Automatic Detection and Masking of Non-Atomic Exception Handling

Christof FETZER Karin HÖGSTEDT Pascal FELBER

AT&T Labs — Research
Florham Park, NJ, USA
{christof, karin}@research.att.com

Institut EURECOM
Sophia Antipolis, France
felber@eurecom.fr

A short version of this article appeared in the Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN), June, 2003.

Abstract

The development of robust software is a difficult undertaking, and is becoming increasingly more important as applications grow larger and more complex. Although modern programming languages such as C++ and Java provide sophisticated exception handling mechanisms to detect and correct runtime error conditions, exception handling code must still be programmed with care to preserve application consistency. In particular, exception handling is only effective if the premature termination of a method due to an exception does not leave an object in an inconsistent state. We address this issue by introducing the notion of *failure atomicity* in the context of exceptions. We propose practical techniques to *automatically* detect and mask the *non-atomic exception handling* situations encountered during program execution. These techniques can be applied to applications written in various programming languages that support exceptions. We perform experimental evaluation on both C++ and Java applications to demonstrate the effectiveness of our techniques and measure the overhead that they introduce.

Index Terms

D.2 **Software Engineering**, D.2.4 Software/Program Verification, D.2.4.g *Reliability*, D.2.5 Testing and Debugging, D.2.5.f *Error handling and recovery*, D.2.5.h *Reliability*, D.2.5.r *Testing tools*

I. Introduction

Developing robust software is a challenging, yet essential, task. A robust program has to be able to detect and recover from a variety of faults such as the temporary disconnection of communication links, resource exhaustion, and memory corruption. Ideally, robust software has to be able to recover from faults without substantially increasing the code complexity. An increase in code complexity can increase the probability of design and coding faults and can thus decrease the robustness of the software. Of course, code complexity and robustness are not antonymous if one can avoid or remove design and coding errors in the error handling code.

Language-level *exception handling* mechanisms allow programmers to handle errors with only one test per block of code. In programming languages without exception handling, such as C, programmers have to check for error return codes after each function call. The use of exception handling mechanisms can thus simplify the development of robust programs.

Although the use of exceptions simplifies the detection of failures, the elegance of language-level exception handling mechanisms might lead to the neglect of recovery issues [1]. The premature exit of a method due to an exception might leave an object in an inconsistent state.

If this inconsistency is not resolved in the error handling code, it might prevent a later recovery, and thus decrease the robustness of the program. In this article, we show how to automatically detect and correct such state inconsistencies.

1) Problem Description: Modern programming languages, such as C++ and Java, provide explicit exception handling support. When a semantic constraint is violated or when some exceptional error condition occurs, an exception is *thrown*. This causes a non-local transfer of control from the point where the exception occurred to a point, specified by the programmer, where the exception is *caught*. An exception that is not caught in a method is implicitly propagated to the calling method. The use of exceptions is a powerful mechanism that separates functional code from the error handling code and allows a clean path for error propagation (see Figures 1 and 2). It facilitates the development of applications that are robust and dependable by design.

```
1  int err = openFile ();
2  if ( err == NO_FILE) { ... }
3  else if ( err == CANT_OPEN) { ... }
4  else {          // OK, file open
5    err = readFile ();
6    if ( err == CANT_READ) { ... }
7    else {        // OK, read it
8      err = testFormat ();
9      if ( err == UNKNOWN) { ... }
10     else {      // OK, keep going
11       ... } } }
```

Fig. 1. Error handling based on return codes.

```
1  try {
2    openFile ();
3    readFile ();
4    testFormat ();
5  }
6  catch(NO_FILE e) { ... }
7  catch(CANT_OPEN e) { ... }
8  catch(CANT_READ e) { ... }
9  catch(UNKNOWN e) { ... }
```

Fig. 2. Error handling based on exceptions.

Exception handling code must however be programmed carefully to ensure that the application is in a consistent state after catching an exception. Recovery is often based on retrying failed methods. Before retrying, the program might first try to correct the runtime error condition to increase the probability of success. However, for a retry to succeed, a failed method also has to leave changed objects in a consistent state. Consistency is ensured if any modification performed by the method prior to the occurrence of the exception is reverted before the exception is propagated to the calling method. This behavior is hard to implement because, when catching exceptions, a programmer has to consider all possible places where an exception might be thrown, and has to make sure that none of these exceptions can cause a state inconsistency.

In this article, we address the challenging problem of ensuring that failed methods always

leave objects in a consistent state after throwing an exception. We classify methods as either *failure atomic* or *failure non-atomic*, depending on whether they do or do not preserve state consistency, respectively. Informally, we say that the *exception handling is atomic* if it ensures failure atomicity. Otherwise, we say that *exception handling is non-atomic*. In general, it is impossible to automatically determine if a given method is failure atomic in all executions. Our main objectives are thus to find mechanisms that help identify failure non-atomic methods, and to develop techniques to automatically transform these methods into failure atomic methods.

2) Approach: In order to identify failure non-atomic methods, we propose a system that systematically tests and validates the exception handling code of applications. Our system automatically injects both declared (i.e., anticipated) and undeclared (i.e., unexpected) exceptions at runtime, and evaluates if the exception handling code ensures failure atomicity. It notifies the programmer of any failure non-atomic method, as in many situations minor code modifications are sufficient to implement failure atomicity (e.g., changing the order of some instructions, or introducing temporary variables). When manual modifications are not possible or not desired, our system can in many cases automatically generate wrappers to render a given method failure atomic with the use of checkpointing and rollback mechanisms.

Our infrastructure for detecting and masking non-atomic exception handling comes in two flavors, which support the C++ and Java programming languages, respectively. The C++ version is optimized for performance, but requires access to the application's source code. The Java version is less efficient, as it uses a combination of load-time and runtime reflection mechanisms, but it also works with applications for which source code is not available.

Note that our current two infrastructures come with a few restrictions that we plan to relax in the future. First, so far we only support one form of consistency (which we call failure atomicity) and in particular, we do not yet support the definition of application specific consistency specifications. Second, our definition of consistency does not yet consider external side effects (e.g., IO) that might need to be compensated for after an exception.

3) Contributions: The contribution of this article is twofold. First, we introduce and formalize the failure atomicity property in the context of exception-based error handling. Second, we introduce novel techniques for *automatically* detecting and masking non-atomic exception handling. These techniques can be applied to both C++ and Java applications, and do not always require access to the application's source code. Furthermore, we present experimental results

that demonstrate the effectiveness and the performance overhead of our techniques.

The organization of this article is as follows: In Section II, we first discuss related work. Section III introduces the failure atomicity problem, and Section IV presents our approach for detecting and masking failure non-atomic methods. In Section V we discuss the implementation details of our system, and Section VI elaborates on the performance of our C++ and Java infrastructures. Section VII concludes the article.

II. Related Work

Exception handling has been investigated for several decades. Goodenough [2] proposed to add explicit programming language constructs for exception handling in 1975, and Melliar-Smith and Randell [3] introduced the combination of recovery blocks [4] and exceptions to improve the error handling of programs in 1977.

Exception handling is still actively investigated. For example, a complete issue of ACM SIGAda Ada Letters [5] was recently dedicated to exception handling, and a 2001 Springer LNCS book addresses advances in exception handling [6]. One of the major issues addressed by researchers is a better separation of functional code and exception handling code. Recent studies have proposed to combine exception handling and reflection to increase this division [7], or to use aspect-oriented programming for reducing the amount of code related to exception handling [8].

Although the goal of exception handling code is to increase the robustness of programs, it has been noted by Cristian in [9] that exception handling code is more likely to contain software bugs (called *exception errors* [10]) than any other part of an application. This can be explained intuitively by a couple of factors. First, exceptions introduce significant complexity in the application's control flow, depending on their type and the point where they are thrown. Second, exception handling code is difficult to test because it is executed only rarely and it may be triggered by a wide range of different error conditions. Furthermore, one study [10] has shown that reducing the occurrence of exception handling failures would eradicate a significant proportion of security vulnerabilities. Therefore, eliminating exception failures would not only lead to more robust programs, but also more secure programs.

Several approaches have been proposed to address the issue of exception errors [10]: code reviews, dependability cases, group collaboration, design diversity, and testing. Testing typically

results in less coverage for the exception handling code than for the functional code [9]. The effectiveness of dependability cases, design diversity, and collaboration for reducing exception handling errors has been studied in [10]. In this article we introduce a novel approach based on exception injection to address certain kinds of exception errors. We do not consider our approach as a replacement of other approaches; we rather believe that it complements other techniques such as dependability cases and collaboration. The advantages of our approach lie essentially in its highly automated operation and fast detection of functions that contain certain exception errors.

