Aspect-Oriented Fault Tolerance for Real-Time Embedded Systems

Francisco Afonso[1] Carlos Silva[1] Nuno Brito[1] Sergio Montenegro[2] Adriano Tavares[1]

[1] Department of Industrial Electronics
University of Minho
Campus de Azurém
4800-058 Guimarães - Portugal

{fafonso, csilva, nunobrito, atavares}@dei.uminho.pt

[2] German Space Agency (DLR)
Compact Satellite Program
Am Fallturm 1
28359 Bremen - Germany

sergio.montenegro@dlr.de

ABSTRACT

Real-time embedded systems for safety-critical applications have to introduce fault tolerance mechanisms in order to cope with hardware and software errors. Fault tolerance is usually applied by means of redundancy and diversity. Redundant hardware implies the establishment of a distributed system executing a set of fault tolerance strategies by software, and may also employ some form of diversity, by using different variants or versions for the same processing.

This paper describes our approach to introduce fault tolerance in distributed embedded systems applications, using aspect-oriented programming (AOP). A real-time operating system sup-porting middleware thread communication was integrated to a fault tolerant framework. The introduction of fault tolerance in the system is performed by AOP at the application thread level. The advantages of this approach include higher modularization, less efforts for legacy systems evolution and better configurability for testing and product line development. This work has been tested and evaluated successfully in several fault tolerant configurations and presented no significant performance or memory footprint costs.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.4.7 [**Operating Systems**]: Organization and Design – Real-Time Systems and Embedded Systems.

Keywords

Aspect-oriented programming, fault tolerance.

1. INTRODUCTION

Embedded systems have a widespread use in several domains of safety-critical applications, as process control, avionics, space and medical systems. These systems usually must satisfy real-time performance requirements, and so the correct response depends also on the time which it is produced. Generally a real-time system executes a series of tasks subjected to deadlines and jitter constraints. For many of these applications, there are serious constraints in physical size and energy consumption, which imply in reduced processing power and memory size. Furthermore, these systems must exhibit high dependability [5], a concept that involves not only reliability, but also other attributes as availability, safety and maintainability.

The means of achieving dependability include fault prevention and fault removal techniques, but usually fault tolerance (FT) techniques are needed to cover transient faults, hardware permanent faults and residual software faults. The application of fault tolerance techniques is rather difficult. Redundant hardware involves extra software coordination, which makes the software system more complex and prone to errors.

The contribution of this work is evaluating the application of aspect-oriented techniques to the development of real-time embedded fault-tolerant software. In contrast with previous works, we studied the usage of AOP at the application thread level, based on a thread model commonly used for embedded systems software development. We considered in this work a small real time operating system named BOSS, and a fault tolerance framework that supports several FT mechanisms, both for hardware and software faults. The proposed solution was evaluated qualitatively and quantitatively in terms of performance and memory footprint in relation to non-AOP implementations. In addition, a case study for testing this approach is described.

2. FAULT TOLERANCE CONCEPTS

A fault is active when it produces an error in the system state. An error may propagate and lead to a subsequent service failure. Fault tolerance is a means of achieving a continuous system service in the presence of active faults [5]. Several FT strategies have been proposed and applied in the last 30 years. Some strategies are based on single version software, and can only be effective with hardware faults and transient software faults. One example is Rollback/Retry, also called “checkpoint and restart” [16]. In this strategy the detection of an error triggers a system rollback to a previously saved state and a re-execution of the same processing. This technique is based on backward error recovery and needs an efficient error detection mechanism. Other strategies apply hardware redundancy to detect and mask errors, as Triple Modular Redundancy (TMR) [18], where error detection is performed by comparison of the results of multiple hardware/software units.

ACP4IS ’08, 31st March 2008, Brussels, Belgium

In order to deal with permanent software faults, multiple version software (software diversity) is needed. Several strategies have been proposed as Recovery Blocks (RB) [17], Distributed Recovery Blocks (DRB) [10] and N-Version Programming (NVP) [6].

RB and DRB perform backward error recovery like Roll-back/Retry, but use different software versions, or variants, in each execution block. The main difference between RB and DRB is the distributed nature of the latter, which allows concurrent execution of variants in two distinct nodes and coordination between them to define what node will send the final output.

