Resilient Multi-core Systems:
A Hierarchical Formal Model for N-variant Executions

Axel Krings[1]   Li Tan[2]   Clinton Jeffery[1]   Robert Rinker[1]

[1]University of Idaho
Moscow, ID 83844-1010
{krings,jeffery,rinker}@cs.uidaho.edu

[2]Washington State University
Richland, WA 99354-1641
litan@wsu.edu

ABSTRACT

This research presents a hierarchical formal model capable of providing adjustable levels of service and quality of assurance, which is especially suitable for multi-core processor systems. The multi-layered architecture supports multiple levels of fault detection, masking, and dynamic load balancing. Unlike traditional fault-tolerant architectures that treat service requirements uniformly, each layer of the assured architecture implements a different level of services and information assurances. The system achieves load balancing by moving between layers of different complexity. Functionalities at different layers range from essential services necessary to satisfy the most stringent requirements for information assurance and system survivability at the lowest layer, to increasingly sophisticated functionalities with extended capabilities and complexity at higher layers. Low-layer functionalities can be used to monitor the behavior of high-layer functionalities.

At each layer of the assured architecture, N-variant implementations make efficient use of multi-core hardware. The degree of the introduced redundancy in each layer determines the mix of faults that can be tolerated. The use of hybrid fault models allows us to consider fault types ranging from benign faults to Byzantine faults. Our framework extends recent work in N-variant systems for intrusion detection, which are demonstrated to be special cases. Furthermore, it allows the movement in a tradeoff space between (1) the levels of assurance provided at different layers, (2) the levels of redundancy used at specific layers, which determine the fault types that can be tolerated, and (3) the desired run-time overhead.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; D.4.6 [**Security and Protection**]: Invasive software; D.4.5 [**Reliability**]: Fault-tolerance; B.1.3 [**Control Structure Reliability, Testing, and Fault-Tolerance**]: Redundant design

General Terms

Reliability, Security, Survivability, Multi-core, N-variant

Keywords

N-variant Execution, Survivability, Resilience, Security, Fault Models, Adaptability

1. INTRODUCTION

Most new general purpose computers incorporate dual or quad-core processors and higher numbers of cores are on the horizon. 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.

Redundant executions to benefit reliability have been extensively used in fault-tolerant systems design, where the evolution of redundancy schemes has gone from homogeneous redundancy to heterogeneous systems. Heterogeneous redundancy implies that dissimilar functionalities which are functionally equivalent are executed under the assumption that this will reduce or eliminate common mode faults. Dissimilarity is typically discussed in the context of N-version programming dating back to the late 70s [1] and N-variant software. In N-version programming it is assumed that several software development groups derive programs based on the same specification in isolation. The expectation is that this helps to reduce or eliminate common mode faults. An approach inspired by N-version software is N-variant or multi-variant software, where different variants are generated in a more automated fashion. Again, the expectation is that a fault affecting one variant will not affect another (in the same way). In both cases a fault is detected if a difference is detected between outputs generated by two versions or variants.

The availability of unused or underutilized cores has been exploited to increase reliability, security and survivability. Redundant executions in multi-core systems has been considered in [4] in the context of transient faults, where the number of replicas determined the degree of fault tolerance. Idle resources have also been used to increase security. For example, in [3] a set of automatically diversified variants execute on the same inputs. Any difference in referencing memory of different variants can be observed and used to detect the execution of injected code. Similarly, multi-variate program execution is used in [7] to defuse buffer-overflow vulnerabilities. Variants use different directions in memory allocation so that buffer overflows "crash" into different

neighboring memory, which can be detected. The concept of N-variance can also be extended to include the representation of data, as was shown in [6].

The common threat in the previous research is that multiple variants execute in order to allow detection or masking of faults. Multiple executions introduce of course significant overhead. In fact, N-folding the amount of work to be done is even augmented with the overhead of managing the different variants. However, given the availability of slack-time in cores this overhead has the potential to be reasonably absorbed, i.e., if variants can execute on idle cores, then this overhead can be largely absorbed. What remains is the overhead induced by the framework that implements redundancy management, i.e., the mechanisms that distribute inputs to different variants and the treatment of the outputs.

What is the degree of redundancy required in the above research? In general this depends on the type and number of faults that are to be detected or masked and it is dictated by the fault models that are used.

1.1 Fault Model

The system is subjected to diverse fault types arising from diverse fault sources. The *fault sources*, describe the source of the fault, e.g., radiation, physical destruction, heat, or EMP, but may also result from malicious behavior or erroneous implementation of subcomponents or subsystems. The later encompasses those fault sources typically addressed in the context of the risks associated with the use of COTS components of third parties. However, the source of faults may originate in incorrect specifications or design.