The robustness of programs can be evaluated using fault injection techniques [11]. There exist software-implemented, hardware-implemented, and simulation-based fault injectors. Our tool performs software-implemented fault injections. Software-implemented fault injectors have been investigated for various types of failures, such as memory corruption [12], [13], invalid arguments [14], or both [15]. There are also various techniques for injecting faults. Some tools, such as FERRARI [16] and Xception [17], inject faults without modifying the applications. Other tools, such as DOCTOR [18], modify the application at compile time, and yet others during runtime. Our tool injects faults in the form of exceptions, by modifying the application either at compile time or at load time. Unlike FIG [19], which tests the error handling of applications by returning error codes to system calls, our tool only injects application-level exceptions.

Our tool does not only evaluate the robustness of programs by performing exception injections, but it also automatically corrects the problems discovered by the fault injections. The automatic wrapping of shared libraries based on injection results has been previously demonstrated in [20]. In this article, we address different types of failures (exception handling vs. invalid arguments) and hence, we use different fault injection and wrapping techniques. Our tool automatically modifies problematic methods by saving the state of specific objects and restoring it upon failure to *roll back* to a consistent state. This behavior is very similar to transactional systems, which ensure atomicity by “undoing” the actions of a transaction that does not complete successfully [21]. While transactions are traditionally a feature of database systems, several programming languages have been designed or extended to directly incorporate various levels of transactional support, independently of any underlying database management system and with little runtime overhead. Much research has been conducted by the parallel computing community [22] on the concurrency control aspects (i.e., transaction isolation) when multiple processors simultaneously access shared

data and a few programming language actually implement abortable transactions. Recent work of Harris and Fraser [23] adds support for lightweight transactions to the Java programming language. We consider this work as complementary to our own research in the sense that lightweight transactions could advantageously replace our checkpointing techniques to automatically mask the failure non-atomic behavior of specific objects and methods. Transactional mechanisms would further improve the robustness and alleviate some of the limitations of our fault recovery, as our system does not explicitly address issues such of concurrency control (i.e.,isolation). Integrating the techniques proposed by Harris and Fraser would however require new tools, as memory needs to be accessed through special functions that cannot be implemented with our current source and object code transformation techniques. As it is true in our current system, one would still need to determine which functions would need to be wrapped in transactions and to reduce the overhead of masking, one would still prefer to fix the source code instead of using automatic masking with the help of transactions.

III. Problem Description and Motivation

Robust software has to be able to detect and recover from failures that might occur at runtime. One way of performing failure recovery is to take advantage of the exception handling mechanism that is provided in many programming languages. Using this mechanism, a method can signal to its caller that it has encountered a failure, be it memory depletion or an unexpected result of a calculation, by throwing an exception. The exception can then be caught by the caller, which provides the programmer with an opportunity to recover from the failure and consequently to increase the robustness of the application.

Failure recovery is however likely to fail, unless extreme care is taken during the programming of the exception handling code. Due to the incomplete execution of the method that threw the exception, one or more objects might be in inconsistent states. Unless consistent states are restored, the application might crash or terminate with an incorrect result.

Example 1: Consider method vector::alloc *in Figure 3. This method allocates a new array with* nb_entries *elements of type T. It stores the new address of the array and the number of entries in the instance variables* x *and* size*, respectively. The call to* new *on line 9 throws an exception if there is not enough memory available or if the constructor of T throws an exception. If this happens, the object upon which the method is called will be in an inconsistent state, since*

```
1  class T;
2
3  class vector {
4    T *x;
5    int size;
6  public:
7    void alloc(int nb_entries) {
8      size = nb_entries;
9      x = new T[nb_entries];
10   }
11 };
12
13 class vector2D {
14   T *x, *y;
15   int size;
16 public:
17   void alloc(int nb_entries) {
18     x = new T[nb_entries];
19     y = new T[nb_entries];
20     size = nb_entries;
21   }
22 };
```

Fig. 3. Sample classes with failure non-atomic methods.

```
1  void vector::alloc(int nb_entries) {
2    x = new T[nb_entries];
3    size = nb_entries;
4  }
5
6  void vector2D::alloc(int nb_entries) {
7    T *tmp_x = NULL, *tmp_y = NULL;
8    try {
9      tmp_x = new T[nb_entries];
10     tmp_y = new T[nb_entries];
11     size = nb_entries;
12     x = tmp_x;
13     y = tmp_y;
14   } catch (...) {
15     delete tmp_x;
16     delete tmp_y;
17   }
18 }
```

Fig. 4. Methods of Figure 3 transformed to be failure atomic.

the value of size *has already been set to its new value while* x *still holds its old value. This is a common mistake, even for experienced programmers: the sketched problem was discovered by our fault-injection tool in a C++ class that we checked.*

This problem can easily be solved by swapping lines 8 and 9, as shown in Figure 4: if the new *on line 2 throws an exception, the state of the object upon which this method is called remains unaffected. Consequently, the object is in a consistent state when the exception is caught, and the caller can safely recover and reissue the invocation (after having reclaimed additional memory or with reduced capacity requirements).*

Another problem that our tool discovered in an existing C++ class is illustrated in method vector2D::alloc *of Figure 3. This method performs two operations that can throw exceptions. If only one of the operations succeeds, the object can be left in an inconsistent state. To solve this problem, one can use the "update a temporary and swap" idiom as shown in Figure 4 (note that it is safe to call* delete *on a null pointer).*

The system presented in this article helps programmers detect which methods might leave an object in an inconsistent state when an exception is thrown. Our system can also automatically revert an object back to a consistent state by automating the "checkpoint, execute, and roll-back on exception" idiom, if the programmer so desires. This is further explained in Section IV and V.

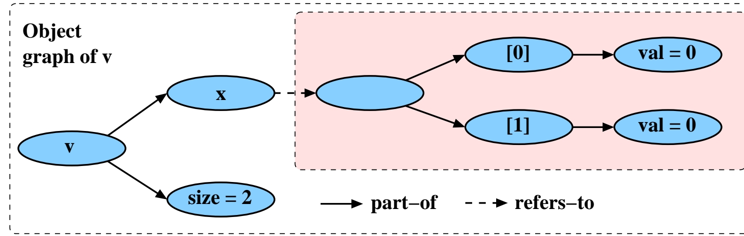


Fig. 5. A sample object graph of an object v of class *vector* in Figure 3.

Before describing our system in more detail, we formally introduce the important notions of *object graphs*, *failure non-atomic methods*, and *failure atomic methods*.

Definition 1: *An object graph is a graph whose nodes are objects or instances of primitive data types, and whose edges are part-of and refers-to relationships, with a single distinguished root object. Nodes are labeled with the associated variable name (if it exists) and, for all basic data types except pointers, with the value of the variable. The instance variables of an object are referred to by the part-of relation. Each non-null pointer has a refers-to edge to the node which is referenced by the pointer. The object graph of an object o is the graph rooted at o.*

This definition corresponds to the notion of object graphs traditionally used in object-oriented languages to delimit the scope of deep copying. Note that multiple pointers may refer to the same node. It follows that the object graph of an object o holds the state of o (i.e., the values of its fields), as well as the state of all objects referenced directly or transitively by o. As an example, the object graph of a sample object v of class *vector* defined in Figure 3 is depicted in Figure 5. For the purpose of the illustration, we let class T have a single instance variable *val* of type *int*.

We say that two object graphs o_1 and o_2 are *equal* if the two root nodes have the same labels and there exists a one-to-one mapping between each child c_1 of one of the root nodes reachable via relation R (part-of or refers-to) and a child c_2 of the other root node reachable via the same relation R such that the object graphs rooted at c_1 and c_2 are equal.

Definition 2: *Let C be a class, and m a method of class C. Given an invocation of method m on an object o with arguments $\alpha_1, \cdots, \alpha_n$, we define the* pre-invocation object graph *for that invocation of m as the object graph consisting of a virtual root node with, as children, the $n + 1$ object graphs of o and the arguments $\alpha_1, \cdots, \alpha_n$ taken just before m is executed. The*

post-invocation object graph *for an invocation of* m *that returns with an exception is defined similarly, except that the children of the virtual root node are taken just after* m *returns with the exception.*

Definition 3: *Let* C *be a class. A method* m *of class* C *is* failure atomic *if for all objects* o *of class* C *and all executions of* m *on* o *with arguments* $\alpha_1, \cdots, \alpha_m$ *that returns with an exception, the pre-invocation and post-invocation object graphs are equal. A method is* failure non-atomic *if it is not failure atomic.*

The failure atomicity definition is modelled along the lines of transactional semantics: an exception causes an abort of the enclosing transaction (i.e., method call) and a roll back to the initial state at the start of the transaction. Alternate exception semantics might also make sense. For example, our definition of failure atomicity property does not enforce the pointers in the pre-invocation and post-invocation object graphs to be the same. Some applications may require such strong semantics (e.g., for pointers to shared objects), but in general we cannot decide automatically which form of equivalence a program needs. Because every two "pointer equivalent" object graphs are also equivalent according to our definition, we only test for the weaker definition. This might result in a slight increase in the number of false negatives, i.e., we do not detect when a method should enforce pointer equivalence but only enforces value equivalence.