NVP is a FT strategy that uses forward error recovery in which multiple variants (at least 3) run sequentially or concurrently. A decision mechanism selects the correct response usually by majority voting. In a multi-computer system, each variant runs in a different node and the decision mechanism (voter) may be replicated too.

In this work, RB, DRB and NVP strategies are supported, as well as single version techniques related to them, as Roll-back/Retry, Pair of Self-Checking Processors (PSP) [11] and TMR.

3. BOSS OPERATING SYSTEM

BOSS is a real-time operating system developed by FHG-FIRST. The BIRD (Bi-Spectral Infrared Detection) satellite [14], designed for early detection of fires, uses BOSS as its multiple-computer control operating system. BOSS has also been applied in several other projects, and future utilizations include CubeSat satellites [15] and robotics in space [13].

BOSS design has been driven by reducing software complexity as a means to achieving dependability, as complexity is the cause of most development faults. The system had several parts validated by formal verification. It was developed using object-oriented programming with C++ and it has been ported to several platforms as PowerPC, x86 and Atmel AVR. There is also available an on-top-of Linux porting, primarily used for early testing.

BOSS supports fault tolerance in hardware redundant systems, by including a middleware layer which carries out transparent communications between nodes, using the publisher-subscriber protocol. A message object can be sent locally to the network, using a string as message subject. Receiving messages must specify which subject they are expected to receive from. Threads are usually consumers of receiving messages, by attaching to mail box objects. The middleware also supports message marshaling and the elimination of duplicate messages, based on a message identification number. This work uses a middleware implementation based on broadcast communications but a unicast version is also available.

4. FAULT TOLERANCE FRAMEWORK

This Section describes our thread model, the basic features of the tolerance framework which was integrated to the BOSS operating system to support application level fault tolerance, and how this framework is applied.

Fault tolerance can be applied to several layers of software, as at the operating system level, function/method level, object level or process level. Our work applies fault tolerance techniques to the thread level, but targeting only application threads, as operating system threads are supposed to be more robust

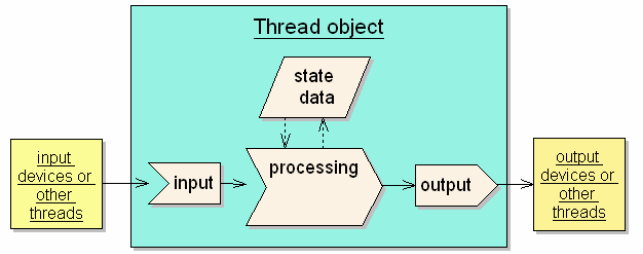

Figure 1. Model for FT threads.

Our purpose is presenting a general description of the framework fault tolerance capacities and how they are employed by the application programs.

4.1 Thread Model

Figure 1 shows the thread model required for fault-tolerant threads. The thread to be made fault-tolerant runs in an infinite loop, reading from input devices or receiving input messages from other threads. After processing the inputs, an output is generated either by writing to an output device or sending a result message to other threads. The model supports both state threads and stateless threads. For state threads, the output result will depend both on the input data and on the previous state data.

An example of a candidate thread for fault tolerance implementation is presented in Figure 2.

```
class ExampleThread : public Thread {

  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg, 20>
                              incommingMessages;
public:
  ExampleThread(){ ... // init code}

  void run () {
    while(1) {
      recMsg = incommingMessages.receive();
      process();
      output();
    }
  }

  void process(){
    ... // uses msg data and state data
  }

  void output(){
    ... // prepares output message
    outMsg.send("exampleResult");
  }
};

ExampleThread myThread;
```

Figure 2. Example of application thread.

In BOSS, all application threads must inherit from the Thread class and implement the run virtual function, which defines the thread run-time behavior. In this example, *ExampleThread* runs cyclically, reading messages from an *IncommingMessageAdministrator* object, which consists of a mailbox for messages of the

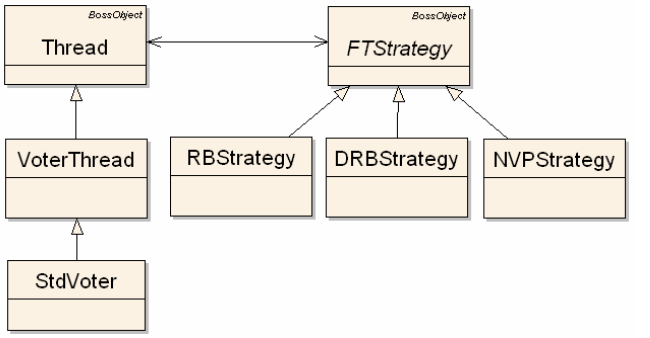

Figure 3. Fault tolerance framework.

