Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components*

Pascal Chevochot, Isabelle Puaut
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France, {pchevoch,puaut}@irisa.fr

Abstract

Mainly for economic and maintainability reasons, more and more dependable real-time systems are built from Commercial Off-The-Shelf (COTS) components. To build these systems, a commonly-used assumption is that computers are fail-silent. The goal of our work is to determine the coverage of the fail-silence assumption for computers executing a real-time run-time support built exclusively from COTS components, in the presence of physical faults. The evaluation of fail-silence has been performed on the HADES *run-time support [1], aimed at executing distributed hard real-time dependable applications. The main result of the evaluation is a fail-silence coverage of 99.1%. Moreover, we evaluate the error detection mechanisms embedded in* HADES *according to a rich set of metrics, which provides guidance to choose the set of error detection mechanisms the best suited to the system needs (e.g. find the best trade-off between fail-silence coverage and overhead caused by error detection).*

1. Introduction

Mainly for economic and maintainability reasons, more and more dependable real-time systems are (partially or exclusively) made of Commercial Off-The-Shelf (COTS) components, i.e. general purpose hardware or software designed without any particular target application in mind. As COTS components are in general not designed to support dependable applications, they must be complemented by fault tolerance mechanisms (error detection, error recovery, fault treatment) in order to execute dependable applications.

The *fail-silence* assumption is commonly-used to build fault-tolerant systems. A system is said to be fail-silent [15] if it behaves either according to its specification or stops working without producing incorrect results in the time and value domains. While fail-silence is a largely used assumption, a relatively low number of studies have been undertaken to verify if such an assumption holds for systems built from COTS components.

*Appeared in Proc. of the International Conference on Dependable Systems and Networks (DSN'01), pages 304–313, Göteborg, Sweden, July 2001

The goal of this work is to determine how far it is possible to reach the fail-silence assumption for computers of a distributed real-time run-time support built exclusively from COTS components, in the presence of physical faults. The run-time support studied in this work is HADES [1] (*Highly Available Distributed Embedded System*). HADES is a run-time support for dependable applications with *hard* real-time constraints (applications whose deadlines should never be missed). It is built as a middleware layer executing on top of a COTS real-time kernel, and complements it with fault-tolerance capabilities: error detection mechanisms, used to detect value and temporal errors and stop the computer whenever an error is detected, and distributed real-time fault-tolerant protocols (reliable multicast, group membership, clock synchronization). In other to avoid the financial cost of using specific hardware, the HADES run-time support is entirely implemented in software.

The evaluation of the fail-silent behavior of computers executing HADES has been achieved using *fault injection*. A software-implemented fault-injector has been used during more than three months (night and day) to inject faults in the run-time support memory, and examine their influence on the behavior of a distributed workload application.

The main contribution of the paper is an evaluation of the coverage of fail-silence in a distributed real-time run-time support exclusively built from COTS components (hardware, operating system). The results show that with the error detection mechanisms we have designed, the fail-silence coverage does not exceed 99.1%, which in our opinion precludes the use of such a run-time support, built from COTS components, to execute safety-critical applications. Another contribution of our work is that the error detections mechanisms introduced in the paper are evaluated thanks to a rich set of metrics: fail-silence coverage, percentage of first detections, latency, memory and CPU cost, overlap between error detection mechanisms. This provides guidance to choose the set of error detection mechanisms suited to the system needs (e.g. find a trade-off between coverage of fail-silence and time overhead caused by error detection).

The rest of this paper is organized as follows. Section 2 presents the HADES run-time support and its error detection mechanisms. Section 3 describes the experimental conditions under which the HADES error detection mechanisms have been evaluated. Section 4 gives the experimental results. Our study is compared with related ones in Section 5.

2. The Hades real-time run-time support

2.1. Overview

Most COTS real-time operating systems do not provide adequate support to execute distributed applications with hard real-time and dependability constraints. Often, their only support for distribution is the provision of message passing primitives, which are generally neither predictable nor tolerant to network and computer failures. Little support (and more often absolutely no support) is provided to detect and mask crashes of remote computers. The HADES run-time support [1] mitigates this lack of functionality of COTS real-time operating systems through the provision of a set of services required for the execution of distributed dependable hard real-time applications: predictable and fault-tolerant multicast, group membership (detection of crashes of remote computers and delivery of views reflecting the current set of crashed computers), deterministic fault-tolerant clock synchronization, distributed scheduling, and synchronization mechanisms. Furthermore, in order for the run-time support running on every computer to be fail-silent, it includes a set of error detection mechanisms that stop the computer whenever an error is encountered (see § 2.2). All HADES services support transient and permanent physical faults (computer crashes and network omissions) and exhibit a timely behavior, even in the presence of faults. In addition, to avoid the financial cost of hardware dedicated to fault-tolerance, all services are entirely implemented in software.

The HADES run-time support is made of a small execution kernel and a set of service tasks. The execution kernel, identified for the sake of portability, provides a set of abstractions (address spaces, execution units, memory segments), that can be easily mapped onto the abstractions of the underlying real-time operating system. So far, HADES has been ported on Chorus, RTEMS, and has been emulated on top of Unix (in this paper, we focus on the Chorus implementation). The service tasks of HADES implement the real-time fault-tolerant services mentioned above (multicast, clock synchronization, etc).

Since HADES is designed for hard real-time systems, it does neither use nor provide dynamic memory allocation. All data structures (e.g. task descriptors, stacks) are dimensioned off-line thanks to the knowledge of the worst-case system load. Moreover, the information required in order to check off-line if the system timing constraints will be met (e.g. tasks worst-case execution times, tasks periodicity, communications and synchronization between tasks) are kept on-line for detecting errors, in the form of a graph containing all static information concerning tasks.