On the other hand, our definition of failure atomicity might be too conservative for some applications. For instance, a class using lazy data allocation or caching might modify some member variables (e.g., an "allocated" or a "dirty" flag) without changing the actual content of its objects and graph equivalence comparisons would report inconsistencies. As automatic tools cannot easily detect this kind of behavior, our approach is to provide the developer with simple tools to ignore such problematic methods using a Web-based interface. Given the wide range of application designs and requirements, one could also extend our system to allow the programmer to specify customizable failure semantics based on the pre-invocation and post-invocation object graphs.

It can be shown that the failure atomicity problem is undecidable, i.e., one cannot accurately determine if a method is failure atomic, by a simple reduction to the halting problem. Therefore, we rather focus on approaches to obtain good approximations of the problem, as discussed in the next section.

IV. APPROACH

Our approach to identify and transform failure non-atomic methods consists of two phases: a *detection* and a *masking* phase. The detection phase uses automated injection of exceptions to identify failure non-atomic methods, and the masking phase transforms failure non-atomic methods into failure atomic methods.

A. Detection Phase

The goal of the detection phase is to determine which methods are failure non-atomic. However, as we mentioned earlier, the failure atomicity problem is not computable. To address this issue, we relax the accuracy requirement. Our system classifies a method as non-atomic only if it has proof that the method is non-atomic, i.e., the system found an execution in which the method is non-atomic (a counter example). If the system cannot find such an execution, it classifies it as atomic. Note that to prove that a method m is atomic one would need to show that m is atomic in all executions—a task impossible to compute.

We use automated exception-injection experiments to classify methods as either failure atomic or failure non-atomic. Although static program analysis may be possible for certain applications and programming languages, we chose to only explore the more widely applicable—but also more time-consuming—approach of dynamic program analysis. In general, the techniques proposed in this article would benefit from being combined with static analysis, when applicable, to speed up the detection phase.

The automated exception-injection experiments are run on test programs or end-user applications that call the methods we want to investigate. We automatically transform the code of these programs to inject exceptions at specific points of their execution. These exception injector programs are then run to generate a list of failure non-atomic methods to be used as input to the masking phase. This process corresponds to steps 1 through 3 in Figure 6.

Step 1: To create an exception injector program P_I from a program P, we first find all methods that are potentially called during the execution of P. To that end, we instrument all the classes that are defined and used by P and we execute the program to discover which methods are invoked at runtime. For each of these method, we identify the exceptions that may be thrown. In Java this includes all the exceptions *declared* as part of the method's signature and that can be determined using Java's reflection mechanisms. Additionally, we assume that each method may

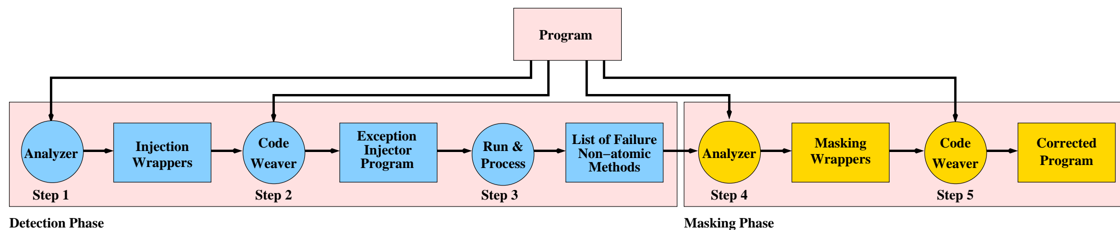


Fig. 6. We automatically transform applications to inject exceptions in their execution, and we use the experimental results to correct the applications.

throw a *runtime exception*, which represents exceptions that may be thrown during the normal operation of the virtual machine but do not need to be declared or caught (e.g., invalid argument or null pointer). In C++, we only inject one type of exception because most methods do not explicitly declare the exceptions they throw.

The *Analyzer* tool then creates an *injection wrapper* for each called method as follows: Assume that method m declares exceptions of types $E_1, \cdots, E_k$ and may also throw runtime exceptions $E_{k+1}, \cdots, E_n$. The *Analyzer* creates an injection wrapper *inj_wrapper_m* for m, which either throws one of these exceptions, or calls method m. In the injection wrapper of m, there are n potential injection points (see lines 2-5 in Figure 7). We determine whether to throw an exception at any of these injection points using a global counter (*Point*), incremented every time the control flow reaches one of these potential injection point; an exception is injected when the counter reaches a preset threshold value (*InjectionPoint*).

```
1  return_type inj_wrapper_m(α1, ···, αm) throw (E1, ···, Ek) {
2    if (++Point == InjectionPoint ) throw E1();
3    if (++Point == InjectionPoint ) throw E2();
4    ...
5    if (++Point == InjectionPoint ) throw En();
6    pre_invocation_graph = create_object_graph ( this , α1, ···, αm);
7    try {
8      return m (...);
9    } catch (...) {
10     if ( pre_invocation_graph .equal( this , α1, ···, αm))
11       mark("m", "atomic", InjectionPoint ); // Atomic in this call
12     else
13       mark("m", "non−atomic", InjectionPoint );
14     throw;
15   }
16 }
```

Fig. 7. Pseudo-code for the injection wrapper of method m (detection phase).

```
1 return_type atomic_m(α1, ···, αm) {
2   pre_invocation_graph = create_object_graph ( this , α1, ···, αm);
3   try {
4     return m (...);
5   } catch (...) {
6     pre_invocation_graph . replace ( this , α1, ···, αm);
7     throw;    // Rethrow exception
8   }
9 }
```

Fig. 8. Pseudo-code for the atomicity wrapper of method m (masking phase).

Step 2: After the *Analyzer* has created the injection wrappers for all methods called by P, the *Code Weaver* tool makes sure that these wrappers, as opposed to the original methods, are called. Modifications can be performed on the program's source files (source code transformation), or by directly instrumenting the application's compiled bytecode (low-level transformation). The result of this transformation is an exception injector program P_I.
Step 3: Once the exception injector program P_I is created, we execute it repeatedly. We increment the threshold value *InjectionPoint* before each execution to inject an exception at a different point in each run. (As we shall see in Section IV-C, injections can be performed in a more efficient manner.) Each wrapper intercepts all exceptions and checks if the wrapped method is failure non-atomic before propagating the exception to the caller.

To determine whether a method m is failure non-atomic, the injection wrapper (Figure 7) takes a snapshot of the pre-invocation object graph (analogous to cloning or performing a deep copy of the target objects). If m returns with an exception, the wrapper catches the exception and compares the saved snapshot with the value of the post-invocation object graph. If they are are equal, we mark the method as failure atomic; otherwise, we mark it as failure non-atomic.

Since different injections may result in different classifications for a method, we classify a method m as failure atomic if and only if it is never marked as failure non-atomic, i.e., if and only if for each injection the pre-invocation and post-invocation object graphs are equal. The output of this phase is a list of all the failure non-atomic methods detected in the original program.

B. Masking Phase

The goal of the masking phase is to transform the failure non-atomic methods identified during the injection phase into failure atomic methods. By doing so, the resulting program becomes more robust, since the incomplete execution of a method due to an exception does not result in an inconsistent program state.

Note that the masking phase is optional, i.e., a programmer might prefer to fix a program manually instead of using automatic masking. Similar to programming level transaction mechanisms, we assume that rolling back the state to a previous checkpoint results in an internally consistent state. However, the program still needs to deal with the effects of non-abortable actions, such

as the sending a message via a socket. Typically, the code that catches the exception needs to handle this issue.

The masking phase consists of steps 4 and 5 in Figure 6.

Step 4: The failure non-atomic methods are automatically transformed into equivalent failure atomic methods. The *Analyzer* performs this task by generating an *atomicity wrapper atomic_m* for each method m in the list of failure non-atomic methods provided by the detection phase. Before the invocation of the target method, the atomicity wrapper (Figure 8) takes a snapshot of the pre-invocation object graph. If m returns with an exception, the wrapper restores the state of the objects using the snapshot and re-throws the exception. This wrapper exhibits failure atomic behavior to its callers.