Msg class. The *process* method is executed next, and implements some computing algorithm using data from the incoming message and possibly from an internal state. (attributes not shown). Finally the *output* method prepares the output message and sends it locally and over the network, using the string "exampleResult" as subject. The instantiation of thread objects is normally static, as shown in the last line of Figure 2. Dynamic memory allocation is avoided for performance reasons.

4.2 Fault-Tolerant Application

Figure 3 shows a class diagram of the FT framework. A fault-tolerant thread must define an *FTStrategy* object that will implement the fault-tolerant functionality. Presently, three FT strategies have been implemented: RB, DRB and NVP, but others can be developed and integrated to the framework. Some strategies, like DRB, involve message exchanges and coordination between multiples nodes, for defining roles, initializing global state and communicating results. All this work is performed by the FT framework, but some specific procedures must be defined by the application, as for instance, the acceptance test in RB and DRB. The degree of transparency depends on the strategy selection and the configuration. The *VoterThread* class implements application dependent majority voting and it is used in NVP to select the correct response among the NVP threads. *StdVoter* is a specialized voter that performs exact majority voting.

The modifications required to make an application thread fault-tolerant include:

- Instantiation and registration of an *FTStrategy* object that will implement the desired fault tolerance strategy, as RB, DRB and NVP.
- Execution of the *executeFT* method of the *FTStrategy* object after the thread activation.
- Implementation of application specific methods related to the selected fault tolerance strategy (as the acceptance test in RB and DRB). Some of them consist of new functionality but others will contain the code originally defined in the processing and output methods.

Figure 4 shows an example of fault tolerant implementation for *ExampleThread*, using the DRB strategy. The main differences between this version and the original code in Figure 2 are highlighted. A concrete *FTStrategy* is instantiated as a *DRBStrategy* (*myDRB*). In the class constructor, the maximum response time for execution is set to 20,000 microseconds and the *setFTStrategy* method is called, assigning the address of the *DRBSstrategy* to the *ftStrategy* pointer. In the run method, the original process and output methods are replaced by a call to the *executeFT* method of the *FTStrategy* class. This method is responsible for executing the particular strategy and for activating the application specific methods defined in the application thread, as for example, *variant1* (primary block) and *acceptanceTest*. Some of these methods correspond to original implementations, but others, like *variant2* (recovery block) and *saveCheckpoint* should be defined to allow the DRB strategy operation.

In this example, *ExampleThread* is stateless; otherwise *FTExampleThread* should also implement the methods *getState* and *setState*, to provide state initialization between the primary and the shadow nodes in DRB. None of these methods are necessary in the original version, as only one *ExampleThread* instance runs in a single node.

```
class FTExampleThread : public Thread {

  DRBStrategy myDRB;
  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg, 20>
                              incommingMessages;
public:

  FTExampleThread(){
    ... // init code
    myDRB.setMaxResponseTime(20000);
    setFTStrategy(&myDRB);
  }

  void run () {
    while(1) {
      recMsg = incommingMessages.receive();
      ftStrategy->executeFT();
    }
  }

  void variant1(){
    ... // same code of original process method
  }

  void sendResult(){
    ... // same code of original output method
  }
  // to be defined
  void variant2(){ ...}
  void saveCheckpoint(){ ... }
  void restoreCheckpoint(){...}
  bool acceptanceTest(){...}
};
```

Figure 4. Example of FT application thread.

4.3 Application Specific Entities

Each FT strategy instantiation and usage demands the definition of strategy attributes and application-specific behavior. These requirements are summarized in Tables 1 and 2.

Table 1 represents requirements for multiple version software and Table 2 for single version software. The fault tolerance strategies in Table 2 use the same *FTStrategy* objects of RB, DRB and NVP, but do not implement their full functionality, as several methods are not defined and so they present a default implementation. For example, the default implementation *save/restoreCheckpoint* is doing nothing and for *acceptanceTest* is returning true (success).

Table 1. Multiple version strategies requirements.