The HADES prototype executes on a 155Mb/s ATM network of Pentium 90Mhz PCs running Chorus 3.1.

2.2. Error detection mechanisms (EDMs)

Due to our resolution to use only COTS components, all HADES error detection mechanisms (EDMs) are software-implemented. HADES EDMs can be classified into two categories (see table 1): *redundancy-based* and *behavior-based* EDMs. In the former class of mechanisms, errors are detected through a comparison of redundant information, while in the latter, errors are detected through on-line verification of the behavior of the run-time support compared to a reference behavior. The behavior-based EDMs verify either the control flow and timing of executions or the integrity of data. In addition of the two above-described classes of mechanisms, two low-level EDMs exist (those implemented in the CPU and operating system). Note that except EDM Seman which is specific to the semantics of the code being executed, all HADES EDMs are *generalist* EDMs that can be used to detect errors in any class of software and not only in operating systems.

3. Experimental conditions

The performance of the EDMs introduced in section 2.2 have been evaluated using software-implemented fault injection (SWIFI). In this section, we define the experimental conditions of the evaluation: model of injected faults (§ 3.1), the sets of experiments used to compute the dependability metrics given in section 4 (§ 3.2), the fault injector (§ 3.3) and the workload used to evaluate the system dependability (§ 3.4).

3.1. Fault model

The objective of fault injection is to simulate the occurrence of physical faults. We simulate transient physical faults, which are the most common physical faults in computer systems [11], by inserting their manifestations (errors) into the run-time support memory[1]. The injection process is triggered at a randomly selected instant. It consists in the corruption of a randomly selected byte within the run-time support memory (code, data and stack of the run-time support tasks, as well as the underlying micro-kernel). The fault injector performs bit flips on the selected byte (a single bit flip in this paper). This model is in accordance with most SWIFI tools (e.g. [3, 10, 4, 17]). The representativity of this fault model with respect to the actual consequences of faults is discussed in [5].

3.2. Injections, activations, justified detections

As a consequence of a fault injection, two outcomes can be observed: (i) the system does not use the corrupted word.

[1]In the following, although we actually *simulate* faults by injecting errors, we still use the term *fault* injection.

	a. Redundancy-based EDMs
Coding	*Data coding in the communication software.* Coding is used to detect corruptions during local data transmissions. Coding is achieved by adding a checksum byte to messages.
Robust	*Robust data structures (lists, stacks).* This mechanism is used to detect errors affecting the run-time support data structures. For instance, a robust list uses redundant information (list identifier, backward pointer) to check the list integrity.
Coding_Static	*Coding of the run-time support static data structures.* This mechanism adds a checksum to a part of the run-time support static data structures (the static graph defining the tasks and the communications among them). The verification is achieved in the run-time support idle loop.
	b. Behavior-based EDMs
Deadline	*Detection of task deadline misses.* This mechanism checks that no task misses its deadline.
Timeout	*Detection of WCET exceeding.* This mechanism monitors the execution of tasks to check if their execution time is not longer than their worst-case execution times (WCET).
Arrival	*Verification of arrival laws.* This mechanism verifies that the arrivals of every task conform to the task expected arrival law (periodic, sporadic).
Callgraph	*Verification of the run-time support call graph.* This mechanism checks the conformance of the run-time support execution flow with a call graph generated off-line. It is implemented by automatically rewriting the source code of the run-time support (addition of code for the control flow verification).
Automaton	*Verification of protocol states.* This mechanism verifies the validity of transitions for the run-time support protocols modeled using finite-state automatons (e.g. states of a task – blocked, active, running, finished).
ECI	*Error capturing instructions.* This mechanism inserts special instructions within the code and data segments in areas that are never executed during normal program execution. These instructions trigger an exception when executed, such that an error causing the processor to execute instructions in a unused area is detected.
Vmemory	*Virtual memory.* This mechanism triggers an exception when a task accesses a memory segment which is not actually mapped into the current address space, or is mapped with inadequate access rights.
Structure	*Verification of the validity of data structures.* This mechanism verifies the validity of data structures without using redundancy. This mechanism includes verifications of array indices, and validity checks on lists.
Seman	*Assertions.* This mechanism consists of assertions inserted into the code of the run-time support to detect violations of invariants that must hold during normal program execution.
	c. Low-level EDMs
CPU	This mechanism represents the set of basic EDMs embedded in the Pentium processor: invalid opcode, overflow, division by zero, stack fault exception, protection violation, FPU exception. This EDM cannot be turned off.
OS	EDMs included into the Chorus real-time kernel (error codes returned by system calls, kernel panics).

Table 1. Error detection mechanisms (EDMs)

Obviously, this situation cannot lead to a system failure ; (ii) the system uses it. In this situation, the fault is said to be *activated*. Let us call S_{inj}, of cardinal N_{inj} the set of fault injections (see figure 1). And let $S_{act} (\subset S_{inj})$ be the set of activated faults. To compute the fail-silence coverage (§ 4.1), like in most related studies using SWIFI, we only consider activated faults. Considering all injected faults would lead to a coverage of the fail-silent assumption that is too highly dependent on the target system.

As introduced in [18], and depicted in figure 1, two orthogonal classes of manifestations can be observed as a consequence of a fault injection that led to a fault activation:

– an error can be detected by an EDM (set of experiments S_{det} of cardinal N_{det});
– the fault can lead to a system failure (set of experiments S_{fail} of cardinal N_{fail}).

Note that since in practice the duration of an experiment after a fault activation is bounded (here 20 s), an absence of failure during the experiment duration can have two explanations: (i) the system has been and will perform as specified (no error or overwritten error); (ii) the experiment duration has been too short in order for the error to declare (the error is *latent*). To increase the confidence in our experimental results, we have tried experiment durations longer than 20 s and observed that it does not change the faults manifestations. We consider in the following that no error stays latent after every experiment.