The *fault types*, describe the types of faults considered such as crash faults, transient faults, intermittent faults, timing faults, value fault, or out-of-bound faults.

Lastly, *fault behavior*, (sometimes also called *fault mode*) describes the effects specific fault types have. Fault behavior is seen in the context of hybrid fault models (such as shown in [2, 5, 8]). An example is the hybrid fault model in [8] that considers three fault behaviors, i.e., *benign* faults, which are globally diagnosable; *symmetric* faults, which imply that values are wrong, but equally perceived by all components that receive the values; and *asymmetric* faults, which have no assumption on the fault behavior. The latter is often called Byzantine fault. Within the context of this research we are mostly concerned with the impact of faults on the fault behavior, rather than fault sources or types. For example, a buffer overflow may result in observable differences in memory management. This in turn can lead to fault detection and/or correction.

1.2 Background and Motivation

The previously discussed research uses N-variant executions in order to detect transient faults [4] and security related faults [3, 6, 7]. The different executions are considered to be functionally equivalent. For example, in [4] the replicas are managed dynamically by a hypervisor layer (a virtual machine monitor) inserted between the hardware and the operating system. The redundant functionalities execute on replica partitions, where the number of partitions is dictated by the fault model considered. In [7] multi-variat executions have very high probability of exposing buffer overflows, e.g., as would be experienced during a buffer overflow attack. Here the dissimilarity is mainly affecting the way memory is allocated. Again the functionalities do not differ with respect to their functional specifications. The same holds for the work in [3]. In fact the application of the principle of N-variant execution is based on functional equivalence of the executions.

The research presented here departs from this equivalency assumption. Whereas we still see the system as being composed of functionalities, we assume that these functionalities may have different levels of functional capabilities. Intuitively, by applying occam's raiser we make the assumption that lower levels of functionality (and thus capability) will ultimately result in lower probability of failure, as will be described in the context of Figure 1 in the next section.

2. SPECIFICATION MODEL AND ARCHITECTURE DESIGN

Our approach starts with a specification model for specifying levels of functional capabilities. We then propose a layered adaptive architecture design to realize the multilevel functional capability model. Finally we introduce a N-variant-based approach to implement the architecture design.

2.1 Multi-Level Functional Capability Model

We propose a formal model to specify functionality with adjustable levels of capability. The model, which we call Multi-Level Functional Capability Model (MLFCM), attaches each functionality to layers of capability. In our approach, MLFCM is used as part of requirement specification. During requirement elicitation, a development team works with stake holders of a project to identify not just functionalities, but also capability levels for each functionality. These capability levels specify how the system shall scale down its service in case of faults or under attacks.
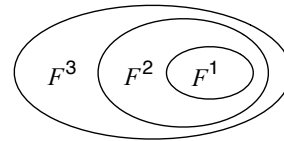
Figure 1: Multi-level functional capability model for functionality F

Assume the view that a system is comprised of functionalities F. Figure 1 shows an example of the MLFCM for functionality F. The requirements for F define three levels of capabilities: F^1 defines the set of core operations that are mission-critical, F^2 includes F^1 and some non-critical but value-added operations, and F^3 adds some more value-added operations. We write $F^1 \preceq F^2 \preceq F^3$, where $\preceq$ is a preorder that induces capability levels. The semantics of $\preceq$ is defined and interpreted based on application context. For instance, in a transaction-based asynchronous system, a functionality F can be specified as a set of sequences of operations $T(F)$ (in requirement elicitation, often referred as scenarios or flow of events). We represent a sequence of operations as $\langle (p_0(I_0), O_0), (p_1(I_1), O_1) \cdots \rangle$, where p_i is the operation at step i, I_i is the input data set, and O_i is the output dataset. By default, $T(F)$ also includes a null sequence τ. In such a system, we can define that $F^j \prec F^{j+1}$ if and only if $T(F^j) \sqsubseteq T(F^{j+1})$, where the piecewise inclusion relation $\sqsubseteq$ is defined as follows:

(i) $T(F^j) \subseteq T(F^{j+1})$; and,

(ii) For each $\langle p_0(I_0), O_0), (p_1(I_1), O_1), \cdots\rangle \in T(F^i)$, there is a sequence of operations $(p'_0(I'_0), O'_0), (p'_1(I'_1), O'_1), \cdots$ and a non-decreasing function g such that $p_i = p'_{g(i)}$, $I_i \subseteq I'_{g(i)}$, and $O_i \subseteq O'_{g(i)}$.

Note that (i) states that every sequence of operations defined at capability F^j shall also be included at capability F^{j+1}. Furthermore, (ii) states that each sequence of operations in F^{j+1} extends a sequence of operations in F^j. Note that (ii) doesn't prohibit the introduction of a completely new sequence of operations in F^{j+1}. In such a case, the new sequence of operations can be seen as an extension of the default null sequence $\tau \in P(F^{j+1})$.