Step 5: After the *Analyzer* has generated an atomicity wrapper for each of the methods that should be transformed, the *Code Weaver* transforms the original program P into an equivalent (corrected) program P_C by replacing all calls to these methods by calls to their atomicity wrappers. This process is similar to Step 2.

The implementation details of both the detection and the masking phases for C++ and Java are discussed in Section V.

C. Speedup of the Detection Phase

In a program P with N method calls and K exception classes, our system injects $N * K$ exceptions. The approach sketched in Section IV-A performs this task by executing the program $N * K$ times. The time complexity of Step 3 (in terms of number of methods called) is hence $O(N^2K)$. For large values of N, this process can be time-consuming in comparison to the normal running time of $O(N)$. Therefore, we implemented a variant that can reduce the complexity to $O(NK)$.

To reduce the execution time of Step 3, we checkpoint the application state before injecting an exception by forking a child process. The child process injects the exception while the parent process waits for the child process to terminate. The child process needs to check the atomicity of all methods that are in execution at the time the exception is injected, i.e., of all methods for which there exists a return address on the stack at the time of the exception injection.

If an application can tolerate the injected exception, i.e., the injected exception does not terminate the application, the child process will run to completion. Hence, for such applications

the time complexity is still quadratic even if the system is using application state checkpointing. To reduce the time complexity, the system terminates a child process as soon as the exception stops propagating. Note that the failure atomicity property is restricted to methods returning with an exception, i.e., the systems stops learning about method failure atomicity after an injected exception has finished propagating.

A child process can thus terminate in two distinct manners after an exception has been injected. First, if the exception propagates to the topmost wrapper on the call stack of the child process, the wrapper terminates the child immediately after determining the failure atomicity classification of the wrapped method. Second, any wrapper that detects that an injected exception has stopped propagating (i.e., a wrapped method that was called before the exception was injected but that returned normally after the injection) terminates the child process. When using this approach, Step 3 now has a time complexity in $O(NK)$. The performance gain resulting from this optimization in C++ is discussed in Section VI.

D. To Wrap or Not To Wrap

There are situations where a failure non-atomic method should *not* be wrapped during the masking phase. First, although very unlikely, the failure non-atomic behavior of a method might have been intended by the programmer. Since transforming a method to become failure atomic changes its semantics, the transformation might *cause* an incorrect result or crash, instead of avoiding it. To deal with this situation, our system provides an easy-to-use Web interface that allows the programmer to indicate which of the methods classified as failure non-atomic should not be transformed.

Second, some failure non-atomic methods can easily be rendered failure atomic *manually*, e.g., by swapping lines of code or by using temporary variables as shown in Figure 4. In that case, the programmer most likely would prefer to rewrite the method himself, since the resulting code is likely to be more efficient. After the programmer corrects these methods, he can re-run the detection phase to validate his modifications.

Third and most interestingly, a method might exhibit failure non-atomic behavior only because the methods it calls are failure non-atomic. We call such methods *dependent failure non-atomic* methods:

Definition 4: A dependent failure non-atomic *method is a failure non-atomic method that*

would be failure atomic if all the methods that it calls (directly or indirectly) were failure atomic. All other failure non-atomic methods are pure failure non-atomic *methods.*

During the execution of the corrected program (produced by the masking phase), all methods called by a dependent failure non-atomic method m will exhibit failure atomic behavior. Thus, by definition, method m is no longer failure non-atomic and it is not necessary to wrap it. Therefore, distinguishing between pure and dependent failure non-atomic methods can help us significantly improve the performance of the corrected program.

To distinguish dependent from pure failure non-atomic methods, we examine the order in which failure non-atomic methods were reported during exception propagation for each run of the exception injector program (Step 3 in Figure 6). If there exists a run in which method m is the first method to be reported failure non-atomic, then m is pure failure non-atomic. Indeed, any failure non-atomic method called by m would be detected and reported before m because of the way exceptions propagate from callee to caller (see Figure 7). If there exists no such run, then m is dependent failure non-atomic.

E. Limitations

The approach that we use to detect and mask failure non-atomic methods has some limitations. First, our approach does not address external side effects of methods: we do not attempt to detect inconsistent external state changes caused by an exception, nor do we try to mask such external state inconsistencies. The proper handling of external state changes in the face of exceptions remains the task of the code that catches such exceptions.

Second, since our (non fork-based) exception injector executes an instrumented program repeatedly to inject exceptions at each possible injection point, the program must behave deterministically to ensure good code coverage. If the program is not deterministic, then our system might inject exceptions in only a subset of the possible injection points and overlook some failure non-atomic methods that would have been detected otherwise. In contrast, a fork-based injector (See Section IV-C) makes sure that an exception is injected at all injection points of a single execution and should therefore be used for non-deterministic applications. One should note that our system only tests the methods that are called by the original program, and the completeness of the results depends on how extensively the program exercises its own objects and methods.

Third, our system does not explicitly deal with concurrent accesses in multi-threaded programs. Such concurrency issues could be addressed using a transaction mechanism instead of a checkpointing mechanism to mask failure non-atomicity.

Finally, our system only injects exceptions in method calls (including object constructors and destructors), and exclusively for exception classes that have a default constructor. It does not take into account failures such as arithmetic errors (e.g., division by zero) or asynchronous events (e.g., signals, thread interruption). Such failures are extremely challenging to account for, as they are often completely unpredictable. In that sense, our approach implements "best-effort" detection mechanisms and may fail to report failure non-atomic methods (false negatives). This limitation also applies for failure non-atomic methods that *by chance* exhibit atomic behavior for all injected exceptions.

Conversely, our system can also produce "false positives" by classifying as failure non-atomic a method that exhibits seemingly inconsistent behavior that was intended by the programmer, or by injecting exceptions in methods that are unlikely to fail such as accessor methods that simply returns the value of a member variable. In practice, however, it is often difficult to determine whether a method is by definition "exception-free" as all the runtime conditions that might lead directly or indirectly to application level exceptions are not necessarily known. For instance, there is always a risk of a method invocation producing a stack overflow. To completely avoid false positives, we need to determine not only whether a method can throw an exception, but also which exceptions it can throw. For instance, the recovery code of a method might correctly handle stack overflows but exhibit failure non-atomic behavior when injecting an arbitrary exception type that cannot occur in practice.

V. IMPLEMENTATIONS

We have investigated two approaches for implementing our system: source code and bytecode program transformation techniques. The first approach requires access to the source code of a program, while the second does not, but instead relies on the strong typing of the language and the large amount of structural information embedded in the bytecode. Both kind of transformations can be aided by the use of aspect oriented programming [24], which allow programmers to easily capture and integrate crosscutting concerns in their applications.

A. Source Code Transformation

We have implemented a first prototype of our system that performs source code transformations to inject and mask non-atomic exception handling in C++ applications. We use the C/C++ interpreter CINT [25] to parse the source code of a given program and gather the type information necessary to generate the checkpointing code and wrappers for each method during the detection phase, as well as the atomicity wrappers for pure failure non-atomic methods during the masking phase. The wrappers are implemented as *aspects* for the AspectC++ [26] aspect-oriented language extension for C++, which are then woven with the source code of the program in such a way that each call to a method m calls instead the wrapper of m. We also generate, for each class, a function *create_object_graph* to checkpoint the state of an instance of that class, and a function *replace* used during the masking phase to restore the state of a previously checkpointed object. The exception injector program is executed repeatedly to inject exceptions at all possible injection points (for the given program input) and the results of online atomicity checks are written out to log files by the injection wrappers. These log files are then processed offline to classify each method.

Due to restrictions of C++ and the tools we are using, our implementation has a few limitations. First, CINT does not support templates and ignores exception specifications. Using a better C++ parsing tool would solve this limitation.