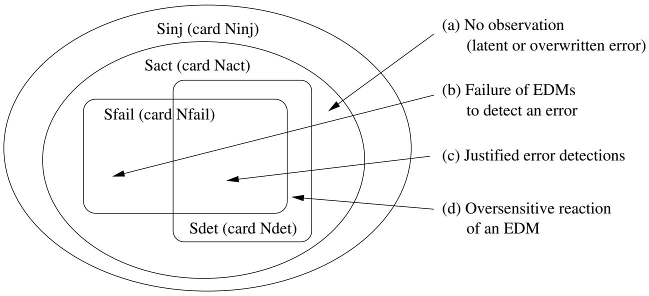

Figure 1. Fault manifestations

The combination of the two classes of manifestations S_{det} and S_{fail} are depicted by labels (a) to (d) in figure 1. We call error detections of the set $S_{just} = S_{det} \cap S_{fail}$ *justified* error detections. The set of experiments $S_{det} \backslash S_{fail}$ indicates an interesting scenario: an “oversensitive” EDM may detect an error that does not result in a system failure. This can occur for instance when an EDM detects an error in an area that is never accessed by the system (except by the EDM itself). Although there is no danger for system malfunction in this situation, it can positively bias the efficiency of some EDMs (see [18] and § 4.3). Note that detecting oversensitive reactions requires it to be known whether an injection leads to a system failure. It thus requires that the system execution is not stopped after an error detection to be able to observe the application results. To compute the percentage of first detections and the latency of EDMs (§ 4.3 and 4.4), in order to avoid positively biased measures for oversensitive EDMs, the measures were obtained by using only justified error detections.

3.3. Fault injector

The implementation of the fault injector highly relies on the Pentium debugging features. We use the Pentium ability to raise an exception (*debug exception*) when a given location is accessed. Fault injection proceeds in three steps:

1. *Selection of the fault injection time:* the time of the fault injection is selected at random, and a timer is programmed to perform the actual injection.
2. *Fault injection:* memory is modified to actually inject the fault. The physical address where the fault should be injected is selected at random. One bit at the selected address is inverted. One of the Pentium breakpoint-address register is set so that a debug exception is raised when this address is accessed.
3. *Fault activation:* the activation of a fault is detected thanks to the triggering of a debug exception.

Steps 1 and 2 are retried (and the effects of non-activated faults are undone) until a fault is activated. We consider that a fault is not activated if more than 10 seconds have elapsed since the fault injection. The delay of 10 s has been selected because it is much larger than the period of all tasks executed in the system. Informations concerning the costs of fault injection and activation can be found in [5].

Figure 2 depicts the environment into which the fault injection campaigns have been conducted. The system is made of a set of five computers connected by ATM and Ethernet. Four target computers (left of the figure) execute applications that exchange messages through ATM. Faults are injected in the run-time support of one target computer (in the memory areas depicted as shaded boxes in Fig 2). The host computer (right of the figure) collects the results of fault injections.

A fault injection campaign consists in a sequence of experiments. During each experiment, a workload applica-

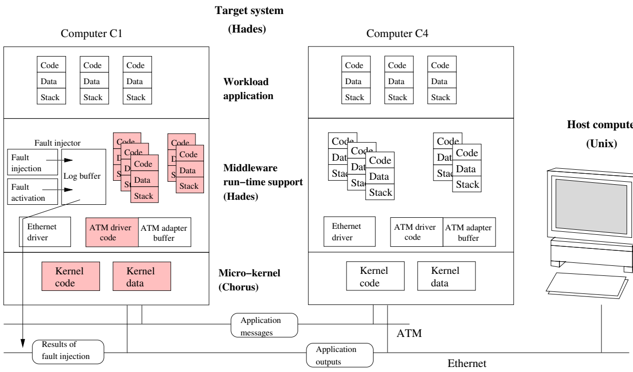

Figure 2. Experimental setup

tion (see § 3.4) is started. The application results are regularly transmitted to the host computer through the Ethernet. Faults are injected until a fault is activated. Subsequently, when an error is detected, the concerned computer is stopped (except when we are interested in justified error detections – § 4.3, 4.4, and 4.6 – in which case the execution continues). Then, in all cases, after a time interval of 20 seconds relative to the fault activation time, the results of the experiment are sent to the host computer. The computers of the target system are then rebooted for the next experiment.

3.4. Workload

The workload is a distributed object tracking application. Its purpose is for slave mobile objects to follow the movements of a master mobile object. Sensors, located in one of the computers, give the current position of all objects, and actuators, located in the same computer, command their movements (in this study, sensors and actuators are emulated in software). The sensors are sampled periodically with a period of $250ms$. The computation of the trajectory of the slave objects is achieved by a task with a period of $250ms$, which is distributed on the four HADES computers, each computer managing a static set of $N/4$ objects. Two small tasks, independent of the object tracking application, and aiming at testing the correctness of synchronization, complete the workload. An interesting aspect of this workload application is that it covers a large percentage of the HADES functionalities. In addition, its correctness in the time and value domains can be easily checked since the application is deterministic.

4. Results of the experiments

The EDMs introduced in Section 2.2 are evaluated according to different metrics: coverage of fail-silence (§ 4.1), percentage of first detections (§ 4.3), latency (§ 4.4), and overlap with other mechanisms (§ 4.6). In addition, paragraph 4.2 analyses the fail-silence violations.

