same erroneous result is $(N-1) \cdot \left(\frac{1-P^{(j)}_{1|k}}{N-1}\right)^k = \frac{(1-P^{(j)}_{1|k})^k}{(N-1)^{k-1}}$.

Based on the previous discussion one can see that $s^{(j)}_k$ and $d^{(j)}_k$ are introduced to describe the similarity of variants and its impact on error distribution. For an implementation using N-variant technology, $d^{(j)}_k$ shall be close to 1.

6 Computational Experiments

To assess the performance of our proposed N-variant architecture, we conducted computational experiments on two models. The first model is the generalized stochastic Petri-net as shown in Figure 3. The second model is the probabilistic automaton-based model described in Section 5. The two models leverage the benefits of the two different stochastic modeling techniques. The Petri-net model in Figure 3 is more intuitive and it gives a higher-level view of the N-variant architecture, but it lacks mechanisms such as the support for state and input variables. Furthermore, it does not support synchronized transitions. The

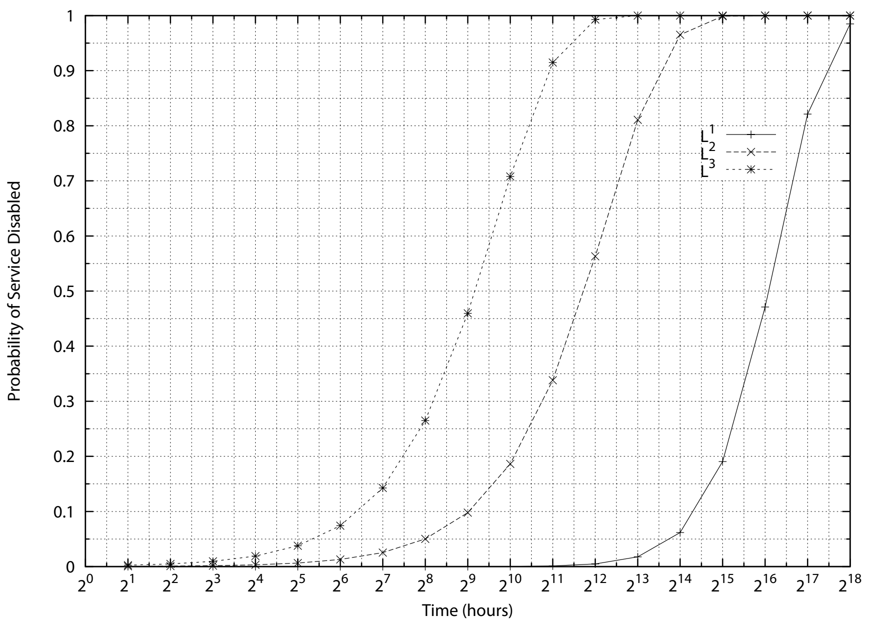

Fig. 8. Probability of services being disabled for the GSPN model.

extension of probabilistic finite automaton supports these features and we use them to model more complex mechanism such as voting algorithms inside layers. For example, in the Petri-net model a variant is marked as *not* working if its output is deemed incorrect. In more realistic scenarios, the voting mechanism does not have an oracle to tell whether the output is correct or not, so it compares all the working variants. Consider the two variants in layer L^2 of F_1 in Figure 2, the Petri-net model marks both variants *not* working if both variants fail. In a more realistic scenario, two failed variants may produce the same erroneous values and hence the voting mechanism marks both variants (incorrectly) as working. This scenario missed by a Petri-net model is essential for modeling information security since it describes a case in which layer L^2 is compromised (i.e., marked as *working* but producing incorrect output). Nevertheless, the benefits of our probabilistic automaton-based model come at a price: it contains more implementation details and hence it is less readable than the Petri-net model.

We use the Symbolic Hierarchical Automated Reliability/Performance Evaluator (SHARPE) [9] to analyze the Petri-net model and the probabilistic model checker PRISM [5] to analyze the probabilistic automaton-based model. PRISM translates a probabilistic automaton-based model to a discrete-time Markov Chain (DTMC) and checks it against a temporal property in the PCTL, a probabilistic extension of Computational Tree Logic (CTL). In our experiments, we measure the probability that layer L^k is disabled within n step. The property can be expressed in the PCTL formula: $P[F^{\leq n}(d^{(k)})]$. The bounded temporal

