Definition and Evaluation of Adaptive Fault-Tolerant Architectures in a Distributed Computing Environment

J. Xu[1], *F. Di Giandomenico*[2], *A. Bondavalli*[3], *and S. Chiaradonna*[2]

[1] Dept. of Computing Science, Univ. of Newcastle upon Tyne, UK

[2] IEI/CNR, Pisa, Italy; [3] CNUCE/CNR, Pisa, Italy

Abstract

This paper discusses the issue of providing tolerance to both hardware and software faults in a distributed computing environment. We define several hybrid-fault-tolerant architectures that can co-exist and work simultaneously at the top of the supporting environment, and introduce a systematic method for evaluating their dependability, efficiency and response time.

Most existing studies assume that a fixed amount of hardware and system resources is bound statically to a given architecture. However, in a general-purpose distributed system multiple unrelated applications may compete for system resources such as processors, memories and communication devices, thereby exhibiting highly dynamic system characteristics. Our architectural solutions are directed at such systems and thus have an important concern with *adaptation* — adaptive execution of redundant programs so as to minimize hardware resource consumption and to shorten the response time, as much as possible, for a required level of fault tolerance. In comparison with static architectures, the analytical results under some realistic scenarios show that adaptive (i.e. dynamic) designs generally make more efficient use of available resources without compromising dependability. Dynamic architectures often have a longer worst-case response time, but in some application scenarios they may have a higher probability of making a timely response than static designs.

Key words — Adaptive architectures, dependability, distributed computing environments, efficiency, hardware and software fault tolerance, response time.

To appear in *Proc. Intl. Conf. on Safety and Reliability* (*ESREL'97*), Springer-Verlag, June 1997.

1 Introduction

General-purpose information-processing distributed systems are coming to life [1]. Such systems may contain computing nodes with very different characteristics which may be connected by different kinds of communication networks. In general, they are designed to support multiple different applications that may compete for both hardware and software resources. Our study in this paper will focus on this type of systems. Distribution in these systems is not a direct solution to dependability, and very high dependability cannot be achieved merely by simple backup or replication; the combined utilization of a wide range of *fault tolerance* techniques, intended to cope with the effects of both hardware and software *faults* and avert the occurrence of *failures* or at least to warn a user that *errors* have been introduced into the state of the system [2], is required.

Two major engineering approaches to the incorporation of fault tolerance into systems may be followed, representing two extremes of several possible choices. The *structured* approach, based on the system structuring principle widely applied in the design of general computing systems, partitions a system into different (and modular) abstraction layers, each performing its own tasks and providing services for the upper ones. In principle, the most suitable and profitable fault-tolerant techniques could be applied to different layers respectively. By isolating the faults within every single layer plus a set of well-defined failures of the underlying layers, the provision of fault tolerance in each layer is often relatively simple and easy to control. However, this approach may cause the loss of efficiency and performance: i) run-time costs introduced by fault tolerance in each layer are basically summed up resulting in a very high run-time overhead in a functioning system, especially in absence of faults, and ii) fault tolerance techniques used in different layers could overlap heavily leading to poor performance. The *integrated* approach is another choice. Redundancy may still be spread over layers but its management and the fault-tolerant actions are concentrated only in some (higher) layer. Faults in lower layers are propagated upwards and are masked, detected and treated by a previously selected higher layer. The overlap of fault-tolerant mechanisms and techniques could be controlled and minimized so as to improve efficiency and performance. However such integration would be relatively complex. (In fact, the two approaches are complementary in many aspects, in terms of their relative merits.)

In this paper we define our fault-tolerant architectures primarily following the integrated approach. Two layers, *software* and *system/hardware*, are distinguished: the software layer consists of multiple different (fault-tolerant, or non-fault-tolerant) applications that may use different techniques to achieve fault tolerance or other goals while the system/hardware layer corresponds to a distributed supporting environment that contains a set of computing nodes connected by a communication network. (A “hardware” processing node or component is composed here of both the hardware and the associated executive software providing necessary services for the execution of a special application in the software layer, and it may have disks organized as stable storage.) The effects of hardware failures may be masked by fault-tolerant mechanisms and schemes applied in the upper layer, but the distributed supporting system is responsible for hardware fault treatment, including fault diagnosis and the provision of continued service.

The objective of our work is twofold. First, based on a similar approach used in [3], we define several architectures by extending the existing software fault tolerance schemes to the treatment of both hardware and software faults (or called hybrid faults). The existing schemes are classified into two types: *static* strategies that always consume a fixed amount of resources and adaptive or *dynamic* strategies that use additional resources only when an error is detected. Our major concern is dynamic schemes in which redundancy is generally organized in a dynamic fashion in the hope that efficiency and performance will be improved. Two typical dynamic schemes are exploited — recovery blocks (RB) [4] and Self-Configuring Optimal Programming (SCOP) [5]. *N*-version programming (NVP) [6] and NVP with a tie-breaker (NVP-TB) [7] are chosen as two representatives of static schemes for the sake of comparison. Secondly, we introduce a method for analyzing the proposed architectures with respect to dependability, efficiency and response time aspects, and perform an evaluation in some realistic scenarios. The related analysis and evaluation are all based on the assumption of a distributed, varying environment, and so any of our results and conclusions is not necessarily true for other supporting environments (e.g. a system that only supports a single application).

There is some similar work on the definition and analysis of hardware and software fault-tolerant architectures [3][8][9]. In particular, Laprie, Arlat, Beounes and Kanoun [3] defined a set of hybrid-fault-tolerant architectures and evaluated three of them. Dugan and Lyu [10] used a combination of fault tree and Markov modeling to analyze three hybrid-fault-tolerant architectures derived respectively from RB, NVP and *N* self-checking programming [3]. Their architectures are however based on the assumption that the fixed hardware resources have been allocated previously or statically. The issue of the dynamic availability of hardware resources, software efficiency in general, was not discussed. Actually, static solutions cannot well match the characteristics of general-purpose distributed systems in which resources must be competed by multiple unrelated but concurrent service requests. When the requested resources are not always available in a varying environment, static architectures are often inefficient or infeasible. Of course, we are clearly aware that dynamic architectures require extra hardware resources as well as time expenditure when one or more errors are detected. This can lead to a relatively long response time, especially in the worst case. The maximum possible delay caused by errors must be analyzed and determined carefully.

The rest of the paper is organised as follows. In Section 2 several hybrid-fault-tolerant architectures are defined at the top of a distributed supporting environment. In Section 3 we evaluate the dependability of the architectures under consideration. Section 4 analyzes these architectures with respect to resource cost and response speed. Conclusions are in Section 5.

2 Distributed Architectural Solutions for Hybrid Fault Tolerance

This section defines a set of architectures for hardware and software fault tolerance under the assumption of a distributed supporting environment. Figure 1 shows a relationship between defined architectures, abstraction layers and the distributed system.