Definition Requirements		RB	DRB	NVP
Entity	Type			
FT Strategy	object	RBStrategy	DRBStrategy	NVPStrategy
Response time	parameter	Yes	Yes	Yes
variant 1	method	Yes	Yes	Yes
variant 2	method	Yes	Yes	Yes
variant 3	method	-	-	Yes
saveCheckpoint	method	Yes	Yes	-
restoreCheckpoint	method	Yes	Yes	-
acceptanceTest	method	Yes	Yes	-
sendResult	method	Yes	Yes	Yes
onFailure	method	Optional	Optional	Optional
Voter Thread	object	-	-	Yes
getState	method	-	state threads only	state threads only
setState	method	-	state threads only	state threads only

Table 2. **Single version strategies requirements**.

Definition Requirements		Restart	Rollback/ Retry	PSP	TMR
Entity	Type				
FT Strategy	object	RBStrategy	RBStrategy	DRBStrategy	NVPStrategy
Response time	param.	Yes	Yes	Yes	Yes
variant 1	method	Yes	Yes	Yes	Yes
variant 2	method	-	-	-	-
variant 3	method	-	-	-	-
saveCheckpoint	method	-	Yes	Yes	-
restoreCheckpoint	method	-	Yes	Yes	-
acceptanceTest	method	-	Yes	Yes	-
sendResult	method	Yes	Yes	Yes	Yes
onFailure	method	Optional	Optional	Optional	Optional
Voter Thread	object	-	-	-	Yes
getState	method	-	-	state threads only	state threads only
setState	method	-	-	state threads only	state threads only

The simpler FT strategy in Table 2 is the *restart* strategy. In this technique only one variant is defined, and the acceptance test is not implemented. Therefore, the only possible error detection is deadline expiration. A deadline is obtained by adding the response time parameter to the thread activation time. Roll-back/Retry can be implemented as a single version simplification of the RB strategy. In this case, only one real variant is defined, and the body of the *variant2* should contain a call to the *variant1* method. In a similar way, PSP is implemented with the DRB strategy and TMR with the NVP strategy.

Voter threads are needed when using TMR or NVP. In the general case, a voter thread is application-specific and must implement virtual methods defined in the *VoterThread* class (see Figure 3). For exact majority voting using messages, a standard voter which compares results byte by byte is provided (*StdVoter* class). It is also possible to define if all replicated voters will send their outputs or if only a master voter will do it. The definition of the master voter in a coordinated voting is performed by the FT framework.

5. AOP IMPLEMENTATION

Our goal is to use AOP to modularize all fault-tolerant code, keeping the original code intact. The process of generating the executable code using this approach is explained as follows. The operating system, already integrated to the fault tolerant framework, is compiled and an OS library is generated. Abstract strategy aspects are developed for each FT strategy in the system. They define virtual pointcuts and standard advices used for all related concrete strategy aspects. A concrete aspect must be defined for advising each future fault-tolerant application thread, as it will be discussed later. The weaving process using AspectC++ [4] generates a fault-tolerant application that is eventually compiled and linked to the OS code.

```
aspect DRBStrategyAbstract {

  pointcut virtual DRBClass() = 0;
  pointcut virtual ProcessMethod() = 0;
  pointcut virtual OutputMethod() = 0;
  int maxResponseTime;

  advice DRBClass(): slice class {
    private:
      DRBStrategy myDRB;
  };
  pointcut constr() = construction(DRBClass());

  advice constr(): after(){
    tjp->target()->myDRB.setMaxResponseTime( maxResponseTime );
    tjp->target()->setFTStrategy(&(tjp->target()->myDRB));
  }

  pointcut compute()= call(ProcessMethod()) && target( DRBClass() ) && !within( "% ...::variant%(...)"
);

  advice compute(): around(){
    tjp->target()->ftStrategy->executeFT();
  }

  pointcut result()= call(OutputMethod()) && target( DRBClass() ) && !within( "% ...::sendResult(...)"
);

  advice result(): around(){
  }
};
```

Figure 5. DRB strategy abstract aspect.