As an example, consider a multi-level secured record-keeping system $\mathcal{S}$ in which each record $d = (d^1, d^2)$ contains two sets of data. Set d^1 contains mission-critical data and d^2 is a set of non-mission-critical but value-added data. A functionality F_r of system $\mathcal{S}$ is that a registered user can retrieve data. For each registered user u, $\mathcal{S}$ stores his public key K^u_{pub} and the user keeps his private key K^u_{prv}. F_r is defined with two levels of capabilities. At level F^1_r, a register user can retrieve the mission-critical data associated with a record. $T(F^1_r)$ contains the following sequence of operations t^1:

$(encry_u(K^u_{prv}, \text{``get\_id''}), req)(sent_{u,s}(\{u, req\}), ack_s)$
$(decry_s(K^u_{pub}, req), \text{``get\_id''})(read_s(id), \{d^1, d^2\})$
$(encry_s(K^u_{pub}, d^1), e^1)(send_{s,u}(e^1), ack_u)$
$(decry_u(K^u_{prv}, e^1), d^1)$

At level F^2_r, a register user can retrieve the mission-critical data as well as value-added data. $T(F^2_r)$ contains a sequence of operations t^2 as follows:

$(encry_u(K^u_{prv}, \text{``get\_id''}), req)(sent_{u,s}(\{u, req\}), ack_s)$
$(decry_s(K^u_{pub}, req), \text{``get\_id''})(read_s(id), \{d^1, d^2\})$
$(encry_s(K^u_{pub}, \{d^1, d^2\}), \{e^1, e^2\})(send_{s,u}(\{e^1, e^2\}), ack_u)$
$(decry_u(K^u_{prv}, \{e^1, e^2\}), \{d^1, d^2\})$

It is straightforward to see that $T(F^1_r) \sqsubseteq T(F^2_r)$, and hence $F^1_r \preceq F^2_r$.

The purpose of MLFCM is to specify not just functional requirement, but also requirements for reconfiguration and adaptiveness. It has two distinctive 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 capability levels defined in MLFCMs.

Second, the definition of capability levels also facilitates reconfigurable design. For example, consider the piece-wise inclusion relation $\sqsubseteq$ we proposed for transaction-based systems. It requires that system behavior at a higher capability level shall be an extension of behavior at a lower capability level. Hence, we can use behavior at a lower capability level as a reference for monitoring behavior at the higher capability level. Therefore, an implementation for capability levels provides a path for a system to scale back itself. Next, we will discuss a architecture design that facilitates reconfiguration as specified by a MLFCM.

2.2 Layered N-variant Architecture

To realize the reconfiguration requirements specified in MLFCM, we propose a layered adaptive architecture design similar to the N-variant techniques in [3, 7]. Figure 2 shows an example of the adaptive N-variant architecture. The architecture has a two-dimension layered structure. Each vertical layer realizes a capability level. It implements sequences of operations specified for its capability level. A layer may be disabled or nonexistent. The capability level of the entire functionality is decided by its highest enabled layer. Each layer is a collection of variants implementing its capability level. Variants are systematically diversified so that it is unlikely that that a common mode fault can occur, 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.
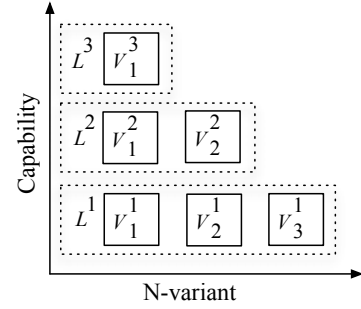

Figure 2: A layered adaptive N-variant architecture design for MLFCM in Figure 1

2.3 Adaptability and Reconfigurability

The layered adaptive N-variant architecture includes a monitoring and reconfiguration module. The module serves as a filter between layered N-variant functionalities and other sensitive components. The module intercepts calls to sensitive components. It uses sequences of operations in a layer to decide whether sequences of operations in the layer above are legitimate.

The layers in the adaptive N-variant architecture serve two purposes:

First, operations executed by a lower layer are used as a reference for monitoring at a higher layer. In section 2.1 capability levels in a MLFCM 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 a capability level F^i in MLFCM, 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 in L^{i+1}. The reason for this is because the lower layer L^i is realized using a higher degree of N-variant of simpler (and thus lower complexity) implementations, and hence L^i shall be considered more reliable than L^{i+1}. This argument is analogous to parallel reliability block diagrams. Recall that the unreliability of a parallel system is equal to the product of the unreliabilities of all parallel components. Here the number of components (degree of N-variant) at layer L^i is

larger than that of L^{i+1} and the fail-rates of the components at L^i are smaller that those at layer L^{i+1}, due to its simpler implementation. The result is a higher reliability at layer L^i.