4.1. Coverage of the fail-silence assumption

The fail-silence coverage is evaluated by injecting faults in the run-time support memory image, and observing their consequences on the workload application of § 3.4. Due to the presence of hard real-time constraints, the applications results are characterized not only by their *value* but also by the *time* when they are produced. An activated fault can have the following manifestations on the workload application output :

Crash	Crash of at least one computer, the others returning correct results in the time and value domains
Wval-Ctim	Wrong values for at least one computer, correct timing for all computers
Cval-Wtim	Correct values for non-crashed computers, wrong timing for at least one computer
Wval-Wtim	Wrong values for at least one computer, wrong timing for at least one computer
Cval-Ctim	Correct results (in the time and value domains) for all computers

The first four classes of fault manifestations given above can further be subdivided depending on whether or not a computer different from the one where the fault was injected has failed (i.e. the fault has propagated). In the following, we call the resulting classes of manifestations Crash-NP, Crash-P, Wval-Wtim-NP, Wval-Wtim-P, Wval-Ctim-NP, Wval-Ctim-P, Cval-Wtim-NP, Cval-Wtim-P and Cval-Ctim (*NP* stands for *NoPropag*, *P* stands for *Propag*).

A computer is fail-silent if the application results are correct in the time and value domain (category Cval-Ctim) or the computer crashes without propagating the error to another computer (category Crash-NP). To compute the coverage, we consider the set of activated errors S_{act}:

$$coverage_{fail-silence} = \frac{(N_{Cval-Ctim} + N_{Crash-NP})}{N_{act}}$$

Table 2 depicts the observed fault manifestations, fail-silence coverage and corresponding confidence interval for two configurations of HADES: (a) without EDMs (i.e. with only the CPU EDMs and kernel panics, that cannot be turned off); (b) with all the EDMs of § 2.2.

In a system without error detection, 19.4% of the faults caused fail-silent violations, which shows that without EDMs, the fail-silence coverage is modest. However, the fail-silence coverage is higher than in systems using processors without error detection. For instance, the study described in [12] reports that 45.6% of faults caused fail-silence violations for a processor without error detection (Z80).

When all EDMs are used, 99.1% of activated faults result in a fail-silent behavior. 99.1% is undeniably below the coverage obtained in systems with hardware support for fault-tolerance. Nevertheless, the EDMs embedded in HADES are rather efficient, since they divide the number of fail-silence violations by a factor 22.

Fault manifestation	(a) No EDM		(b) All EDMs	
Injected faults (N_{inj})	39025		29591	
Activated Faults (N_{act})	3152		3012	
Cval-Ctim	1839	(58.34%)	1122	(37.25%)
Crash-NP	703	(22.30%)	1862	(61.82%)
Wval-Ctim-NP	10	(0.32%)	2	(0.07%)
Cval-Wtim-NP	22	(0.70%)	0	(0%)
Wval-Wtim-NP	1	(0.03%)	0	(0%)
Crash-P	102	(3.24%)	22	(0.73%)
Wval-Ctim-P	127	(4.03%)	4	(0.13%)
Cval-Wtim-P	295	(9.36%)	0	(0%)
Wval-Wtim-P	53	(1.68%)	0	(0%)
Coverage of fail-silence	80.65%		99.07%	
Confidence interval (confidence level of 95%)	[79.49%,81.79%]		[98.80%,99.33%]	

Table 2. Fail-silence coverage

4.2. Analysis of fail-silence violations

This paragraph examines in more details the fault injections that led to fail-silence violations. Note that due to the number of injected faults and the fail-silence coverage, the number of fail-silence violations is very small (28). Thus, the confidence interval of the figures given in this paragraph is rather large and as a consequence all figures given in the paragraph should be looked at with precautions.

Consequences of fail-silence violations on applications. For 26 out of 28 fail-silence violations (92 % of fail-silence violations), there are fault propagations across computers. However, for 22 of these cases (78 % of fail-silence violations), fault propagations only provoke computer stops without the production of incorrect results at the application level. For a lot of applications, such a failure mode does not lead to catastrophic consequences.

Faults responsible of fail-silence violations. To identify memory regions *sensitive* to faults, i.e. regions in which faults are likely to lead to fail-silence violations, we give the repartition of the faults responsible of fail-silence violations according to the memory regions where the faults were injected. In the table, the last row corresponds to experiments in which no information was received by the host computer because the system under test has crashed without sending fault injection results.

Memory region	Number of errors responsible of fail-silence violations	
Run-time support code	10	(35 %)
Run-time support data	13	(45 %)
Run-time support stacks	2	(7 %)
Micro-kernel code	0	(0 %)
Micro-kernel data	0	(0 %)
Unknown memory region	5	(3 %)

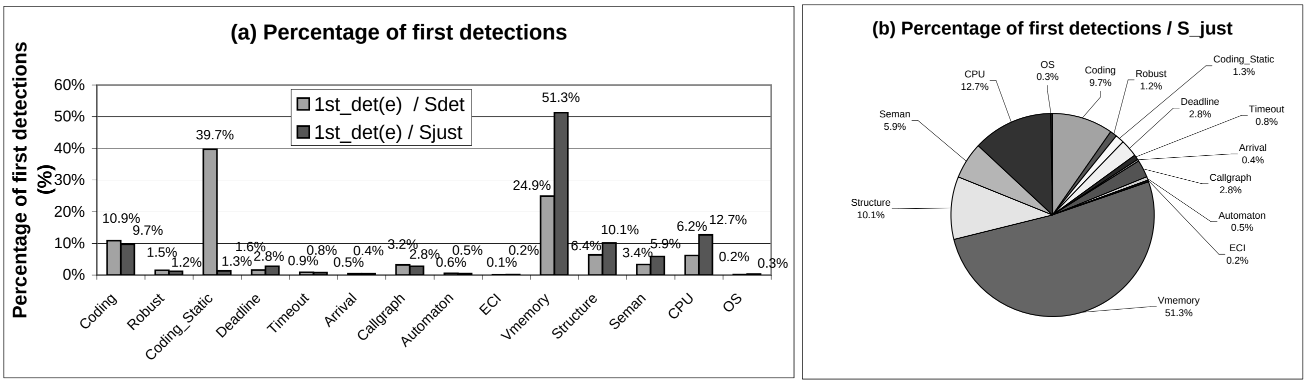

Figure 3. Percentage of first detections of the EDMs

The figures given in the table show that data regions are slightly more sensitive than code regions. Moreover, the run-time support memory regions are more sensitive to faults than the micro-kernel memory regions.