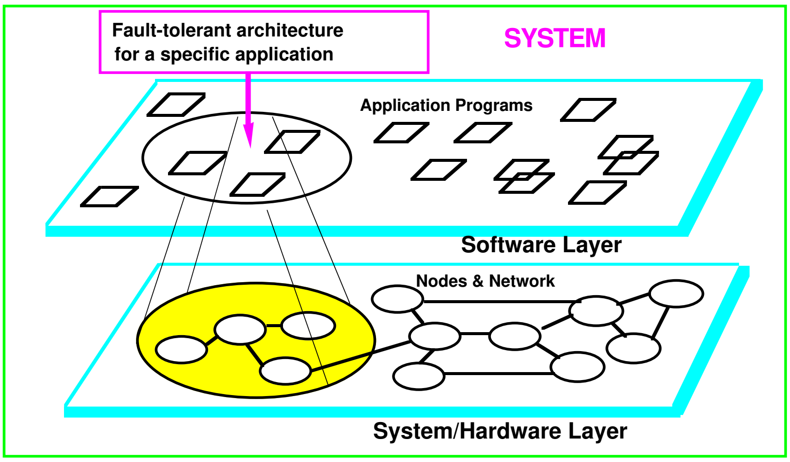

Figure 1 Architectures, abstraction layers and system.

Each of our architectural solutions is single-application-oriented. Conceptually an architecture must request processing resources from the supporting system upon invocation and return them to the system when the required computation terminates. During the computation, the architecture may apply for additional resources if necessary. For a given fault-tolerant application, an architecture contains a set of *software variants* designed independently (mainly for coping with residual design faults), an *adjudicator* [11] (e.g. an acceptance test or voter) for the selection of an acceptable output from the results of those variants, and a *control program* managing the execution of the variants and taking proper actions in response to the adjudicator output. The related programs and input/output data may be stored on the disks of some nodes. The architecture, or more precisely its control program, must guarantee that the state information and output results are produced dependably with a required probability so that they can be recorded correctly on stable storage. It is therefore its responsibility to support error recovery in the software layer as well as to report faults to the system.

In order to simplify the definitions and evaluation of the proposed architectural solutions, we introduce some assumptions that are common to all the architectures. An adjudicator is supposed to be replicated on all the hardware nodes supporting a specific architecture, but a selected node is responsible for taking a final decision from the local decisions and for producing outputs of the architecture. (As it would be short and simple, the final adjudication is assumed to be highly dependable.) Control programs are organised in a manner similar to the organization of adjudicators.

We characterize an architecture with respect to three aspects: 1) levels of fault tolerance, 2) hardware resource consumption, and 3) impacts on response time. An architecture is denoted by a group of multi-elements $\boldsymbol{X}(F, N, H_b, H_{max}, ...)$ where

- X indicates a specific architecture for hybrid fault tolerance, equivalent to the name of the selected scheme for software fault tolerance such as RB and NVP;
- F indicates the number of (hardware- and software-) faults to be tolerated and is further expressed by a detailed form: (f, i, j) in which f is the number of hybrid faults to be tolerated, i is the number of hardware faults to be tolerated assuming perfect software, and j is the number of software faults to be tolerated assuming perfect hardware;
- N is the number of application-specific software variants;
- H_b is the basic (minimum) number of hardware nodes an architecture needs to achieve the given level of hybrid fault tolerance F;
- H_{max} is the maximum (total) number of hardware nodes an architecture needs to achieve a given level of hybrid fault tolerance F when the worst fault situation occurs.

Due to the system-specific characteristics of response speeds (such as scheduling algorithms for resource allocation, and mechanisms for remote access), we will not incorporate specific labels for response time into the $\boldsymbol{X}(F, N, H_b, H_{max}, ...)$ expression though we may add such labels to the expression whenever the need arises. (An analysis of the response time will be presented in section 4.2.)

Although various architectural solutions could be constructed based on the chosen software fault tolerance schemes, we will restrict our interests to several particular instances with the form $\boldsymbol{X}((1, 2, 1), 2 \text{ or } 3, ...)$. Since realistic examples of implementing software fault tolerance are most based on two or three software variants [3], these instances have a more practical implication than other possible variations.

2.1 Dynamic Architectures

2.1.1 The SCOP Architecture

The **SCOP**((1, 2, 1), 3, 2, 4) architecture requests two basic hardware nodes and two further hardware nodes when a failure occurs in the software layer, as shown in Figure 2.

Three software variants are distributed on, or made accessible to, these basic and additional hardware components.

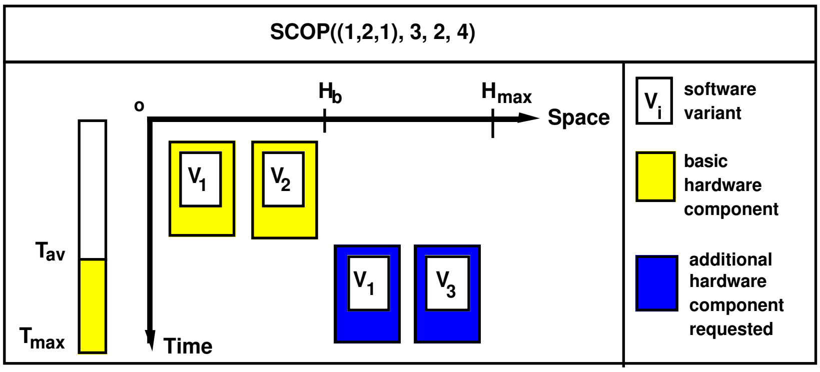

Figure 2 An instance of the SCOP architecture.

An execution of the **SCOP**((1, 2, 1), 3, 2, 4) architecture is divided into two phases. In the first phase, Variant 1 and Variant 2 run on two hardware components and the adjudicator compares their results. Consistent results are accepted immediately. Otherwise, the second phase begins and executes Variant 3 and Variant 1 on two additional hardware components. The adjudicator will decide in the end according to all the four results, seeking a 2-out-of-4 majority. This instance of SCOP can tolerate one (hardware or software) fault at least. If no software fault manifests itself during computation, up to two hardware faults will be tolerated. Normally two different variants are executed in parallel on only two hardware processing nodes. In the presence of faults, the third variant will be executed in parallel with one of the other variants on two additional nodes requested, leading to a heavy increase in response time.

2.1.2 The RB-Type Architecture

The primary variant V_1 in the **RB**((1, 2, 1), 2, 2, 3) architecture is executed on two hardware components (see Figure 3), and the results produced by the replicated variants are compared. If they agree, acceptance tests are applied to them. This agreeing result will be released unless both acceptance tests reject it. In the last case, an additional hardware node will be requested and the variant V_2 will be executed on it.

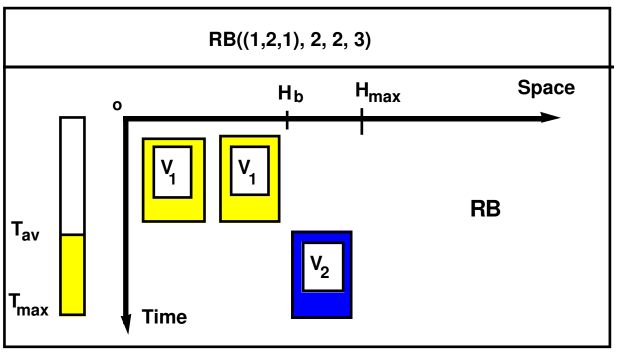

Figure 3 An instance of the RB architecture.