We will present an example of how to apply AOP to make the *ExampleThread* of Figure 2 fault tolerant, using the DRB strategy. Figure 5 shows the abstract aspect related to the DRB strategy. Initially this aspect declares three virtual pointcuts which will be defined in the concrete aspect. These pointcuts represent the thread class under modification *(DRBClass*) and the original methods for processing (*ProcessMethod*) and output (*OutputMethod*). The integer *maxResponseTime* will keep the maximum response time for execution, which must be defined by the concrete aspect. The introduction of the *DRBStrategy* object definition is carried out using the AspectC++ slice construction, which is used to extend the static structure of a program. The initialization of this object, as well as its registration, is performed by the advice with the *constr* pointcut, similarly as done in the constructor code of the non-AOP version in Figure 4.

The *compute* pointcut defines a condition in which the processing method of the non-FT thread is called in the original code. The around advice related to this pointcut will replace this call with the activation of the *executeFT* method of the *FTStrategy* class. Similarly, the *result* pointcut defines a condition in which the output method of the non-FT thread is called in the original code. The around advice related to this pointcut will just suppress this call, as the activation of the thread output is going to be controlled by the *FTStrategy* object.

The concrete aspect to make the *ExampleThread* fault-tolerant is shown in Figure 6. The aspect inherits from the *DRBStrategyAbstract* aspect and initially defines its virtual pointcuts. In this case, the target thread is "ExampleThread", the processing method is "process" and the output method is "output", as seen in Figure 2.

```
aspect DRBExampleConcrete: public
                          DRBStrategyAbstract {

  pointcut DRBClass() = "ExampleThread";
  pointcut ProcessMethod()= "% ...::process()";
  pointcut OutputMethod() = "% ...::output()";

  DRBExampleConcrete(){
    maxResponseTime = 20000;
  }

  advice DRBClass() : slice class {
    public:
      void variant1(){ process(); }
      void sendResult(){output(); }

      // methods to be defined
      void variant2(){ ... }
      void saveCheckpoint(){ ... }
      void restoreCheckpoint(){...}
      bool acceptanceTest(){...}
  }
};
```

Figure 6. **Concrete DRB aspect example**.

The maximum response time for this strategy is set to 20.000 microseconds in the aspect constructor, by initializing a base abstract variable. After that, several methods are introduced in the target thread. The virtual method *variant1* is responsible for running the primary block in DRB, and in this case it must execute the original processing of *ExampleThread*. Similarly, the virtual method *sendResult* must call the original *output* method. Here it should be noticed that the calls to *process* and *output* in the introduced methods *variant1* and *sendResult* will not trigger the execution of the advices defined by the *compute* and *result* pointcuts in the *DRBStrategyAbstract* aspect, because the scope pointcut function *within* is being applied. Finally, the application specific methods are defined for this strategy, as *variant2* (recovery block) and *saveCheckpoint*. After the process of weaving, the new *ExampleThread* code becomes functionally equivalent as the non-AOP version of Figure 4.

An alternative approach for code generation is applying aspects to connect the fault tolerance framework to the operating system. In this case, the weaving process now applies also to the original operating system code. The FT framework is injected in the OS by an aspect. In our framework this aspect has to modify the original *Thread* class to introduce a pointer to an *FTStrategy* object and the virtual functions shown in Tables 1 and 2. Using this approach it is possible to reduce the code size for non-FT implementations and also to apply aspects for fault tolerance and other concerns at the operating system level, as logging, synchronization and middleware customization [1]. However, there is no modification in abstract and concrete FT aspects used at the application level.

6. EVALUATION

In this Section we present two case studies. The first one was used to validate the AOP implementation by verifying if it produces the correct behavior. The second case study was meant to measure the AOP costs. Finally, we discuss of advantages and disadvantages of the AOP approach.

6.1 Radar Filtering System

We applied the AOP approach to a radar filtering system. In this application a portable PC running an on-top-of Linux implementation of BOSS simulates a radar system and generates detection data of several planes periodically. The data generation includes simulated errors in bearing and distance, typical of this kind of equipment. This data is received by three PowerPC 823 boards (80 MHz clock) running an application that filters the planes' position, using an alpha-beta filter, and calculates the planes' course and speed. The results are sent back to the portable PC, where they are displayed. Our development environment consisted of a PC host running Linux Fedora 3, GNU gcc 3.2.3 as cross-compiler and AspectC++ 1.0pre3 as aspect weaver [4].