The following table gives more information concerning the 25 faults injected into the run-time support, that led to fail-silence violations. The figures show that the communication software is responsible of most fail-silence violations.

Service component	**Nb. of errors resp. of fail-silence violations**	
Execution kernel (scheduler, synchro.)	5	(20 %)
Clock synchronization	2	(8%)
Reliable multicast protocol	13	(52 %)
ATM driver	5	(20 %)

One typical example of fail-silence violation is the corruption in the communication protocol of the local variable identifying the message recipient just before the message checksum is computed. Such a fault remains undetected on the sender side and will only be detected upon message receipt. Other examples of fail-silence violations are given in [5]. Among the 28 fail-silence violations, and especially in those concerning the communication software, we were unable to find a low cost generalist EDM able to detect the errors (should it be done in high level or low level communication software) before it propagates to other computers.

4.3. Percentage of first detections

A metric to evaluate the efficiency of an EDM e is the percentage of first detections achieved by e on a set of experiments. Let e be an EDM, and S be a set of experiments into which an error is detected by any EDM. Let $det(e)$ be the number of times e was the first to detect the error. The percentage of first detections achieved by e, and noted $1st\_det(e)$, is defined as $1st\_det(e) = \frac{det(e)}{|S|}$.

To detect oversensitive EDMs, we have computed $1st\_det(e)$ for every EDM e on two sets of experiments S:

– the whole set of experiments where an error has been detected ($S = S_{det}$);
– the set of experiments where an error has been detected in a justified manner ($S = S_{just} = S_{det} \cap S_{fail}$). Note that computing S_{just} requires that the system execution is not stopped when an error is first detected. This metric corresponds to the metric noted $1st\_det(e)$ in [18].

The percentage of first error detections was computed on $N_{inj} = 71068$ fault injections, that led to $N_{act} = 7123$ fault activations. Among activated errors, $N_{det} = 4288$ of them were detected, and $N_{just} = 2086$ error detections were justified.

The left part of figure 3 depicts $1st\_det(e)$ for the sets of experiments S_{det} and S_{just}. It shows that Coding_Static and less importantly Coding and Callgraph are oversensitive EDMs. The unjustified detections of Coding_Static come from the fact that this mechanism detects errors on *all* the run-time support static task graph, even the parts that are never used by the workload (initialization tasks, padding data). The oversensitive detections of Coding come from the fact that messages have a fixed size in our system, and that Coding adds a checksum for the whole message, even if only a part of the message contains useful data. Callgraph has oversensitive reactions when a fault is injected in the code in charge of checking the validity of the system control flow.

The right part of figure 3 gives the values of $1st\_det(e)$ for the set of justified error detections S_{just}. It shows that a majority of errors (51.3%) is detected by mechanism Vmemory. A lower percentage of errors is detected by Coding, Structure and CPU (approximately 10% each). The efficiency of Structure is due to the heavy use of arrays and lists in HADES. Approximately 3% of errors are detected by mechanism Callgraph, which checks the run-time support control flow. The other mechanisms, and in particular the mechanisms that verify the tasks timing, detect less that 2.5% of errors.

4.4. Error detection latency

We define the *latency* of error detection as the delay between fault activation and the first error detection by an EDM. In order not to bias the results in favor of oversensitive EDMs, the latency is computed only on a set S_{just} of justified error detections. Latencies were computed on the set of experiments used in the previous paragraph. Latencies (as well as all other timing data given in the paper) are obtained by reading the Pentium cycles counter.

For 2086 justified error detections, the average error detection latency is 47.47 ms. An examination of the repartition of the error detection latencies exhibits a small number of very high detection latencies (from 1 to 20 seconds). When the extreme 10% (highest and lowest) values are excluded, the average error detection latency is 7.80 ms. For 90% of justified error detections, the detection latency is lower that 10 ms.

The average delay between the injection of an error and its activation is rather high (76.5 ms, and 37.7 ms when excluding the extreme 10%); in 60% of the experiments, the duration between the fault injection and its activation is lower than 10 ms.

Figure 4 gives the average error detection latency of every EDM. In the figure, EDMs marked with an asterisk are those that detect a low number of errors (less than 20), thus leading to imprecision in the computation of the average latency.

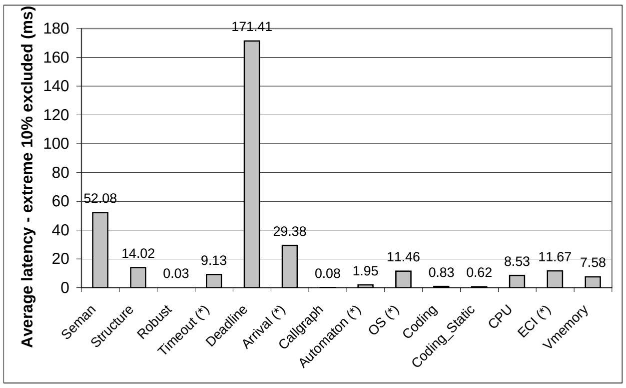

Figure 4. Average detection latency per EDM

According to their error detection latencies, the EDMs can be divided into three broad categories. The mechanisms of the first class (Deadline, Timeout, Arrival) are the ones whose speed to detect an error depends on the temporal characteristics of executed tasks. For instance, mechanism Deadline cannot detect an error before the task deadline has been missed. Due to the temporal properties of the tasks executed in the experiments (periods of tens of milliseconds for periodic tasks, safe and therefore overestimated execution times and deadlines), this class of mechanisms have large error detection latencies (from 20 ms to 140 ms in average). The mechanisms of the second category (Vmemory, CPU) are hardware-implemented EDMs; their average error detection latency is 8 ms. Finally, all other EDMs can be classified in a third category, which includes only software-implemented EDMs with a rather low error detection latency (in average, less than 1 ms, except for mechanisms Seman and Structure).