If the results produced in the first phase disagree, a diagnostic routine must be applied to the two hardware nodes employed. If only one of the two nodes is diagnosed as faulty, then the result produced by the variant running on the non-faulty node is released, otherwise an additional node is requested to execute the variant V_2. Note that, to tolerate the hypothesised faults, it is not conceptually necessary to check the result of V_2, although the acceptance test would still be used in practice to detect more erroneous situations. However, in order to make a fair comparison between the different architectures and not to disadvantage any single one with respect to a specific aspect (e.g. response time), we assume that the RB architecture takes the minimum measures just necessary to tolerate the hypothesised faults. This RB instance can tolerate at least one hardware or software fault. It is also highly efficient when no fault manifests itself during computation — the most likely situation. However, the application that uses this architectural solution must be prepared to accept a rare, but still possible heavy degradation — the response time would be much longer while performing self-diagnosis and executing the variant V_2 in order to mask the effect of two hardware faults.

2.2 Static Architectures

2.2.1 The NVP Architecture

The **NVP**((1, 2, 1), 3, 4, 4) instance requests four hardware nodes. Three software variants are executed in parallel on these nodes (according to the schema of Figure 4) and their results compared seeking a 2-out-of-4 majority. This architecture can tolerate any one hybrid fault. If no software fault manifests itself during computation, two hardware faults can be tolerated. The response time is guaranteed by executing the variants within a single phase, but it may be affected negatively by the time of requesting four hardware components and that of possible remote access. In the worst case that its request for hardware components cannot be fully met, this instance will not work.

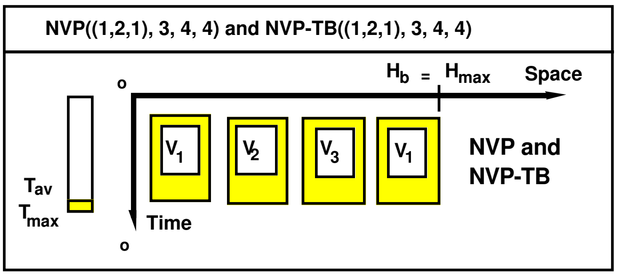

Figure 4 Instances of NVP and NVP-TB architectures.

2.2.2 The NVP-TB Architecture

The NVP-TB approach has been recently introduced [7] with the aim of enhancing performance of basic NVP by modifying the operational usage of the NVP computational redundancy. To obtain tolerance to one hybrid fault or two hardware faults, the **NVP-TB**((1, 2, 1), 3, 4, 4) architecture requests three software variants distributed on four hardware nodes as in the NVP architecture. The variants are executed in parallel, but as soon as two results by two different variants are produced, a first adjudication phase starts and, only if disagreeing results are observed, a second adjudication phase is executed involving all the four results, seeking a 2-out-of-4 majority. From the operational point of view, the NVP-TB instance differs from the SCOP instance (see Figure 2) for the fact that the three software variants plus the replicated one are always started on four required hardware nodes, independent of the results of the first adjudication phase. However, this architecture, although classified as a static one, does improve on NVP in performance, since in most cases only the first two results produced by two faster variants are needed to complete the computation.

3 Dependability Evaluation

In this section, a detailed dependability analysis of the architectures defined in Section 2 will be performed adopting a Markov approach. In [3, 12], the dependability analysis of hybrid architectures tolerating only one hardware or one consecutive software fault is conducted by first determining two models separately for the hardware and for the software and then combining the obtained results into a single model. We extend this kind of analysis by considering a different set of faults our architectures are to tolerate and by introducing a different model that allows to analyze both hardware and software faults in a combined framework. Our analysis starts from a set of special software failures that would lead to the failure of the whole architecture despite of the hardware conditions. Hardware failures are considered only when they affect the whole architecture alone or together with some software failures. (The term adjudicator is used here to represent both the adjudication function and the control program.)

Basic assumptions for our evaluation are as follows:

1) Failures of hardware processing nodes are independent; this is a reasonable assumption considering the nowdays well-established techniques for hardware design. The probability that correct software variants running on the failed hardware nodes produce the same incorrect outputs is assumed to be negligible;

2) Compensation among failures doesn't happen, neither between software variants, nor between variants and their adjudicator nor between hardware components and variants;

3) For dynamic architectures with multiple phases, the adjudicator exercised in more than one phase will show the same erroneous or correct behaviour throughout all the phases (from the software behaviour point of view);

4) Hardware faults are independent of software faults (and vice versa): a failure in an hardware component will cause an incorrect output of the software variant running on it, but will have no influence upon activating a fault in the variant itself;

5) To further simplify the analysis, failures of the underlying communication system are not addressed explictly (though a failure in the link connecting two nodes may be considered as a failure in the sending or receiving node.)

Table 1 shows the relevant types of failures of software and hardware components for the **SCOP**((1, 2, 1), 3, 2, 4) architecture.

Failure Types	**Prob.**
3 variants fail with consistent results	q_{3v}
2 variants fail with consistent results (the 3rd variant may fail with a different result)	q_{2v}
The adjudicator fails, selecting an erroneous result (given that an erroneous result has been produced)	q_{vd}
A variant fails, given that none of the above events happens	q_{iv}
Given the existence of a majority, the adjudicator fails to recognize it (without releasing any result)	q_d
An hardware node fails during an execution, affecting the variant and/or the adjudicator running on it	q_h

Table 1 Failure types and notation for SCOP((1, 2, 1), 3, 2, 4).

The detailed dependability model of the SCOP architecture is illustrated in Figure 5.

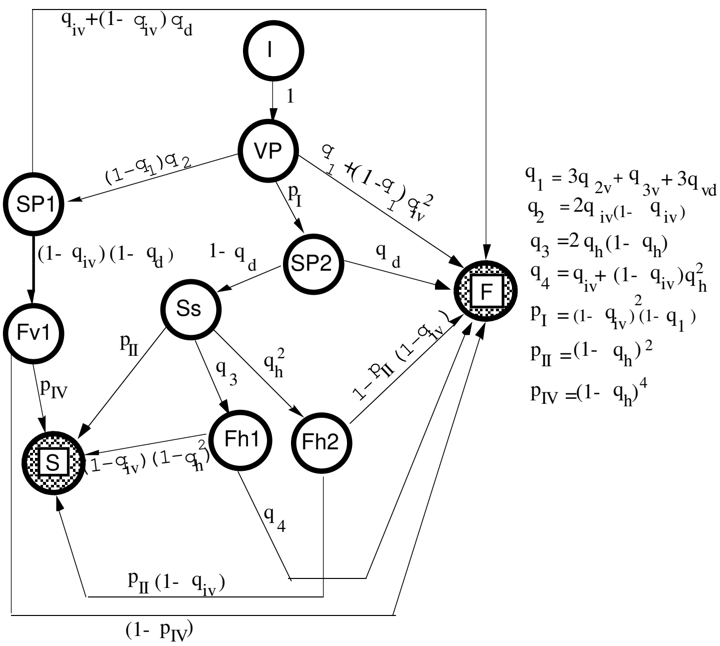

Figure 5 The dependability model for SCOP((1, 2, 1), 3, 2, 4).