Initially we had a single node, non-FT version of the filtering application, and then we applied AOP to create an FT application using TMR. The FT configuration is shown in the UML deployment diagram of Figure 7, where nodes are represented by cubes and application threads by rectangles. All Filter threads send their results with "unvoted_data" as subject, which are received by voter threads in the PowerPC boards. In this particular configuration, only the master voter thread sends the final results to the Display thread in the PC.

The AOP implementation was compared to a non-AOP implementation and presented the correct behavior (timing and values) at all times.

6.2 Performance and Memory Footprint

We now describe an experiment to test and compare performance and footprint of the fault tolerant AOP implementations with respect to non-AOP implementations.

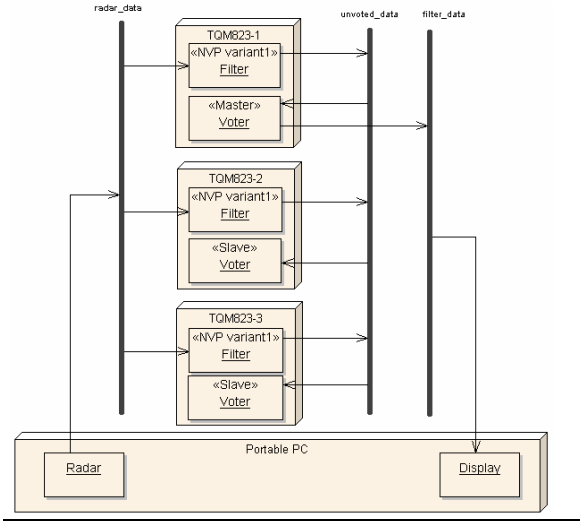

Figure 7. TMR strategy configuration.

In order to evaluate performance, we decided to measure the CPU utilization on the target board, in a test case with 10 RB threads with small processing times and high activation frequencies. A single processor configuration using only local messages was selected, aiming to eliminate the amount of CPU time spent on sending and receiving messages over the communication network, which in our case is not affected by the usage of aspect-oriented artifacts.

A PowerPC 823 board executes an application with one Sender thread, which periodically generates an array of 5 integer random numbers and sends them using a local message. Then, the 10 identical Receiver threads obtain this message and sort the numbers using the insertion sort algorithm as the primary block and the selection sort algorithm as the recovery block. The acceptance test is executed by checking if the integers are in the correct order within the result array. The Receiver threads output consists of preparing an output message with the sorting results but this message is not sent so as to decrease the CPU utilization.

In this experiment three different software versions were evaluated:

- Non-FT implementation – In this version, the Receiver thread does not use any FT strategy and just sorts the arrays of integers with the insertion sort algorithm. The Receiver thread code is similar to the one presented in Figure 2.
- FT implementation – This version uses a Receiver thread that follows a standard object-oriented RB fault tolerance implementation, similar to the one presented in Figure 4.
- FT-AOP implementation – In this version the Receiver thread is the same of the non-FT implementation and the RB fault tolerance in injected by AOP, as described in Section 5.

Figures 8 and 9 show performance results of these three implementations for different compiler optimizations and thread activation periods. Figure 8 shows CPU utilizations with no compiler optimization (-O0 option in gcc) for activation periods of 5 and 10 milliseconds. The non-FT implementation has the lower CPU utilization as expected, because it does not involve any fault tolerant control and fault detection mechanisms, as the

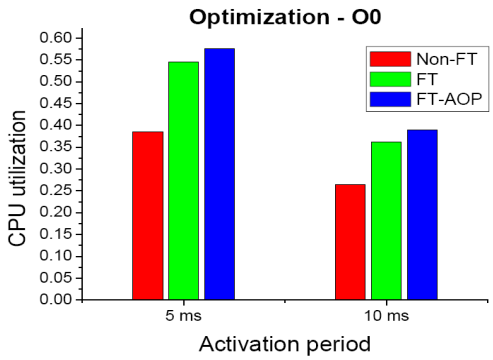

Figure 8. CPU utilization with no optimization.