Second, checkpointing C++ objects is not trivial. In particular, C++ allows pointer manipulations that make it hard, in some situations, to discover the complete object graph of an object at runtime. While there exist techniques to address this problem, they can be prohibitively expensive: in particular, we experimented with a variant where the masking wrapper saves the state of the complete process by forking a child process before executing a call to a purely non-atomic method m. If an exception occurs during the execution of m, the state is rolled back by terminating the parent process and letting the child process throw the same exception as the parent without actually executing m. This approach has the advantage that the recovered post-invocation object graph is not only equal but also identical to the pre-invocation object graph, and it allows us to also recover the state of all global and static variables. Using fork to checkpoint processes has been proposed and implemented previously, e.g., in [27]. Unlike our approach, most checkpointing systems save the state of a process to disk. The authors

of [28] proposed mechanisms to save the content of a process in memory and showed that application checkpointing can be performed very efficiently. As these mechanisms were not available on our target operating systems, we had to rely on process forking instead. Fork uses copy-on-write to optimize the copying, which might be advantageous to checkpoint large objects that experience few updates during the execution of an non-atomic method. While our initial performance measurements showed that process forking introduces significant runtime overhead as we shall see in Section VI, we believe based on the results of [28] that a very well tuned process checkpointing mechanism might provide acceptable performance.

Third, unlike Java, C++ does not enforce thrown exceptions to be declared as part of the method's signature. Hence, for completeness, the C++ exception injector might have to inject a wide range of different exception types in application that do not declare exceptions. This problem can be solved using source code analysis or through automated fault injection experiments.

Fourth, one needs to clean up the memory that is implicitly discarded when rolling back to an object checkpoint. To do so, our tool adds an automatic reference counting mechanism to objects. However, this mechanism only works for acyclic pointer structures. For cyclic pointer structures, one can use an off-the-shelf C++ garbage collector. Using the fork mechanism to checkpoint objects solves this problem too.

B. Bytecode Transformation

With languages that offer adequate reflection mechanisms, it is possible to add functionality to an application without having access to its source code, by applying transformations on the compiled bytecode. We have followed this second approach in the Java version of our infrastructure for detecting and masking non-atomic exception handling. The key properties that make this approach possible are Java's strong typing, and the comprehensive structural information embedded in the bytecode and exposed by the virtual machine.

To inject and mask failures in Java classes, we have developed a tool, called the Java Wrapper Generator (JWG), which uses load-time reflection to transparently insert pre- and post-filters to any method of a Java class. These generic filters allow developers to add crosscutting functionality (as with aspect-oriented programming) to *compiled* Java code in a transparent manner. Filters are attached to specific methods at the time the class is loaded by the Java virtual machine, by using bytecode instrumentation techniques based on the BCEL bytecode engineering library [29].

Filters can be installed at the level of the application, individual classes, instances, or methods. They can modify the behavior of a method by catching and throwing exceptions, bypassing execution of the active method, or modifying incoming and outgoing parameters.

The Java implementation of our framework works along the same lines as its C++ counterpart, with just a few notable differences. Wrappers are attached to the application at load-time, by instrumenting the classes' bytecode. These wrappers have been programmed to be generic, i.e., they work with any class; they obtain type information about classes, methods, parameters, and exceptions at runtime using Java's built-in reflection mechanisms. The methods that checkpoint and restore the state of an object are also generic; they essentially perform a deep copy of the object's state using Java's reflection and serialization mechanisms.

A major limitation with Java bytecode transformation is that a small set of core Java classes (e.g., strings, integers) cannot be instrumented dynamically. This limitation applies to all systems that perform Java bytecode transformations, and is not specific to our implementation. It can be overcome by instrumenting the bytecode of core classes offline and replacing their default implementations by the instrumented versions.

In addition, the Java version of our framework does not incorporate the fork-based optimizations previously mentioned because of the lack of adequate mechanisms to spawn child processes in Java.

VI. Experimental Results

To validate our exception injection tool, we first developed a set of synthetic "benchmark" applications in C++ and Java. These benchmarks are functionally identical in both languages, and contain the various combinations of (pure/dependent) failure (non-)atomic methods that may be encountered in real applications. We used these benchmarks to make sure that our system correctly detects failure non-atomic methods during the detection phase, and effectively masks them during the masking phase. These applications were used for the code coverage experiments and some performance experiments presented in the section.

We then performed stress tests and assessed the robustness of some legacy applications. For that purpose, we tested two widely-used Java libraries implementing regular expressions [30] and collections [31]. Such libraries are basic building blocks of numerous other applications and are thus expected to be robust. We also tested Self★ [32], a component-based framework in C++

that we are currently developing. We ran experiments with several applications that use Self★ to help us detect failure non-atomic methods and improve the robustness of the framework.

	Application	# Classes	# Methods	Total # Injections
C++	*adaptorChain*	16	44	10122
	stdQ	19	74	9585
	xml2Ctcp	5	19	6513
	xml2Cviasc1	23	102	12135
	xml2Cviasc2	23	89	13959
	xml2xml1	18	70	8068
Java	*CircularList*	8	58	5912
	Dynarray	7	50	2528
	HashedMap	10	40	3271
	HashedSet	8	32	1149
	LLMap	10	41	7543
	LinkedBuffer	8	38	2737
	LinkedList	9	62	7500
	RBMap	11	55	7133
	RBTree	9	51	8056
	regexp	4	32	1015

TABLE I

C++ AND JAVA APPLICATION STATISTICS.

Table I lists the number of classes and methods in the applications that we used for our experimental evaluation, together with the total number of exceptions injected during the detection phase (note that this value corresponds to the number of method and constructor calls during the execution of the test programs). We ran separate experiments for each individual application; however, because of the inheritance relationships between classes and the reuse of methods, some classes have been tested in several of the experiments.

Experiments were conducted following the methodology described in Section IV: we generated an exception injector program for each application, and ran it once for each method and constructor call in the original program, injecting one exception per run. The C++ experiments were run on a 866 MHz Pentium 3 Linux machine (kernel 2.4.18) with 512 MB of memory and the Java tests were run using Java 1.4.1 on a 2 GHz Pentium 4 Linux machine (kernel 2.4.18) with 512 MB of memory.

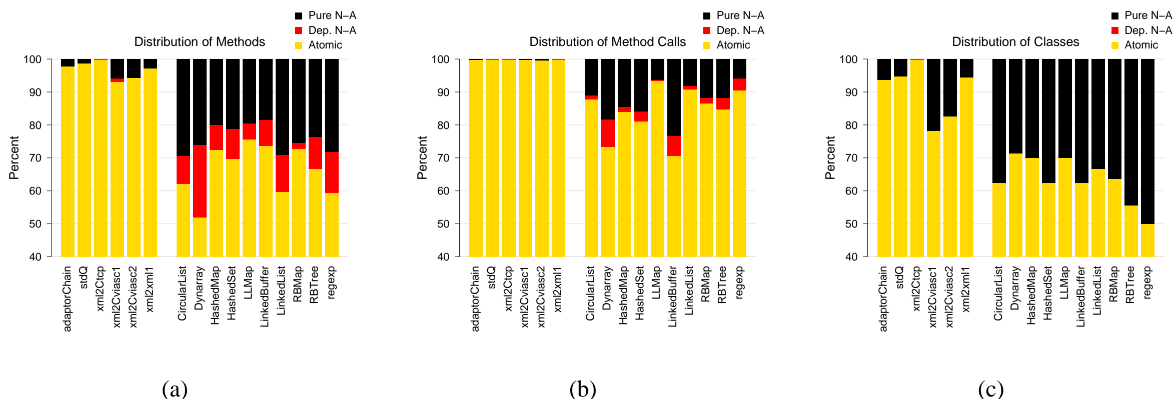
A. Fault Injection Results

We first computed the proportion of the methods defined and used in our test applications that are failure atomic, dependent failure non-atomic, and pure failure non-atomic. The results, presented in Table II and Figure 9(a), show that the proportion of “problematic” methods, i.e.,

	Application	# Methods reported as: Atomic	Dep. N-A	Pure N-A	# Method calls reported as: Atomic	Dep. N-A	Pure N-A	# Classes reported as: Atomic	Dep. N-A	Pure N-A
C++	*adaptorChain*	43	0	1	6331	0	12	15	0	1
	stdQ	73	0	1	5944	0	8	18	0	1
	xml2Ctcp	18	0	0	3982	0	0	5	0	0
	xml2Cviasc1	95	1	6	7261	1	21	18	0	5
	xml2Cviasc2	84	0	5	8360	0	30	19	0	4
	xml2xml1	68	0	2	4887	0	3	17	0	1
Java	*CircularList*	36	5	17	4894	72	612	5	0	3
	Dynarray	26	11	13	1844	211	459	5	0	2
	HashedMap	29	3	8	2542	45	441	7	0	3
	HashedSet	23	3	7	874	34	171	5	0	3
	LLMap	31	2	8	6821	30	455	7	0	3
	LinkedBuffer	28	3	7	1919	163	634	5	0	3
	LinkedList	37	7	18	6463	79	580	6	0	3
	RBMap	40	1	14	6339	120	863	7	0	4
	RBTree	34	5	12	6949	288	958	5	0	4
	regexp	19	4	9	872	35	57	2	0	2

TABLE II

FAILURE-ATOMICITY PROPERTIES OF EACH C++ AND JAVA APPLICATION, IN TERMS OF NUMBER OF METHODS, METHOD CALLS, AND CLASSES.