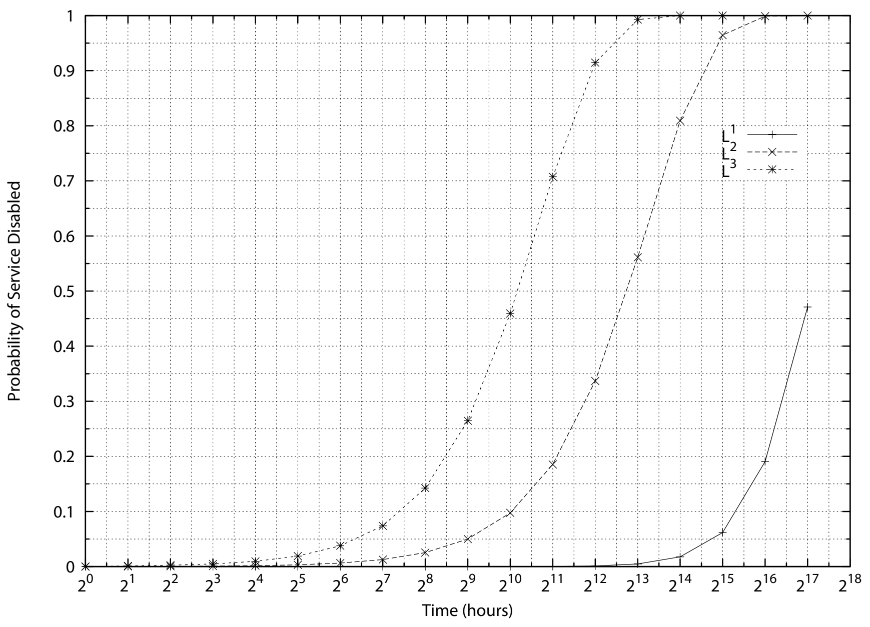

Fig. 9. Probability of services being disabled for the probabilistic automaton model.

operator $F^{\leq n}(f)$ describes the bounded eventuality that the state formula f becomes true within n steps, and $P\phi$ asks the model checker the probability that the path formula ϕ is held on all the executions. Interested readers may refer to [5] for the details of PCTL and PRISM.

Figure 8 shows the probability of L^1, L^2, and L^3 being disabled for the GSPN model. We assume that the failure rates of a variant in L^1, L^2, and L^3 is 10^{-5}, 10^{-4}, and 10^{-3} per hour respectively. The cross-monitoring detection rates C_1 and C_2 in Figure 3 are both 10^{-6}. Figures 9 shows the probability of L^1, L^2, and L^3 being disabled for the probabilistic automaton-based model, using the same failure rates and $N = 10$ as the size of the sample space. Figures 9 shows a slightly gain over Figure 8 for availability. That is because the probabilistic automaton-based model considers the case that variants may produce the same erroneous result while the Petri-net model does not.

7 Conclusion

This research analyzes the reliability, security, and performance of an N-variant layered software architecture for survivable systems with multi-core hardware. The primary goal of our proposed software architecture is to harness the computational power of under-utilized cores for improving reliability and survivability of critical applications. The layered architecture design supports on-the-fly reconfiguration and allows the use of existing techniques such as those described in [3, 4] at each layer. To leverage the benefits of two different stochastic analysis techniques, we propose reliability model using GSPN and a probabilistic automaton model for use with probabilistic model checking. Whereas the GSPN model gives a system designer an intuitive tool to outline the layered redundancy scheme suitable for the required reliability specification, a probabilistic automaton-based model is closer to the real behavior of the layered architecture and it is of great value for an implementation. Modeling with two different techniques increases the confidence in the models and simulations. The computational experiments confirm that the two independently derived models are compatible and it demonstrated that the pure theoretical model, i.e, the GSPN model, is more conservative since it cannot account for value related false positives or false negatives, which were caught by the probabilistic automaton-based model. Lastly, the probabilistic automaton-based model also addresses the degree of independence, which ranges from total independence of faults to common mode behavior, by introducing *correct* and *error* similarity coefficients.

References

1. A. Avizienis , *The Methodology of N-version Programming*, Software Fault Tolerance, edited by M. Lyu, John Wiley & Sons, 1995.
2. M.H. Azadmanesh, and R.M. Kieckhafer, *Exploiting Omissive Faults in Synchronous Approximate Agreement*, IEEE Trans. Computers, 49(10):1031-1042, 2000.
3. B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. *N-Variant Systems: A Secretless Framework for Security through Diversity*, 15th USENIX Security Symp., Vancouver, Aug. 2006
4. C.M. Jeffery and J.O. Figueiredo. *Towards Byzantine Fault Tolerance in Many-core Computing Platforms*, 13th IEEE International Symposium on Pacific Rim Dependable Computing, 2007.
5. M. Kwiatkowska, G. Norman, and D. Parker. *Probabilistic symbolic model checking with PRISM: A hybrid approach. International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128-142, 2004.
6. R. Laddaga, P. Robertson, and H. Shrobe. *Introduction to Self-adaptive Software: Applications*, Proc. 2nd Workshop on Self Adaptive Software, LNCS 2614, pp 275-283, May, 2001.
7. M. O Rabin. *Probabilistic Automata*, Information and Control 6(3):230-245, 1963.
8. B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz. *Multi-Variant Program Execution: Using Multi-Core Systems to Defuse Buffer-Overflow Vulnerabilities*, Proc. CISIS'08, pp. 843-848, 2008.
9. R. Sahner, K. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems*, Kluwer Academic Publishers, 1996.
10. Li Tan, and Axel Krings, *A Hierarchical Formal Framework for Adaptive N-variant Programs in Multi-core Systems*, Proc. 9th Int'l Workshop on Assurance in Distributed Systems and Networks (ADSN 2010), 2010.
11. P. Thambidurai, and Y.-K. Park, *Interactive Consistency with Multiple Failure Modes*, Proc. 7th Symp. on Reliable Distributed Systems, pp. 93-100, 1988.
12. A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, *Security through Redundant Data Diversity*, Proc. DSN'08, 2008.