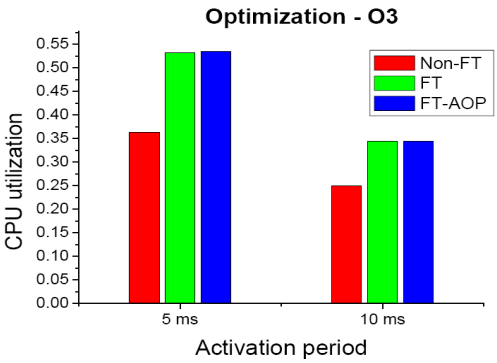

Figure 9. CPU utilization with maximum optimization.

acceptance test in RB. Comparing fault tolerant implementations, we verified that, in this test case, the AOP version has a higher CPU utilization of about 6%, for un-optimized programs. On the other hand, if maximum performance optimization is performed by the compiler (-O3 in gcc), as shown in Figure 9, this difference in performance drops to less than 0.5%. That performance difference is directly dependent on the application thread activation. Consequently, we conclude that the performance difference measured in this experiment is greater than in real applications, as they usually have longer processing times and larger activation periods.

Table 3 shows program memory sizes in bytes used for code (text), data and uninitialized data (bss) for each implementation, considering -O3 compiler optimization. The FT-AOP implementation uses more 260 bytes for code and 16 bytes for bss, comparing to the normal FT implementation. The increase in code size is caused by inlining after and around advices that make use of the AspectC++ joinpoint data structure. The extra bytes in bss are related to the creation of aspect objects and pointers.

Table 3. Memory footprint results.

version	text	data	bss	total
Non-FT	65,987	6,384	207,152	279,523
FT	66,783	6,424	207,792	280,999
FT-AOP	67,043	6,424	207,808	281,275

Based on this experiment we conclude that the utilization of AOP for application-level fault tolerance implementation in an embedded real-time application does not imply a significant increase in run-time or memory footprint.

6.3 Discussion

We used AOP to modularize all fault tolerant code at the application thread level, keeping the original code intact. The advantages of this approach are:

- Less prone to errors in porting a non-FT system to a FT one. The task of changing an existing system to introduce fault tolerance capabilities may insert software faults in the original code. Using AOP the original code is preserved.
- The programmer can initially write applications without fault tolerance in mind, and concentrate his efforts in the development of the functional code. Using AOP, fault tolerance can be applied in a second stage, after validating the core functionality.
- Facilitates the evaluation and comparison of several FT configurations, as the developer may easily select what set of application threads will be made fault tolerant and on which strategy.
- Contributes to product line development, as single or redundant systems may be generated by introducing or not fault tolerant aspects.
- Contributes to code reuse, because the same functional code can be applied in other projects with different dependability requirements.

Using this approach we noticed that the base code remains oblivious to the fault tolerant concern, but on the other hand, the aspect code is very dependent on the base code it applies to. This fact is related to the nature of fault tolerance domain, where for each FT instantiation we may need to define deadlines, extra fault detection, alternative procedures, checkpoints, state coordination, voting specifications, and so on. For that reason, concrete aspects are normally heterogeneous and can target only one application thread. However, depending on the characteristics of the application process and the selected fault tolerant strategy, less application specific code may be needed. In our opinion, completely transparent fault tolerance injection is very hard to achieve.

7. RELATED WORK

The work in [7] proposed the use of aspect-orientation in real-time systems for distribution, timeliness and dependability domains. An example of the application for each domain is given, using CORBA in a logging application as test case. This work does not address any fault tolerance mechanism other than execution time surveillance.

Herrero et al [8] designed a replication model called JReplica, based on AO techniques, to allow the specification of fault tolerance behavior and requirements. This model works at design time, using UML. Only passive replication is supported. The model includes new entities to intercept input and output messages and interact with replication aspects.

In [9] the authors feel that it is hard and potentially dangerous to separate concurrency control and failure management from the main application. They prepared a case study based in transactions and conclude that homogeneous aspects yields poor performance and the functional code keeps semantically coupled with the non-functional part (the aspect). Besides, any maintenance in one should trigger a modification on the other.

The work with more similarity with ours is described in [2]. They address the question of whether AOP can provide a base for implementing fault tolerant mechanisms in non-distributed environments. For the implementation of the recovery cache mechanism, AspectC++ had to be extended with the "set" joinpoint [3]. This work presents examples of aspects for single node computing, as time-redundant execution, assertions and Recovery Blocks. However, the FT mechanisms are applied at the method level while ours is applied at the thread level.