This model is a slightly simplified one in which the states representing the execution of the second phase are introduced only when necessary in order to distinguish among different behaviours of the architecture. Table 2 briefly explains the meanings of the states and arcs in the figure.

I	initial state of an execution
F	failure of the architecture (absorbing state)
S	success of the architecture (absorbing state)
VP	two variants are executed on two nodes in the first phase; the arc from VP to F is labelled with the sum of the probabilities of the software failures causing the failure of the whole architecture without considering the hardware behaviour (i.e. independent failure of both the executed variants, and common mode failures between the variants and between the variants and the adjudicator)
SP1	one of the two variants executes correctly while the other fails, and the second phase is performed
SP2	the two variants execute correctly; 1) if the adjudicator fails to recognize the agreeing result, the second phase will be performed (unnecessarily); 2) if the adjudicator works correctly, the state representing the correct execution of the software components during the first phase will be reached
Fv1	just one variant fails after the first phase; success or failure of the whole execution depends on the hardware behaviour throughout the two phases
Ss	in the first phase, software components including the adjudicator execute correctly; according to the hardware behaviour the final state S is reached or the second phase executed (states Fh1 and Fh2)
Fh1	the second phase operates due to the failure of one hardware component during the first phase; success or failure of the whole execution depends on the behaviour of both hardware components and the software variants in the second phase
Fh2	the second phase operates due to the failure of two nodes; success or failure of the whole execution depends on the behaviour of both hardware components and the variants in the second phase

Table 2 Meanings of the states and arcs in Figure 5.

By using the set of intermediate parameters shown in the right side of Figure 5, the failure probability of the **SCOP**((1, 2, 1), 3, 2, 4) architecture is:

$$
\begin{aligned}
Q_{SCOP((1,2,1),3,2,4)} = \; & q_1+(1-q_1)q_{iv}{}^2+p_I(q_d+(1-q_d)(q_h{}^2(1-p_{II}(1-q_{iv}))+q_3q_4))+ \\
& +(1-q_1)q_2(q_{iv}+(1-q_{iv})q_d+(1-q_{iv})(1-q_d)(1-p_{IV}))
\end{aligned}
$$

Similar models are derived for the other architectures. Owing to the limitation of space, we omit details of these models and show the solutions directly. Given the similarities between the operational behaviour of SCOP and NVP-TB, exactly the same reliability models and hence the same reliability expressions are derived for the two architectures. Note that Table 1 can be applied to **NVP**((1, 2, 1), 3, 4, 4) as well, while Table 3 introduces the relevant types of failures of software and hardware components for the **RB**((1, 2, 1), 2, 2, 3) architecture. (Because of its extreme simplicity, i.e. comparison of two replicas, the failure of the comparator used by the RB architecture is assumed to be negligible.) By the use of intermediate parameters illustrated in Table 4, the derived expressions of the failure probability are:

$$
\begin{aligned}
Q_{NVP((1,2,1),3,4,4)} = \; & q_1+(1-q_1)q_2+3q_{iv}(1-q_1)(1-q_{iv})^2(q_d+(1-q_d)(1-p_{IV}))+ \\
& +(1-q_1)(1-q_{iv})^3(q_d+(1-q_d)q_I)
\end{aligned}
$$

$$
\begin{aligned}
Q_{RB((1,2,1),2,2,3)} = \; & q_1+(1-q_1)q_5+(1-q_1)q_p(1-q_a)(1-q_s)(1-P_{III})+ \\
& +(1-q_1)(1-q_p)(1-q_a)((q_h{}^2(1-c_{AT})^2+2q_h{}^2c_{AT}(1-c_{AT}))(q_s+(1-q_s)q_h)+q_h{}^2c_{AT}{}^2q_s)
\end{aligned}
$$

$$
Q_{NVP\text{-}TB((1,2,1),3,4,4)} = Q_{SCOP((1,2,1),3,2,4)}
$$

Failure Types	Prob.
Primary and secondary alternates fail with consistent results which pass AT	q_{psa}
Primary alternate fails but AT accepts its result	q_{pa}
Primary and secondary alternates fail with consistent results which don't pass AT	q_{ps}
Primary or secondary alternate fails independently, given the above events do not occurr	q_p, q_s
The acceptance test fails, rejecting a result, given the result is correct	q_a
A node fails during an execution, affecting the variant and/or the adjudicator running on it	q_h
An hardware node fails but affects only the AT running on it	c_{AT}

Table 3 Failure types and notation for RB((1, 2, 1), 2, 2, 3).

NVP intermediate parameters	RB intermediate parameters
$\mathbf{q_1}=3q_{2v}+q_{3v}+3q_{vd}$	$\mathbf{q_1}=q_{pa}+q_{psa}+q_{ps}$
$\mathbf{q_2}=3q_{iv}^2(1-q_{iv})+q_{iv}^3$	$\mathbf{q_5}=q_pq_a(1-q_s)+q_pq_s+(1-q_p)q_a$
$\mathbf{q_I}=q_h^4+4q_h^3(1-q_h)$	$\mathbf{p_{II}}=(1-q_h)^2$
$\mathbf{p_{IV}}=(1-q_h)^4$	$\mathbf{p_{III}}=(1-q_h)^3$

Table 4 Intermediate parameters for the NVP and RB architectures.

The derived expressions are too complex to allow a precise comparison among all the architectures. As an example, Figure 6 gives a plot of the functions representing failure probabilities of the four architectures under consideration. To produce the plot, some numerical values are chosen for the dependability parameters, as listed in Table 5.

Recovery Blocks	NVP, NVP-TB and SCOP
$q_{ps} = q_{pa} = q_{sa} = q_p \times 10^{-3}$ $q_{psa} = 10^{-10}$ $q_a = 2 \times 10^{-7}$ $q_p = q_s$: variable from 10^{-5} to 10^{-3} $q_h = 10^{-9}$ $c_{AT} = 10^{-3}$	$q_{2v} = q_{iv} \times 10^{-3}$ $q_{3v} = q_{vd} = 10^{-10}$ $q_d = 10^{-9}$ q_{iv}: variable from 10^{-5} to 10^{-3} $q_h = 10^{-9}$

Table 5 Values of the dependability parameters used in the example.

In Table 5, i) the acceptance test (AT) is assumed to have a higher probability of failure than the adjudicators used in NVP, NVP-TB and SCOP due to its complexity and dependence on specific applications; ii) a positive correlation is assumed between the failure of multiple variants resulting in identical or similar errors (and the same between the alternate and the AT in RB); iii) the probability of independent failure of variants varies between 10^{-5} and 10^{-3}; iv) the probability of hardware failure per execution is determined assuming that the probability of hardware failure is 10^{-4} per hour (a usual estimation provided by manufacturers) and the time duration of a single execution is about 50 milliseconds. It must be mentioned that these values chosen for the parameters, despite of certain practical implications, simply constitute a line in the space of all the possible combinations. This doesn't allow to derive any general conclusion about the dependability of the four architectures. However, the example seems to be quite consistent with some intuitive conclusions. Since the influence of the hardware failure upon an architecture is relatively small according to the set of parameters chosen, it is the software behaviour that makes major contributions to failure of the whole architecture. Previous work in the literature has showed, for example in [5], that there is no evidence that, in the general case, one of the RB, NVP and SCOP schemes is significantly better than the others from the dependability point of view, especially when comparable parameters values are used. This explains why the three curves appear very close in the figure although, we would remind the reader, the RB architecture needs perfect diagnostic routines.