(a) (b) (c)

Fig. 9. Failure-atomicity properties of each C++ and Java application, in terms of percentage of (a) methods, (b) method calls, and (c) classes. Absolute numbers are given in Table II.

those that are pure failure non-atomic, remains pretty small with C++. This may indicate that the tested Self★ applications have been programmed carefully, with failure atomicity in mind. In contrast, the Java results exhibit a different trend. The proportion of pure failure non-atomic is pretty high, as it averages $20-25\%$ in the considered applications. The proportion of dependent failure non-atomic methods is smaller, but still significant. These relatively high numbers can be explained by the fact that the Java code, unlike its C++ counterpart, contains a significant number of trivial methods, such as accessors, which are unlikely to throw an exception (although this

can still happen in some situations, e.g., when the call stack cannot be further extended in low memory conditions). By explicitly disabling exception injections in these methods, we observed a reduction of 50% of the failure non-atomic methods. Furthermore, we observed that the average depth of the calling stack in the Java programs was higher than in the C++ applications; this can explain why we observe more dependent failure non-atomic methods in Java than in C++. While different programming language encourage different programming styles, the proportion of failure non-atomic methods depends in priority on the nature, the design, and the implementation of the considered application.

Table II and Figure 9(b) show the results of the method classification weighted by the number of invocations to each method. Results show that failure non-atomic methods are called proportionally less frequently than failure atomic methods. This trend may be explained by the fact that bugs in methods frequently called are more likely to have been discovered and fixed by the developer. Since problems in methods that are infrequently called are harder to detect during normal operation, our tool is quite valuable in helping a programmer find the remaining bugs in a program. For example, the pure failure non-atomic methods of the "xml2Cviasc" applications are called very rarely, and would probably not have been discovered without the automated exception injections of our system.

As another illustration of the usefulness of our system, the output of the fault injector allowed us to discover some severe bugs in rarely executed error-handling code of the *LinkedList* Java application. In particular, when adding a set of elements to a list, if any of the new elements but the first is invalid (e.g., it has a null value), an "illegal element" exception is correctly thrown and the elements are not inserted; however, the variable storing the size of the list is incremented, which leaves the list in an inconsistent state that soon triggers a fatal error. This kind of bug is hard to detect without automated exception injections. Using just trivial code modifications, and by identifying methods that never throw exceptions (see Section IV-D), we managed to reduce the number of pure failure non-atomic methods in that application from 18 (representing 7.8% of the calls) to 3 (less than 0.2% of the calls). This further demonstrates the importance of carefully testing error handling code.

Finally, Table II and Figure 9(c) show the proportion of the classes in our test applications that are failure atomic (i.e., only contain failure-atomic methods), pure failure non-atomic (i.e., contain at least one pure failure non-atomic method), and dependent failure non-atomic (i.e., all

other classes). The results clearly demonstrate that failure non-atomic methods are not confined in just a few classes, but spread across a significant proportion of the classes (up to 25% for C++ tests, and from 30 to 50% for Java tests).

B. Completeness of Failure Atomicity Detection

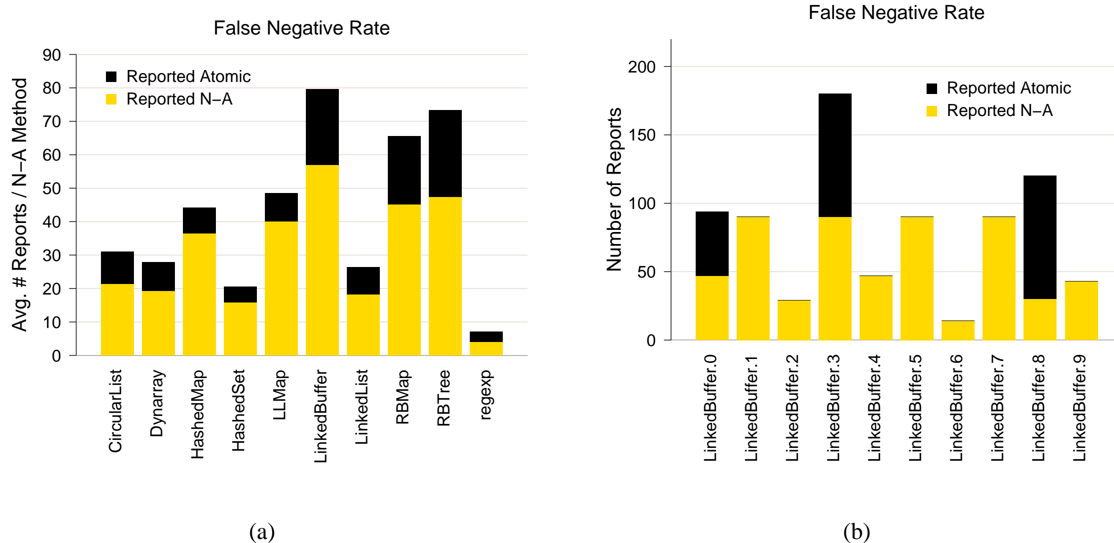
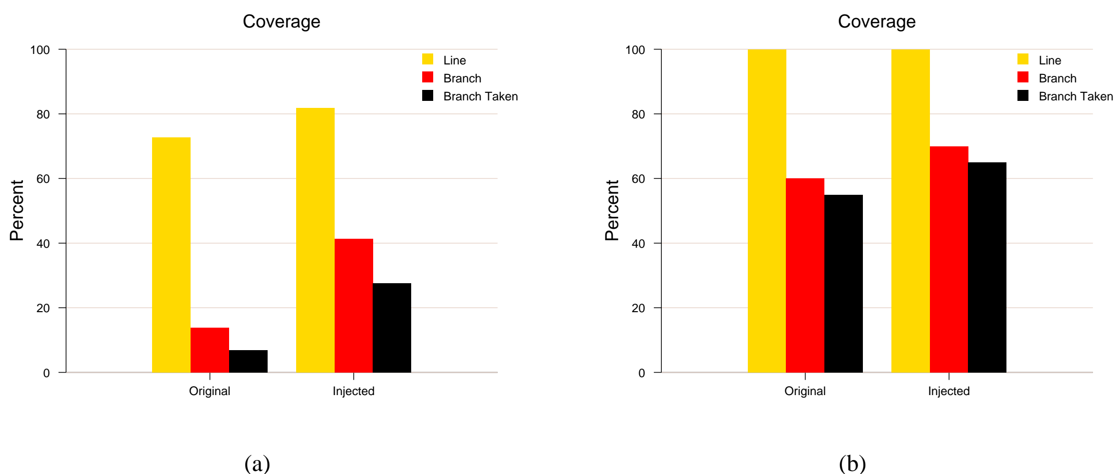


Fig. 10. Completeness of the failure atomicity detection mechanisms, in terms of false negatives, for (a) all Java applications, and (b) each of the failure non-atomic method of the “LinkedBuffer” class.

To assess the completeness of our failure atomicity detection mechanisms, we computed the ratio of “false negatives” for each failure non-atomic method f, i.e., the ratio of the number of injections that reported failure atomicity to the total number of injections into f. A high ratio of false negatives indicates that the method only exhibits failure non-atomic behavior in rare situations and is hard to detect, and our mechanisms are hence greatly useful. On the other hand, low ratios mean that the method behaves in a failure non-atomic manner most of the time and it is thus critical to fix it. Note that as long as there is at least one report of failure non-atomicity, our system correctly classifies that function as failure non-atomic, i.e., these are false negatives for a single run but not for the classification of the method. This experiment gives an indication of the number of injections necessary to discover that a method is failure non-atomic.

Figure 10(a) shows the average, over all failure non-atomic methods of each Java application, of the number of runs where the method was correctly reported as failure non-atomic, and the number of runs where it was incorrectly reported as failure atomic. We observe ratios of false negative in the range of 20% to 30%, i.e., failure non-atomic methods may prevent consistent error recovery in 3 out of 4 runs. Figure 10(b) details the failure atomicity reports for each of the failure non-atomic methods of the application with the highest average number of injections per method, i.e., *LinkedBuffer*. We notice that most methods consistently behave non-atomically, while false negatives are confined in only 3 of the methods with ratios from 50% to 75%; these failure non-atomic methods may have been easily overlooked without systematic exception injections.