Detailed quantification of AspectC++ run-time and memory costs have been presented in [12]. In this work, extra cycles and memory consumption are measured for each aspect-oriented feature and also for a refactored and extended AOP version of the ECOS operating system kernel. In contrast, our work measures the AOP performance in a demanding fault-tolerant application, based on CPU utilization

8. CONCLUSION

In this paper we described and evaluated an approach for the application of AOP to the development of real-time embedded fault-tolerant software. Our work differs from previous works for injecting fault tolerance at the application thread level, and considering several fault tolerant mechanisms and redundant hardware/software configurations.

We conclude that AOP is very useful in this domain because it reduces efforts and errors in making a legacy system fault-tolerant, simplifies system development by allowing the validation of the functional part in advance, facilitates the evaluation and comparison of various FT configurations, and contributes to product line development and code reuse.

Future work will include the application of AOP for operating system fault tolerance and additional run-time overhead measurements in cycles or time.

9. ACKNOWLEDGMENTS

This work has been supported by the Portuguese Foundation for Science and Technology.

10. REFERENCES

[1] F. Afonso, C. Silva, S. Montenegro and A. Tavares. Applying Aspects to a Real-Time Embedded Operating System. *In Proceedings of the 6th Workshop on Aspects, Components and Patterns for Infra-structure Software - ACP4IS*, Vancouver, Canada, 2007.

[2] R. Alexandersson, P. Ohman and M. Ivarson. Aspect Oriented Soft-ware Implemented Node Level Fault Tolerance. In *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications -SEA*, Phoenix, Arizona, USA, 2005.

[3] R. Alexandersson and P. Ohman. Implementing Fault Tolerance Using Aspect Oriented Programming. *LNCS Dependable Computing*, vol. 4746/2007, Springer-Verlag, 2007.

[4] AspectC++ project homepage: http://www.aspectc.org.

[5] A. Avizienis, J.-C. Laprie and B. Randell. Fundamental Concepts of Dependability. *Technical Report 739*, Department of Computing Science, University of Newcastle upon Tyne, 2001.

[6] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Proceedings of FTCS-8*, pp. 3-9, Toulouse, France, 1978.

[7] A. Gal, O. Spinczyk and W. S-Preikschat. On Aspect-Orientation in Distributed Real-time Dependable Systems. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems - WORDS*, pp. 261-267, 2002.

[8] J. Herrero, F. Sánchez and M. Toro. Fault Tolerance as an Aspect using JReplica. In *Proceedings of the 8th IEEE Workshop on Future Trends in Distributed Computing Systems - FTDCS*, pp. 201-207, 2001.

[9] J. Kienzle and R. Guerraoui. AOP: Does it Make Sense? The Case of Concurrency and Failures. In *Proceedings of the 16th European Conference on Object Oriented Programming*, pp. 37-61, 2002.

[10] K. Kim and O. Welch. Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Transactions on Computers*, vol. 38, Nº 5, pp. 626-636, 1989.

[11] K. Kim. Toward Integration of Major Design Techniques for Real-Time Fault-Tolerant Computer Systems. In *Journal of Integrated De-sign and Process Science*, vol. 6, issue 1, pp. 83-101, 2002.

[12] Lohmann D. et al. A Quantitative Analysis of Aspects in the eCos Kernel. *In Proceedings of EusoSys2006*, Leuven, Belgium, 2006.

[13] P. Massa, et al. HiPeRCAR: the High Performance Resilient Computer for Autonomous Robotics. In *Proceedings of Data Systems on Aerospace - DASIA*, Berlin, Germany, 2006.

[14] S. Montenegro and F. Zolzky. BOSS /EVERCONTROL OS/Middleware Target Ultra High Dependability. In *Proceedings of Data Systems on Aerospace -DASIA*, Edinburgh, Scotland, 2005.

[15] S. Montenegro, K. Briess and H. Kayal. Dependable Software (BOSS) for the BEESat Pico Satellite. In *Proceedings of Data Systems on Aerospace - DASIA*, Berlin, Germany, 2006.

[16] D.K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice-Hall, Inc., 1996.

[17] B. Randell System Structure for Software Fault Tolerance. *IEEE Trans. Software Engineering*, vol. 1, no.2, pp. 220-232, June 1975.

[18] B. Randell, P. Lee and P.C. Treleaven. Reliability Issues in Computing System Design. *ACM computing Surveys*, vol. 10, issue 2, pp. 123-165, 1978.