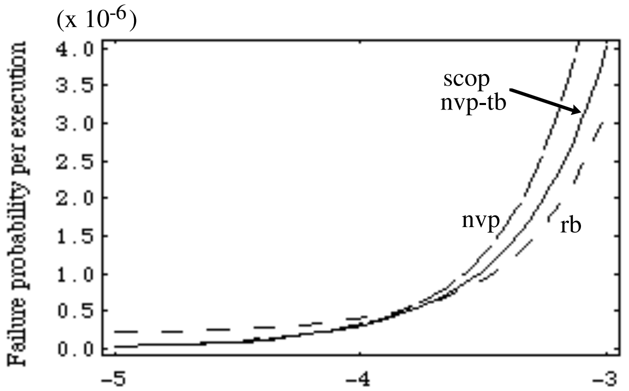

Figure 6 Plot of the probabilities of failure for the three architectures.

4 Resource Cost and Response Time

In this section the average resource consumption (i.e. the average number of hardware nodes required) and response time are estimated for each execution of a given architecture.

4.1 Average Resource Consumption

The SCOP architecture may require two phases. From the dependability evaluation of the architecture and using the same set of intermediate parameters, we obtain the probability that the architecture terminates at the end of the first phase:

$$p_{1SCOP} = p_I(1-q_d)p_{II} + (q_{2v} + q_{3v})(1-q_d)p_{II} + 2q_{vd}.$$

Then, the average resource consumption by the SCOP architecture in one execution is:

$$AV.RES_{SCOP} = 2 + 2(1 - p_{1SCOP}).$$

Similar to SCOP, the RB architecture may consist of up to two phases and, equivalently, its probability to stop at the end of the first phase (including the case where it is necessary to run diagnostic routines) is:

$$p_{1RB} = ((1-q_a)(1-q_p)(1-q_1) + q_{pa} + q_{psa})p_{II} + 2q_h(1-q_h)(1-c_{AT}) + \\ 2q_h(1-q_h)c_{AT}(q_{ps} + (1-q_1)(q_p + (1-q_p)(1-q_a))).$$

The average resource consumption by the RB architecture in one execution is $AV.RES_{RB} = 2 + (1 - p_{1RB})$.

NVP((1, 2, 1), 3, 4, 4) is not organised in phases — it executes all of its variants in parallel. Therefore it has a constant resource consumption that is equal to four, i.e., $AV.RES_{NVP} = 4$.

NVP-TB((1, 2, 1), 3, 4, 4), although it stops as soon as two equal results are produced by two different variants, generally executes all of its variants in parallel[1]. Thus, in most cases it has a resource consumption equal to four: $AV.RES_{NVP-TB} = 4$.

From this simple analysis, we can conclude that dynamic architectures have average resource consumption lower than static architectures. In particular, Figure 7 shows the plot of the average utilisation of processing nodes required by the SCOP and RB architectures during each execution as a function of the probability of independent failures.

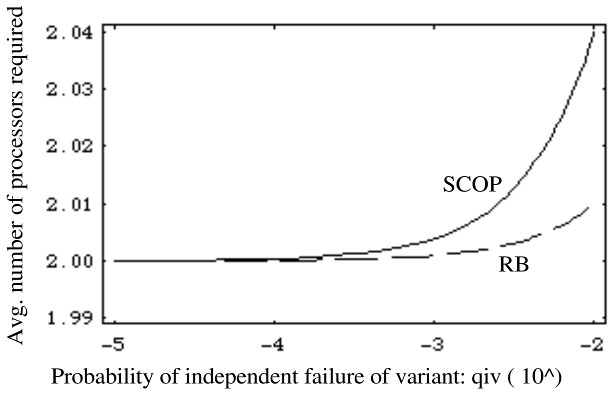

Figure 7 Average processing node utilisation vs. failure probability.

The same values already used in the previous section and reported in Table 5 are assigned to the dependability parameters. Again, the realism of such kind of evaluation depends on realistic values and ranges which must be derived for each individual realization. However, for most plausible values the probability that SCOP or RB stops at the end of the

---

1 Not all the variants are executed when the execution time of the variants is much shorter than the time necessary to require a processing node and the first two different variants produce equal results before the other variant(s) start(s) its(their) own execution. We will not deal with such a special case in our response time analysis, and assume the acquisition of all the necessary hardware nodes as a prerequisite for the execution of the software to be started.

first phase is very high. This means that the average hardware consumption requested by these dynamic architectures is almost constant and very close to the amount required to start with.

4.2 Response Time

The response time analysis is conducted in two different scenarios: 1) all the processing nodes are required from the supporting system, and the proper software is then loaded on them before execution takes place; and 2) basic processing nodes required by each architecture are supposed to be pre-allocated and thus immediately available once the execution is invoked and only additional components need to be requested from the supporting system when necessary. Although scenario 1) would be the most appropriate to general-purpose distributed systems which we focus upon in this paper, the alternative scenario 2) is considered in order to understand to which extent static architectures gain an advantage over dynamic ones in an application where their employment would be more appropriate.

To analyze response time we will follow the same approach used in [7, 13], but include times relative to hardware components. It is assumed that the time needed to obtain the processing nodes and to load the software is an independent and exponentially distributed random variable W_i with parameter λ_{wi}. The execution times of different combinations of variant/node pairs are also assumed to be independent and exponentially distributed random variables E_i with parameter λ_i, particularly E_d with parameter λ_d, for the adjudicator. Designating with Y_c the duration of an execution of the given architecture without considering any interruption of the execution because of a watchdog timer, we will derive the distribution of Y_c and its mean μ for the purpose of comparison. The probability p_{bt} that an execution violates a timing constraint τ set at each invocation of a given architecture (that is, Y_c exceeds τ) can provide further information.

4.2.1 Dynamic Acquisition of Processing Nodes

In this scenario, the necessary processing nodes are first required from the supporting system, and the proper software is then loaded on them before execution takes place. Let $Y_{w1} = \max\{W_1, W_2\}$ and $Y_{E1} = \max\{E_1, E_2\}$ denote the times necessary for obtaining two processing nodes and execute two variants in the first phase respectively. Similarly, concerning the second phase, $Y_{w2} = \max\{W_3, W_4\}$ and $Y_{E2} = \max\{E_3, E_4\}$. Therefore, the execution time Y_c for SCOP is:

$$Y_c = \begin{cases} Y_{c1} = Y_{W1} + Y_{E1} + Y_d = \max\{W_1, W_2\} + \max\{E_1, E_2\} + Y_d, & \text{with probability } p_{1SCOP} \\ Y_{c2} = Y_{W1} + Y_{E1} + Y_d + Y_{W2} + Y_{E2} + Y_d, & \text{with probability } (1 - p_{1SCOP}) \end{cases}$$

where p_{1SCOP} is the probability that the SCOP architecture stops at the end of the first phase.