C. Code Coverage



(a) (b)

Fig. 11. Injecting exceptions can increase the line coverage by exercising *catch* blocks. Even for code with 100% line coverage, injecting exceptions can help to improve the coverage of branches that are evaluated and branches that are taken.

As previously mentioned, our system helps detecting problems in code and methods that are executed only rarely. In particular, exception injections allows us to systematically test exception handling code and to increase the line and branch coverage by executing the statements in *catch* blocks. Moreover, even for programs without *catch* blocks, branch coverage can be increased with exception injections because of the "hidden" branches that are only exercised during the propagation of exceptions. These branches are taken, for example, during the unwinding of the

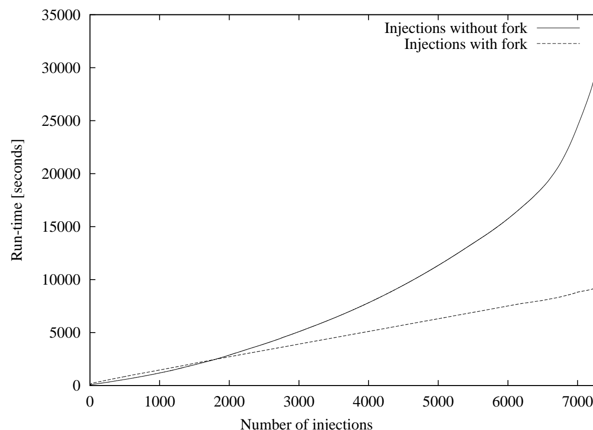


Fig. 12. Time needed to inject x exceptions at the first x injection points of an application (*xml2Cviasc*).

stack to call the destructors of the objects that were allocated on the stack, and they are important because the programmer has to make sure that the program maintains failure atomicity for each branch.

We ran test coverage experiments to quantify the additional portions of code that can be tested by our system. Of course, the increase in coverage depends on many factors such as the test cases and the structure of the program and in particular, one cannot give guarantees on by how much the coverage will increase. Our experiments were conducted on two benchmark programs: the first one had a *catch* block, while the second one had none. To ensure a fair comparison of the coverage, these two programs were not transformed; instead, exceptions were injected via external methods. The first application shows an increase in line and branch coverage (Figure 11(a)). The second application had already 100% line coverage and, hence, shows no increase (Figure 11(b)). The branch coverage, however, is more than 10% higher with exception injections. Code coverage can thus strongly benefit from automated exception injections: not only does the line coverage increase for code with *catch* blocks, but also the branch coverage for code without catch blocks.

D. Speedup of Fork-based Injection

The performance of the Detection phase might not always be of great importance. However, the injection performance is important in situations such as when (1) there are only a limited number of machines that can be used for the injection experiments, (2) runs take a very long time, or (3) the software is changing rapidly and developers want fast feedback about the behavior of the current version of the software.

To evaluate the performance gain resulting from the fork-based optimization described in Section IV-C, we measured the time needed to inject exceptions in an application (*xml2Cviasc*), both with and without forking child processes. Figure 12 depicts the time to inject x exceptions by forking x child processes versus running the program x times. This measurement shows the linear increase in injection time when forking a child process to do the injection, versus the quadratic increase without a fork. Using a fork-based exception injector can thus be very useful for applications with large numbers of injections points.

E. Fault Masking Results

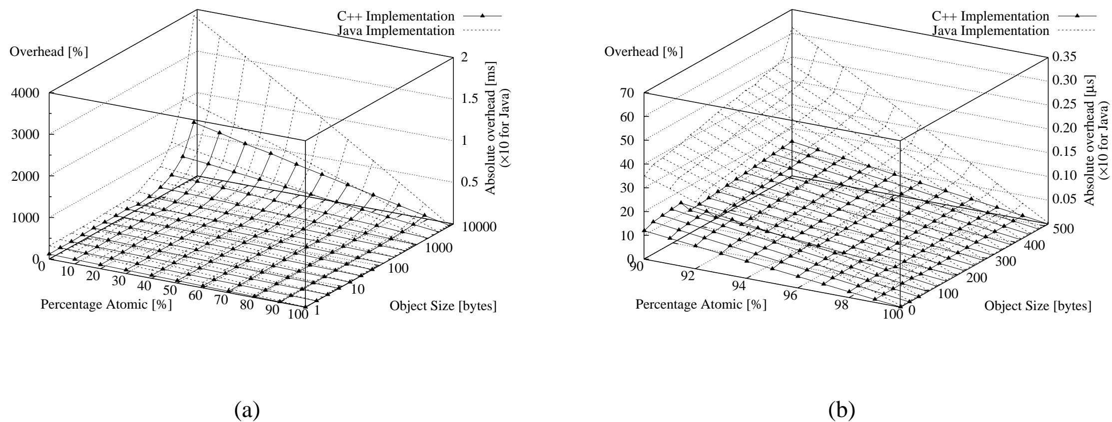


Fig. 13. (a) Performance overhead for masking a fixed-duration method in C++ ($\approx 0.5 \mu s$) and in Java ($\approx 5 \mu s$). (b) Zoom on the front-right corner of (a).

We measured the overhead of method masking in C++ and Java, as a function of the checkpointed object size and percentage of failure atomic method calls. Results are shown in Figure 13. Each data point is the median of 100 runs and the fixed processing time per method in the original

program was set to $0.5\mu s$ in C++, and $5\mu s$ in Java.[1] Exactly one byte in the object is read and written during each method execution. Note that the C++ overhead does not increase linearly with the object size because the function called to copy the state of an object uses byte copy for small objects and word copy for large objects. We observe a similar behavior with the Java overhead for objects larger than 500 bytes, most probably caused by the size increase and reallocation of the buffers used for object serialization.

Not surprisingly, the performance of our automated masking mechanism is highly dependent of the frequency of calls to the transformed methods. Obviously, we have to pay a higher performance penalty as the percentage of calls to the transformed methods increases. The overhead typically grows with the size of the checkpoints and, as there is no upper limit to the size of objects, this overhead cannot be bounded. We observe the same behavior in both the C++ and Java implementations, although the performance loss is more than one order of magnitude higher in the Java implementation. When analyzing the source of the overhead in Java, we observed that the generic filters inserted during load-time instrumentation added a fixed cost of around $6\mu s$ per masked method, and the reflection and serialization mechanisms used in our generic checkpointing wrappers had a variable overhead 5 to 10 times higher than when using a custom wrapper tailored for the method. This demonstrates that performance could be significantly improved by also performing source-code modifications in Java.

In the programs we investigated, we observed that the checkpoint sizes and the percentage of failure non-atomic method calls remain small. For example, the largest percentage of calls to failure non-atomic methods in our C++ applications was less than 0.4% (Figure 9(b)). In the Java programs, the pure non-atomic methods that we could not easily render failure atomic (by performing trivial modifications) accounted for less than 0.2% of the calls. Although the performance overhead seems to be high, we can obtain reasonable performance as long as the object sizes and the percentage of failure non-atomic method calls remain reasonably small, as shown in Figure 13(b).

The actual performance overhead of the masking mechanism clearly depends on the amount of work done by a method. The measurements shown in Figure 13 assumed a fixed amount of work independent of the size of the object. We also measured the overhead for a method that

[1]We use different values for the processing time because C++ is inherently faster than Java.

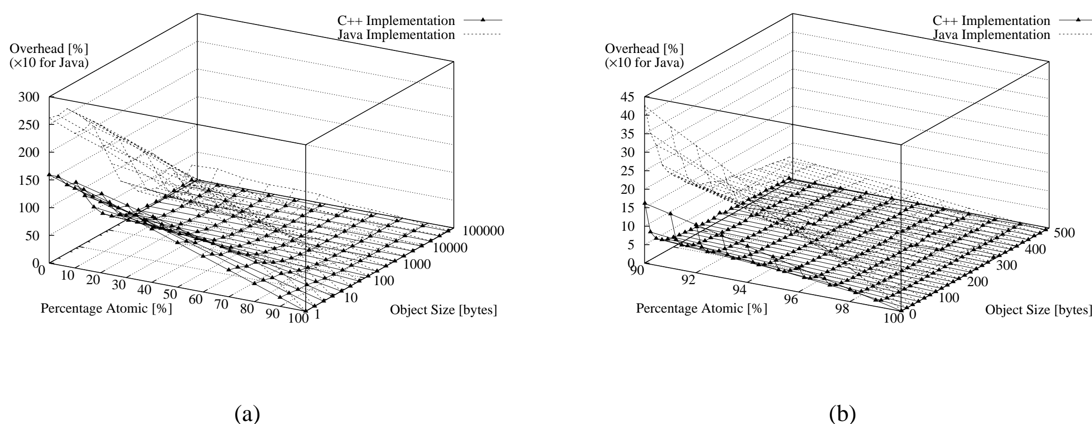


Fig. 14. (a) Performance overhead for masking a variable-duration method (proportional to the object size) in C++ and Java. Note the different scaling of the vertical axis for Java measurements. (b) Zoom on the front-right corner of (a).