Second, layers provide a reconfiguration plan in which a functionality can scale back its services in an orderly manner, thus providing 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 monitoring and reconfiguration module detects faults in a layer L^{i+1}, it disables L^{i+1} and the system automatically scales its capability to the level implemented by L^i. For completeness shake it should be noted that capabilities can not only be decreased, but also extended should the need arise, e.g., after recovery or repair.

As an example, consider the adaptive N-variant architecture design for the multi-level record-keeping system in Section 2.2. The design contains two layers, L_r^1 and L_r^2, that implement capability levels F_r^1 and F_r^2, respectively. Requests for retrieving data are sent to both layers for processing. If a request is from a registered user, L_r^1 executes t^1 and L_r^2 executes t^2. Note that t^1 and t^2 include a command $read_s(id)$ to access S's protected database subsystem. A monitoring and reconfiguration module intercepts all the calls to the database subsystem. Query $read_s(id)$ from L_r^2 is released if and only if a similar call $read_s(id)$ also received from L_r^1. If the monitoring and reconfiguration module does not receive a call $read_s(id)$ from L_r^1, it infers that layer L_r^2 has been compromised and hence it disables L_r^2. Consequently S scales its capability to F_r^1, which is implemented by L_r^1. This action constitutes a survivability feature with respect to the functionality F_r.

The layered adaptive N-variant architecture is designed for improving system survivability for mission-critical applications. By implementing functionalities in layers the capability of shifting to a lower layer upon detection of a fault provides a contingent plan that allows a system to scale back its services towards essential services as the result of faults or malicious attacks. The layered architecture is also designed to support information security. As can be seen in the example above, the monitoring and reconfiguration module ensures that sensitive information of d^1 will not be leaked by a higher layer, even the latter is compromised by an attacker. In the example, a fault detection at layers L^i and L^{i+1} resulted in an action by the monitor, blocking the release of d^1 and d^2.

2.4 Special Cases

The N-variant approaches described in [3, 4, 6, 7] are actually special cases and can be described within the architecture. First, it should be noted that all of these approaches 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 as described in subsection 2.3 is not supported.

The systems above can be described at one level. For example, the multi-variant scheme described in [7] operate on two variants at layer L_1, e.g., V_1^1 and V_2^1. The authors mainly focus on two variant memory referencing. The model in [3] can be specified similarly.

It should be noted however that fault masking is actually simpler than typically observed in redundant systems, e.g., k-of-N or NMR. For example, in a triple modular redundant systems two faulty modules can produce the same result and

consequently the TMR would vote on the incorrect value in the majority vote. Given the schemes described in [7] and [6] it is statistically very unlikely that two fault modules produce the same fault. This is very advantageous when trying to determine thresholds for non-faulty values and to reduce the degree of N-variants at each layer.

3. CONCLUSION

This research introduced a new way of looking at N-variant executions by defining a hierarchical formal model for N-variant executions especially suitable for systems based on modern multi-core architectures. One dimensions of the model N-variant implementations allow detection and masking of faults, dependent on the degree of variants and the fault model. At this layer we can take full advantage of many different solutions that have been presented in the literature associated with multi-variant systems. In the other dimension adaptation is introduced, which is an integral part of a survivable or resilient system. The basis for adaptation is the introduction of different levels of specification that result in layers of implementation, each of which in turn can be N-variant. Providing a hierarchy of capabilities, where lower level specifications are subsets of higher level specifications, has several advantages. First, lower level functionalities can effectively cross-monitor higher layers, which has positive implications for security and reliability. Second, during adaptation executions can be shifted to lower layers which increases survivability and resilience.

4. 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), pp. 1031-1042, Oct. 2000.

[3] B. Cox, et. al., N-Variant Systems A Secretless Framework for Security through Diversity, 15th USENIX Security Symposium, Vancouver, BC, August 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] F. J. Meyer, and D. K. Pradhan, *Consensus with Dual Failure Modes*, IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 2, pp. 214-222, April, 1991.

[6] A. Nguyen-Tuong, et. al., Security through Redundant Data Diversity, 38th IEEE/IFPF International Conference on Dependable Systems and Networks, Dependable Computing and Communications Symposium. Anchorage, June 2008.

[7] B. Salamat, et. al., Multi-Variant Program Execution: Using Multi-Core Systems to Defuse Buffer- Overflow Vulnerabilities, International Conference on Complex, Intelligent and Software Intensive Systems, Vol. 00, pp. 843-848, 2008.

[8] P. Thambidurai, and Y.-K. Park, *Interactive Consistency with Multiple Failure Modes*, Proc. 7th Symp. on Reliable Distributed Systems, Columbus, OH, pp. 93-100, Oct. 1988.