Considering the time necessary for the RB instance to compare the results produced by the two copies of the primary variant and the time spent in running diagnostic routine on a hardware node, we introduce independent and exponentially distributed random variablesY_{com} with parameter λ_{com} and D_i with parameters λ_{Di}. $Y_D = \max\{D_1, D_2\}$ represents the time spent in running diagnostic routines. $Y_w = \max\{W_1, W_2\}$ indicates the time for obtaining the two basic nodes in the first phase, $Y_E = \max\{E_1, E_2\}$ the time for executing the primary on them and $Y_2 = \{W_3 + E_3\}$ the time for acquiring the third node and for running a variant on it. According to the specific operation of this architecture, four different situations may arise:

1) equal results are produced by the two copies of the primary and the AT is executed (with probability p_1);

2) after step 1), the third processing node is required to run the alternate variant (with probability p_2);

3) different results are produced by the two copies of the primary and the diagnostic routine is run (with probability p_3);

4) after step 3) a third node is required to run the alternate variant (with probability p_4).

The probabilities of these events may be derived from the dependability analysis. So, the execution time Y_c for RB becomes:

$$Y_c = Y_W + Y_E + Y_{com} + p_1(Y_d + p_2(Y_2)) + p_3(Y_D + p_4(Y_2)).$$

For the NVP architecture, $Y_w = \max\{W_1, W_2, W_3, W_4\}$ designates the time for obtaining the four nodes necessary to start the execution and $Y_E = \max\{E_1, E_2, E_3, E_4\}$ as the time for executing the variants. Then, the execution time Y_c for NVP is:

$$Y_c = Y_W + Y_E + Y_d = \max\{W_1, W_2, W_3, W_4\} + \max\{E_1, E_2, E_3, E_4\} + Y_d.$$

Similar to NVP, the NVP-TB architecture requires four processing nodes, but its execution can stop after the first two results are produced by two different variants (without having to wait for the slowest one) if they are found to be equal (which happens in most cases, with probability $p_{1NVP\text{-}TB} = p_{1SCOP}$, as derived from the dependability evaluation). Here, $Y_w = \max\{W_1, W_2, W_3, W_4\}$, Y_{F1} is the time for obtaining the first two results provided by two (generally faster) different variants. Y_{F2} the time for completing the execution of the slowest variant (which already started in parallel with the other variants), that is, $Y_{F2} = \max\{E_1, E_2, E_3, E_4\} - (Y_{F1} + Y_d))$.$\delta$ indicates the probability that the execution time of the slowest variant is equal or greater than the time for obtaining two results plus an adjudication phase, that is, the probability that $(\max\{E_1, E_2, E_3, E_4\} - Y_{F1}) \geq Y_d$. Notice that, since the architecture employs three variants plus a replica of one of these variants, Y_{F1} must be determined considering both the case where the time for producing two results by two different variants is given by the time for executing the two faster variants and the case where it is necessary to wait until the third result is produced, since the first two come from the two replicas of the same variant. Thus, Y_c for NVP-TB is:

$$Y_c = \begin{cases} Y_{c1} = Y_W + Y_{F1} + Y_d, & \text{with probability } p_{1NVP-TB} \\ Y_{c2} = \begin{cases} Y_W + Y_{F2} + Y_d, & \text{with probability } (1 - p_{1NVP-TB})\delta \\ Y_W + Y_{F1} + 2Y_d, & \text{with probability } (1 - p_{1NVP-TB})(1-\delta) \end{cases} \end{cases}$$

In order to give an estimation in a realistic situation, we now assign reasonable values to the timing parameters, as shown in Table 6. In this table, it is assumed that software variants have similar distributions for the execution time. This choice brings special benefit to the NVP architecture which otherwise would show worse response time due to the necessary synchronization with the slowest variant. The execution time of the comparator used by the RB architecture is lower than the execution time of the adjudicator used by the other two architectures since it is simpler to compare the results of replicas than of variants[2]. The probability that SCOP, NVP-TB or RB stops at some phase has been computed using the values reported in Table 5 and assigning the value 10^{-4} to q_{iv}.

RB	NVP, NVP-TB and SCOP
$\lambda_p = \lambda_s = \lambda_{Di} = 1/5$	$\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 1/5$
$\lambda_d = 1/3$	$\lambda_d = 2$
$\lambda_W = 1/50$, 1/5 and 2	$\lambda_W = 1/50$, 1/5 and 2
$\lambda_{com} = 4$	

Table 6 Values of the timing parameters used, measured in msecs^{-1}.

Using such setting, we show in Figure 8 the plots of the pdf of Y_c in the case of no timing constraints; in Table 7 a collection of the mean of Y_c and the probability P_{bt} (in presence of timing constraints) determined for some values of τ are given. When the time to acquire an hardware node is significantly longer than the execution time of a variant, dynamic architectures are better than static ones with respect to the average response time due to the lower number of nodes the former necessitates to start an execution (see Figures 8(a) and the first part of Table 7). In the case that the time for acquiring a node reduces to be equal or smaller than the execution time of a variant, the average response time is mainly determined by the execution time of variants and, with the parameters values chosen for this example, NVP-TB shows the best behaviour (see Figures 8(b) and (c) and second and third part of Table 7).

2 In principle, results produced by different software variants could be expressed in different formats or considered as equal although they differ for a small quantity and so on.

However, this superiority of NVP-TB (a static scheme with some dynamic characteristics) over the dynamic schemes shouldn't be simply generalized to a definitive superiority of static schemes over dynamic schemes (in fact, NVP is always worse than the others).