Figure 5 gives for every range of error detection latencies the number of errors detected.

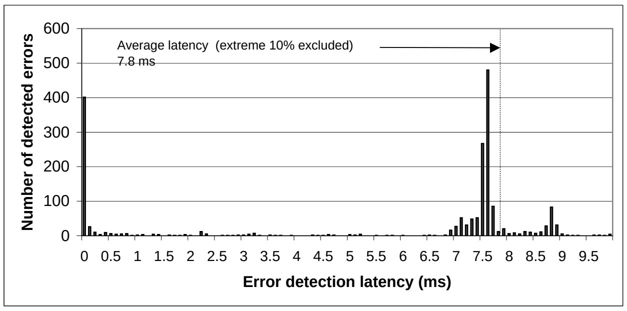

Figure 5. Repartition of detection latencies

Two peaks can be observed in the histogram: one corresponding to EDMs with a latency below 1 ms (third class of EDMs introduced above), and one to EDMs with a latency between 7 and 9 ms (second class of EDMs).

4.5. Memory and timing costs of EDMs

Figure 6 details the EDMs memory and timing costs. The memory and timing costs have been measured by adding the EDMs one by one by decreasing order of percentage of first error detections (i.e. according to metric 1st_det(e) computed on justified error detections – see figure 3.b). The memory cost of the Chorus kernel (ranging from 169Kb to 191Kb depending on whether memory protection is used or not) are excluded from figure 6.a.

Figure 6.a details the memory cost of the EDMs for the code and data segments. The code size of the run-time support with all EDMs included is 6.7 times as big as the code size of a run-time support without EDMs. Mechanism Vmemory alone multiplies the volume of code of the run-time support by a factor 3.5. This is due to the duplication of library code – libc and other HADES internal libraries – in several address spaces (the tool chain of the selected Chorus configuration does not provide direct support for shared libraries). Mechanism Structure also adds a large code overhead. This is because the data structures covered by Structure are heavily used in the run-time support, and thus have been in-lined. Mechanism Callgraph also has a rather high code overhead. This is due to the presence in HADES, for the sake of modularity, of a large number of small functions. Concerning the increase of the size of data, only mechanism Coding is noticeable. All other EDMs use stack-allocated variables having no impact on the data size of the run-time support.

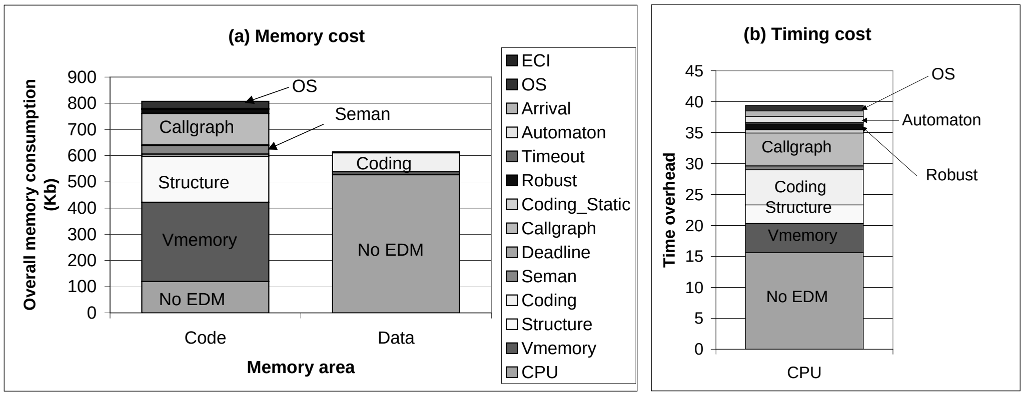

Figure 6. Memory and timing costs of EDMs

The timing overhead of the EDMs on the run-time support activity is depicted in figure 6.b. The time spent in the run-time support[2] is multiplied by a factor 2.5 in consequence of the addition of error detection. The more time-consuming EDMs are, by decreasing time consumption, Coding (14.4% of the time spent in the run-time support), Callgraph (13.1%), Vmemory (12.0%) and Structure (7.5%). The other EDMs have an overhead lower than 3%.

A factor 2.5 is in our opinion rather satisfactory for entirely software-implemented EDMs. The alternative would have been to use temporal redundancy for the run-time support activities to check that they behave in a fail-silent manner. Using temporal redundancy for this class of activities seems to us very tricky to implement and costly to execute, because run-time support activities usually do not return any value but rather have side effects (IOs, modifications of the run-time support data structures like the run-queues). Using temporal redundancy would increase the execution time of the run-time support activities by at least a factor two, but probably much more in order to check the equivalence of successive executions.

4.6. Overlap between EDMs

In this paragraph, we are interested in finding EDMs that overlap. The final objective is to remove some EDMs, in order to reduce the overall cost of error detection with only a slight decrease of the fail-silence coverage.

Note that it is not possible to complete fault injection campaigns to evaluate the fail-silence coverage obtained for every possible combination of EDMs. Indeed, a fault injection campaign for 3000 activated errors takes at least four days. Trying all possible combinations would require $2^{13} * 4$ days of experiments (90 years). In addition, our fault injection system does not allow to have a precise knowledge of the fail-silence coverage that would be obtained if one or several EDMs were removed.

To guide in the process of removing EDMs, we have defined a new metric, named *ordered overlap*. This metric, for every EDM e is computed on the set of experiments where e is the first to detect an error and detects it in a justified manner. It counts the average number of EDMs having detected the error *after* e. The higher is this metric for an EDM, the more we can expect that the removal of this EDM is compensated by the others EDMs. In comparison with the metric *overlap* proposed in [18], *ordered overlap* takes the detection order into account, which is important to enforce fail-silence. The evaluation of *ordered overlap* for EDMs (whose percentage of first detections is upper than 1 %) is the following:

EDM	*1st_det*	*ordered overlap*
Vmemory	51.25 %	0.00
Structure	10.12 %	1.12
Coding	9.68 %	3.30
Seman	5.85 %	1.85
Deadline	2.78 %	2.83
Callgraph	2.78 %	1.17
Coding_Static	1.34 %	1.54
Robust	1.20 %	0.88

To choose the EDMs to remove, we have made four fault injection campaigns: the first three evaluate the fail-silence obtained when removing one of the three EDMs with the highest *ordered overlap* (Coding, Deadline and Seman); the fourth one evaluates the fail-silence coverage when removing Vmemory, for which the *ordered overlap* metric is perverted because the execution is necessary stopped after an error is detected. The following fail-silence coverage have been obtained on campaigns with 3000 activated faults: 97.5 % with all EDMs except Coding, 98.3 % with all EDMs except Deadline, 98.7 % with all EDMs except Seman, and 98.9 % with all EDMs except Vmemory.

[2] As mechanism Coding_Static executes in the run-time support idle loop, its timing cost is not measured like the other EDMs by measuring the percentage spent in the run-time support executing this EDM. We only measure the indirect overhead Coding_Static adds on the other mechanisms (due to its poor locality of reference, it makes the number of cache misses in the system increase).

From these results, we choose to remove the EDM which has the smallest influence on the fail-silence coverage, EDM Vmemory. We also remove EDMs Timeout, Automaton, Arrival, OS and ECI, whose percentage of first error detections is less than 1 % (a campaign in which only these EDMs are removed showed a fail-silence coverage of 98.7 %). The following table evaluates the new set of EDMs, i.e. EDMs Coding, Robust, Coding_Static, Deadline, Callgraph, Structure, Seman and CPU.

	All EDMs	Reduced set of EDMs
Fail-silence coverage	99.07 %	98.34 %
Confidence interval (confidence level 95%)	[98.80%,99.33%]	[97.98%, 98.70%]
Average latency (extreme 10% excluded)	7.8 ms	3.6 ms
Time overhead	x 2.5	x 1.7
Overall memory overhead	x 1.8	x 1.1
Memory overhead, code	x 6.7	x 1.4
Memory overhead, data	x 1.2	x 1.1

With the reduced set of EDMs, the time overhead due to error detection is reduced by 32%, the memory overhead is reduced by 45%, and this with only a slight decrease of the fail-silence coverage (0.73%).

5. Related work

There are a large number of hardware or software methods to detect errors. They can be classified in two categories. *Redundancy-based* EDMs use redundancy of data, hardware or pieces of software to detect errors. Examples of redundancy-based EDMs are error detecting/correcting codes, checksums, robust data structures, temporal or lock-step replication of processes, recovery blocks or N-version programming. *Behavior-based* EDMs detect errors through the detection of a deviation of the system behavior compared to a *reference* behavior. This category of mechanisms include mechanisms that verify the system timing and control flow: from simple watchdog timers to watchdog processors [13], software-implemented control flow checking mechanisms such as [14]. This category of mechanisms also include mechanisms that check data validity: verification of value domains [8], Algorithm-Based Fault-Tolerance [9]. The study described in this paper uses both categories of mechanisms. However, to avoid the financial cost of using dedicated hardware, we concentrate on software-implemented mechanisms, and focus on those that have a low cost (i.e. do not replicate entire computations) and require a modification of neither the compiler nor the operating system. Moreover, except EDM Seman, the EDMs integrated in HADES can be applied without modification to a different kind of software.

Fault injection has been used in many studies to evaluate the dependability of systems. Some of these studies analyze the failure modes of (hardware or software) systems in general, while other ones concentrate on a specific dependability metric which is the coverage of the fail-silence assumption.

In the first class of studies, Damn [6] compares the set of EDMs integrated into a real-time operating system (hardware mechanisms, parameter checks, process replication, robust data structures, check of time-limits) in the presence of hardware faults. The main performance metric used is the percentage of first detection of the mechanisms computed on all detected errors. This study shows a much higher percentage of first detections achieved by the hardware (68000 processor) that in our results, but a similar percentage of first detection for timing checks (less than 4% in both studies). Salles et al. [17] propose an approach (use of reflexivity and fault-containment wrappers) to contain the effect of external and internal faults that can affect the behavior of microkernels. A set of fault-containment wrappers have been designed. In the presence of physical faults in the text region of the Chorus microkernel, the fail-silence coverage ranges from 97.7% (faults in the scheduler) to 99.5% (faults in the synchronization module). Note that in this study, in contrast to our work, the workload is not distributed, and the costs of the fault containment wrappers is not given. An experimental evaluation of the communication capabilities of Delta-4 in the presence of physical faults is reported in [2]. The basis for the Delta-4 multipoint services is an atomic multicast protocol (AMp) implemented in communication servers called (NACs) that exhibit a fail-silent behavior thanks to self-checking hardware. The study evaluates through pin-level fault injection both the coverage of the NACs self-checking mechanisms and the behavior of the AMp in the presence of faults.