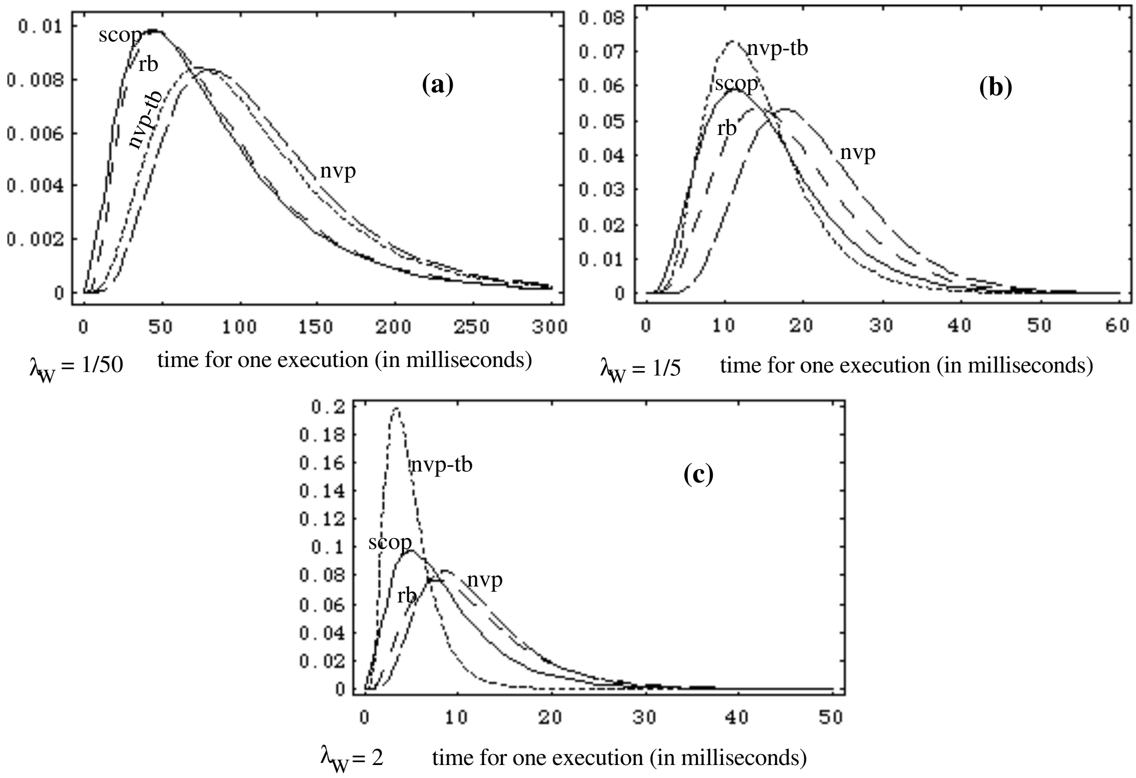

Figure 8 Distribution of Y_C under dynamic acquisition of hardware resources: (a) λ_W = 1/50, (b) λ_W = 1/5 and (c) λ_W = 2.

		NVP	NVP-TB	RB	SCOP
λ_W = 1/50	Mean of Yc (msecs)	115.0833	108.0020	85.7555	83.0166
	P_{bt} (τ = 30 msecs)	0.986965	0.971752	0.892773	0.870053
	P_{bt} (τ = 50 msecs)	0.910135	0.867497	0.704340	0.677491
	P_{bt} (τ = 70 msecs)	0.767781	0.710069	0.519826	0.496567
λ_W = 1/5	Mean of Yc (msecs)	21.3333	14.2515	18.2510	15.5031
	P_{bt} (τ = 30 msecs)	0.144972	0.025092	0.093209	0.054074
	P_{bt} (τ = 50 msecs)	0.005831	0.000466	0.003568	0.001805
	P_{bt} (τ = 70 msecs)	0.000166	0.000003	0.000100	0.000048
λ_W = 2	Mean of Yc (msecs)	11.9583	4.87640	11.5005	8.7517
	P_{bt} (τ = 30 msecs)	0.013599	0.000009	0.014728	0.006445
	P_{bt} (τ = 50 msecs)	0.000250	~0.000001	0.000279	0.000118
	P_{bt} (τ = 70 msecs)	0.000004	~0.000001	0.000005	0.000002

Table 7 Some results of the timing evaluation with dynamic acquisition of hardware resources.

The choice of identical (exponential) distributions for the execution time of the variants contributes to favouring NVP-TB (for example, the time for executing two variants among exactly two comes out to be significantly greater than executing two variants among four). Changing the distribution and/or the parameter λ_i, could lead to different results. Moreover, notice that NVP-TB, although classified as a static architecture because of its fixed requirement of four hardware to start an execution, has a dynamic operative behaviour. In this experiment, the RB architecture behaves worse than SCOP, but still no worse than NVP. (Again, the results shown by this evaluation example are not meant to lead to definitive conclusions about the behaviour of the four architectures examined, rather to show how such an evaluation can be made.)

4.2.2 Static Allocation of Basic Nodes with Dynamic Acquisition of Extra Nodes

In this scenario, only additional processing nodes other than basic ones need to be applied for when necessary. Thus, NVP and NVP-TB do not suffer from any delay due to the acquisition of hardware components. Based on the same approach as for the analysis conducted in the previous subsection, the new assumption may have impacts on the duration of an execution of the given architecture, Y_c, because it changes the calculation of the time W_i needed to obtain the processing nodes and load the software. Without showing all the details (easily derivable following the analysis in 4.2.1), the new expressions of Y_c for the architectures considered, adopting the same notation as in the previous analysis, are as follows.

SCOP: $$Y_c = \begin{cases} Y_{c1} = Y_{E1} + Y_d = \max\{E_1, E_2\} + Y_d, & \text{with probability } p_{1SCOP} \\ Y_{c2} = Y_{E1} + Y_d + Y_{W2} + Y_{E2} + Y_d, & \text{with probability } (1 - p_{1SCOP}) \end{cases}$$

RB: $Y_c = Y_E + Y_{com} + p_1(Y_d + p_2(Y_2)) + p_3(Y_D + p_4(Y_2))$.

NVP: $Y_c = Y_E + Y_d = \max\{E_1, E_2, E_3, E_4\} + Y_d$.

NVP-TB: $$Y_c = \begin{cases} Y_{c1} = Y_{F1} + Y_d, & \text{with probability } p_{1NVP-TB} \\ Y_{c2} = \begin{cases} Y_{F2} + Y_d, & \text{with probability } (1 - p_{1NVP-TB})\delta \\ Y_{F1} + 2Y_d, & \text{with probability } (1 - p_{1NVP-TB})(1 - \delta) \end{cases} \end{cases}$$

The evaluation here is based on the same values of the timing parameters already used in the previous evaluation and showed in Table 5. The static allocation of hardware resources greatly improves the response time of the architectures with respect to the previous case analyzed, especially those of the static architectures (see Figure 9 and Table 8). However, while NVP-TB shows better figures than those of dynamic schemes, NVP is still the worst architecture among those considered in most cases. Again, this result is highly dependent on the choice of the exponential distribution with identical parameter for the execution time of the variants and on the particular setting of values used.