The studies presented in [12], [7] and [16] focus on the coverage of the fail-silence assumption in the presence of hardware faults. Madeira and Silva [12] evaluate the fail-silence coverage in systems without hardware and software replication. The EDMs used are behavior-based EDMs (ECI, hardware-implemented control flow checking). The coverage obtained using pin-level physical fault injection demonstrated a fail-silence coverage of 98.1% on a 68K, and 99.6% on a Z80. When only software-implemented EDMs are used (semantic checks, ABFT, very simple control flow checking) another study from the same research group has shown a fail-silence coverage of 98.2% on a 68K processor [16]. It is difficult to compare the studies completed in [12] and [16] because of the different nature of the software which is evaluated (scientific applications for [12] and [16], operating system in our study). Fuchs [7] evaluates the fail-silence coverage of the Mars system using SWIFI. Mars includes EDMs in the hardware (CPU plus dedicated hardware), operating system (assertions) and in the application (CRC, temporal replication). The results of this study are a coverage of 92% when only hardware, operating system and CRCs are used, and 100% when temporal replication is added. Compared with this work, our study shows a lower coverage of the fail-silence assumption, but without the use of dedicated hardware.

6. Conclusion

In this paper, we have presented an evaluation of the fail-silent behavior of computers executing a run-time support for hard real-time dependable applications built from COTS components. The run-time support contains error detection mechanisms (EDMs), entirely implemented in software, used to enforce fail-silence. The main result of the evaluation is a fail-silence coverage of at most 99.1%, obtained when the whole set of error detection mechanisms is used. Compared to the initial coverage of 80.6 %, obtained when no EDM is used, this figure is rather good (the number of fail-silence violations is divided by a factor 22). However, this coverage is clearly not sufficient to be used in safety-critical systems. Achieving a fail-silence coverage of 99.1% comes with a CPU and memory cost: the time spent in the run-time support is multiplied by a factor 2.5 when EDMs are introduced, and its memory size is multiplied by a factor 1.8. The average detection latency is of several milliseconds. These costs are far from being negligible, and mainly come by the fact that EDMs are software-implemented. However, a study of the overlap between EDMs has shown that some of them can be removed, thus lowering the memory and CPU cost of error detection (x 1.1 and x 1.7) with only a slight decrease of the fail-silence coverage (98.3 %).

As a future work, an interesting study would be to compare on the same platform our EDMs with hardware-implemented EDMs. Another research direction would be to design a general methodology to select sets of EDMs.

Acknowledgements This work is partially supported by the French Department of Defense (DGA/DSP), #98.34.375.00.470.75.65. The authors wish to thank D. Powell and the anonymous reviewers for their valuable comments on this work.

References

[1] E. Anceaume, G. Cabillic, P. Chevochot, and I. Puaut. A flexible run-time support for distributed dependable hard real-time applications. In *Proceedings of the 2nd International Symposium on Object-oriented Real-time distributed Computing*, pages 310–319, St Malo, France, May 1999.

[2] J. Arlat, M. Aguera, Y. Crouzet, J. Fabre, E. Lartins, and D. Powell. Experimental evaluation of the fault tolerance of an atomic multicast system. *IEEE Transaction on Reliability*, 39(4):455–467, Oct. 1990.

[3] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–581, Apr. 1990.

[4] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, Feb. 1998.

[5] P. Chevochot and I. Puaut. Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components. Technical Report 1370, IRISA, Oct. 2000. ftp://ftp.irisa.fr/techreports/2000/PI-1370.ps.gz.

[6] A. Damm. The effectiveness of software error-detection mechanisms in real-time operating systems. In *Proc. of the 16th International Symposium on Fault-Tolerant Computing Systems*, pages 171–176, Vienna, Austria, June 1986.

[7] E. Fuchs. Validating the fail-silence assumption of the MARS architecture. In *Proc. of the 6th Dependable Computing for Critical Applications Conference*, pages 225–247, Garmisch-Partenkirchen, Germany, 1998.

[8] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proc. of the 2000 International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, pages 24–33, New York, USA, June 2000.

[9] K. H. Huang and J. Abraham. Algorithm-based fault-tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, June 1984.

[10] G. Kanawati, N. Kanawati, and D. Tang. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, 1995.

[11] P. Lala. *Fault Tolerant and Fault Testable Hardware Design*. Prentice Hall, 1985.

[12] H. Madeira and J. G. Silva. Experimental evaluation of the fail-silent behavior in computers without error masking. In *Proc. of the 24th International Symposium on Fault-Tolerant Computing Systems*, pages 350–359, Austin, USA, June 1994.

[13] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb. 1988.

[14] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin. Two software techniques for on-line error detection. In *Proc. of the 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 328–335, Boston, USA, June 1992.

[15] D. Powell, G. Bonn, D. Seaton, P. Veríssimo, and F. Waeselynck. The delta-4 approach to dependability in open distributed computing systems. In *Proc. of the 18th International Symposium on Fault-Tolerant Computing Systems*, pages 246–251, Tokyo, Japan, June 1988.

[16] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental evaluation of the fail-silent behavior in programs with consistency checks. In *Proc. of the 26th International Symposium on Fault-Tolerant Computing Systems*, pages 394–403, Sendai, Japan, June 1996.

[17] F. Salles, M. R. Moreno, J. Fabre, and J. Arlat. Metakernels and fault containment wrappers. In *Proc. of the 29th International Symposium on Fault-Tolerant Computing Systems*, pages 22–29, Madison (WI), June 1999.

[18] A. Steininger and C. Scherrer. On finding an optimal combination of error detection mechanisms based on results of fault injection experiments. In *Proc. of the 27th International Symposium on Fault-Tolerant Computing Systems*, pages 238–247, Seattle, WA, USA, June 1997.