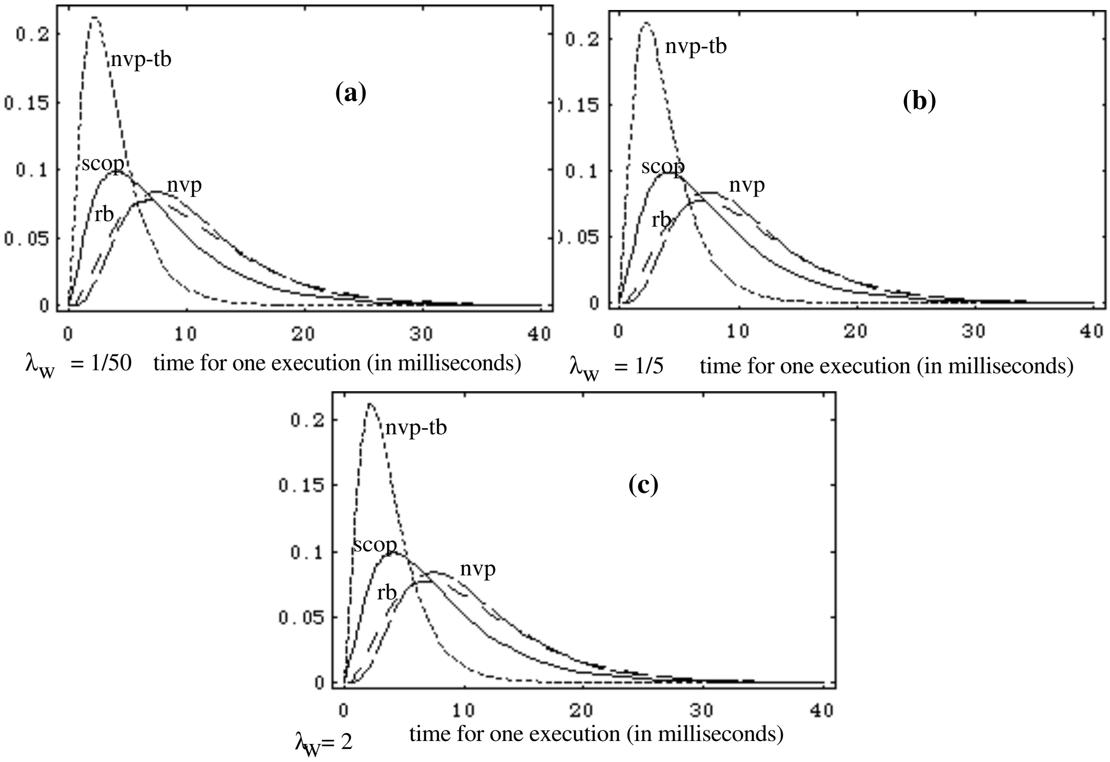

Figure 9 Distribution of Y_C under static allocation of hardware resources: (a) $\lambda_W = 1/50$, (b): $\lambda_W = 1/5$ and (c): $\lambda_W = 2$.

		NVP	NVP-TB	RB	SCOP
$\lambda_W = 1/50$	Mean of Yc (msecs)	10.916667	3.834733	10.755517	8.016616
	P_{bt} (τ = 30 msecs)	0.010971	0.000004	0.012709	0.005686
	P_{bt} (τ = 50 msecs)	0.000202	0.000006	0.000290	0.000252
	P_{bt} (τ = 70 msecs)	0.000004	0.000006	0.000039	0.000115
$\lambda_W = 1/5$	Mean of Yc (msecs)	10.916667	3.834733	10.751003	8.003103
	P_{bt} (τ = 30 msecs)	0.010971	0.000004	0.012649	0.005544
	P_{bt} (τ = 50 msecs)	0.000202	0.000006	0.000239	0.000103
	P_{bt} (τ = 70 msecs)	0.000004	0.000006	0.000004	0.000002
$\lambda_W = 2$	Mean of Yc (msecs)	10.916667	4.87640	10.750552	8.001752
	P_{bt} (τ = 30 msecs)	0.010971	0.000004	0.012640	0.005513
	P_{bt} (τ = 50 msecs)	0.000202	0.000006	0.000239	0.000101
	P_{bt} (τ = 70 msecs)	0.000004	0.000006	0.000004	0.000002

Table 8 Some results of the timing evaluation with static allocation of hardware resources.

Notice that the figures for NVP-TB and NVP are exactly the same, independent of the different values for λ_w; for SCOP and RB the differences are not so relevant, since the contribution of the time necessary to acquire extra hardware resources is very small (i.e. the second phase is rarely executed).

5 Conclusions and Final Remarks

Several architectures for tolerating both hardware and software faults have been defined at the top of a distributed supporting environment. These architectures are classified as dynamic or static, according to their ability to adapt the execution to the manifestation of faults so as to minimize resource consumption and shorten response time. A method for evaluating the defined architectures has been developed with respect to dependability, resource consumption and response time aspects, and an evaluation using some realistic parameter values has been performed. Although it is difficult to derive precise and definitive conclusions on dependability and efficiency in the general case due to many dynamic factors of the supporting environment under consideration, the analytical results show that dynamic architectures have better resource utilization than static ones, without heavily compromising dependability. Dynamic architectures often have a longer worst-case response time, but in certain application scenarios they may have a higher probability of making a timely response than static designs.

While our analysis and evaluation are obviously dependent on the basic assumptions made early, we have found that the assumptions, although quite realistic, or at least generally accepted in the literature, might be still questionable and thus limit the validity of the obtained results in the general case, including for example the assumption that the execution time of software components is independently and exponentially distributed.

Further studies are necessary for understanding what special features an architecture must have and what variations of the architectures defined in this paper could be developed so as to fit better a concrete application. Note that an architecture may behave better than another under a certain aspect, but in an opposite way in other aspects (for example, in our analysis NVP-TB behaves better than SCOP under response time, but worse than it under resource consumption when considering the scenario of static resource allocation). Thus, another research direction would be to investigate how to define and analyze different measures in order to find a good compromise as to which an engineering decision among different design alternates could be made.

Acknowledgement

This work was supported by the CEC-sponsored ESPRIT Basic Research Actions 3092 and 6362 on Predictably Dependable Computing Systems (PDCS and PDCS2). Jie Xu is also supported by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa).

References

[1] L. Svobodova, "Attaining Resilience in Distributed Systems," in *Dependability of Resilient Computers*, (T. Anderson, editor,) Oxford: BSP Professional Books, 1990, pp. 98-124.

[2] T. Anderson (ed.), *Resilient Computing Systems*, London, UK: Collins Professional and Technical Books, 1985.

[3] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," *IEEE Computer*, vol. 23, pp. 39-51, (Special Issue on Fault Tolerant Systems) 1990.

[4] B. Randell, "System Structure for Software Fault Tolerance," *IEEE TSE*, vol. SE-1, pp. 220-232, 1975.

[5] A. Bondavalli, F. Di Giandomenico and J. Xu, "Cost-Effective and Flexible Scheme for Software Fault Tolerance," *Journal of Computer Systems Science and Engineering*, vol. 8, pp. 234-244, CRL Publishing Ltd. 1993.

[6] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Program Execution," in *Proc. COMPSAC 77*, 1977, pp. 149-155.

[7] A. T. Tai, A. Avizienis and J. F. Meyer, "Performability Enhancement of Fault-Tolerant Software," *IEEE Trans. on Reliability,* vol. 42, pp. 227-237, 1993.

[8] K.H. Kim and H.O. Welch, "Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Comput.*, vol. 38, no. 5, pp.626-636, 1989.

[9] J.H. Lala and L.S. Alger, "Hardware and Software Fault Tolerance: A Unified Architectural Approach," *Proc. 18th. IEEE International Symposium on Fault-Tolerant Computing*, 1988, pp. 240-245.

[10] J.B. Dugan and M. Lyu, "Dependability Modeling for Fault-Tolerant Software and Systems," in *Software Fault Tolerance*, (M. Lyu, editor), WILEY, 1995, pp.109-138.

[11] T. Anderson, "A Structured Decision Mechanism for Diverse Software," in *Proc. 5th Symposium on Reliability in Distributed Software and Data Base Systems*, 1986, pp. 125-129.

[12] J.-C. Laprie, J. Arlat, C. Beounes, K. Kanoun and C. Hourtolle, "Hardware and Software Fault-Tolerance: Definition and Analysis of Architectural Solutions," in *Proc. 17th. IEEE International Symposium on Fault-Tolerant Computing*, 1987, pp. ll6-121.

[13] S. Chiaradonna, A. Bondavalli and L. Strigini, "On Performability Modeling and Evaluation of Software Fault Tolerance Structures," in *Proc. 1st European Dependable Computing Conference (EDCC-1)*, pp. 97-114, 1994.