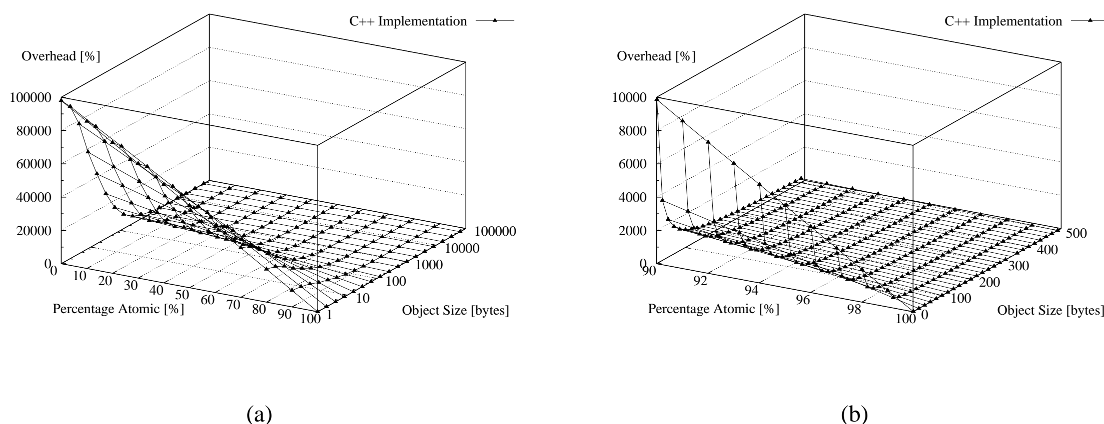


Fig. 15. (a) Performance overhead for masking a variable-duration method (proportional to the object size) in C++ using fork-based checkpointing. (b) Zoom on the front-right corner of (a).

clears the state of the object, i.e., the work performed by the method depends directly on the size of the object. As we can see in Figure 14, the relative overhead actually decreases for larger objects, because the method performs a higher amount of work per byte than the checkpointing code does. As for the fixed-length method, the Java implementation is consistently more than one order of magnitude slower than its C++ counterpart.

We finally measured the overhead of forking a child process to checkpoint the state of an object in C++: a child process is forked before a each call to a pure non-atomic method and, if

an exception is caught in the wrapper, the application state is rolled back to the state of the child process before propagating the exception (now in the child process). The performance results, shown in Figure 15, indicate that this mechanism is too expensive. Fork-based checkpointing does not appear to be practical, except for applications with very large objects and very few calls to pure non-atomic functions. Based on results of [28], however, we believe this is an artifact of our implementation and not inherent to the process checkpointing approach.

VII. Conclusion

In this article we have introduced the failure atomicity problem and proposed a system that addresses it. Our system can automatically detect which methods are failure non-atomic, and in most cases automatically turn them into failure atomic methods. To discover failure non-atomic methods, we inject exceptions at runtime into each method executed by an application, and we compare the states of the objects before the method is called and after it was abruptly terminated by the exception. Methods that cause an object to enter an inconsistent state are classified as failure non-atomic. To transform a failure non-atomic method into a failure atomic method, we take a snapshot of the state of the object before each call to the method; if an exception is thrown, we restore that state before propagating the exception to the caller.

Our exception injection system alerts the programmer when finding failure non-atomic methods. In many situations, the programmer can correct the problem by applying simple modifications to his code (such as reordering a couple of statements). In other cases, more elaborate modifications are required to implement failure atomicity. In those situations, the programmer can use the automatic masking mechanisms provided by our system.

We have implemented our infrastructure for detecting and masking non-atomic exception handling in both Java and C++. Experimental results have shown that our system is effective and can be of great help to the developer of robust applications.

References

[1] T. Cargill, "Exception handling: A false sense of security," *C++ Report*, vol. 6, no. 9, November-December 1994.

[2] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[3] P. M. Melliar-Smith and B. Randell, "Software reliability: The role of programmed exception handling," in *Proceedings of an ACM conference on Language design for reliable software*, 1977, pp. 95–100.

[4] J. X. B. Randell, “The evolution of the recovery block concept,” in *Software Fault Tolerance*, M. Lyu, Ed. Wiley, 1995, pp. 1–21.

[5] “Exception handling for a 21st century programming language proceedings,” *ACM SIGAda Ada Letters*, vol. XXI, no. 3, 2001.

[6] A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, Eds., *Advances in Exception Handling Techniques*. Springer Verlag, 2001.

[7] S. E. Mitchell, A. Burns, and A. J. Wellings, “Mopping up exceptions,” *ACM SIGAda Ada Letters*, vol. XXI, no. 3, pp. 80–92, 2001.

[8] M. Lippert and C. V. Lopes, “A study on exception detection and handling using aspect-oriented programming,” in *Proceedings of the 22nd international conference on Software engineering*. ACM Press, 2000, pp. 418–427.

[9] F. Cristian, “Exception handling and tolerance of software faults,” in *Software Fault Tolerance*, M. Lyu, Ed. Wiley, 1995, pp. 81–107.

[10] R. Maxion and R. Olszewski, “Eliminating exception handling errors with dependability cases: a comparative, empirical study,” *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 888 – 906, 2000.

[11] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martin, and D. Powell, “Fault injection for dependability validation: A methodology and some applications,” *IEEE Trans. on Software Eng.*, vol. 16, no. 2, pp. 166–182, 1990.

[12] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin, “Fiat — fault injection based automated testing environment,” in *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, Tokyo, Japan, June 1988, pp. 102–107.

[13] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, “Goofi: Generic object-oriented fault injection tool,” in *Proc. International Conference on Dependable Systems and Networks (DSN 2001)*, Gothenburg, Sweden, 2001.

[14] N. P. Kropp, P. J. K. Jr., and D. P. Siewiorek, “Automated robustness testing of off-the-shelf software components,” in *Symposium on Fault-Tolerant Computing (FTCS)*, 1998, pp. 230–239. [Online]. Available: citeseer.nj.nec.com/kropp98automated.html

[15] J.-C. Fabre, M. Rodriguez, J. Arlat, and J.-M. Sizun, “Building dependable cots microkernel-based systems using mafalda,” in *2000 Pacific Rim International Symposium on Dependable Computing (PRDC’00)*, Los Angeles, California, December 2000, pp. 85–94.

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, “Ferrari: A tool for the validation of system dependability properties,” in *Proc. of 22nd International Symposium on Fault Tolerant Computing (FTCS-22)*. Boston, Massachusetts: IEEE, 1992, pp. 336–344.

[17] J. Carreira, H. Madeira, and J. G. Silva, “Xception: A technique for the experimental evaluation of dependability in modern computers,” *Software Engineering*, vol. 24, no. 2, pp. 125–136, 1998. [Online]. Available: citeseer.nj.nec.com/carreira98xception.html

[18] S. Han, K. Shin, and H. Rosenberg, “Doctor: An integrated software fault injection environment for distributed real-time systems,” 1995. [Online]. Available: citeseer.nj.nec.com/han95doctor.html

[19] P. Broadwell, N. Sastry, and J. Traupman, “Fig: A prototype tool for online verification of recovery mechanisms,” in *ACM ICS SHAMAN Workshop*, Ney York, NC, June 2002.

[20] C. Fetzer and Z. Xiao, “An automated approach to increasing the robustness of C libraries,” in *International Conference on Dependable Systems and Networks*, Washington, DC, June 2002. [Online]. Available:

http://www.research.att.com/c̃hristof/papers/rwrapper.pdf

[21] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[22] D. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, vol. 30, no. 2, pp. 123–169, 1998.

[23] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, (OOPSLA 2003)*, Anaheim, CA, USA, October 2003, pp. 388–402.

[24] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[25] M. Goto, "CINT C/C++ interpreter," http://root.cern.ch/root/Cint.html.

[26] O. Spinczyk, A. Gal, and W. Schrder-Preikschat, "AspectC++: an aspect-oriented extension to C++," in *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 18-21 2002.

[27] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix, Tech. Rep. UT-CS-94-242, 1994. [Online]. Available: citeseer.ist.psu.edu/plank95libckpt.html

[28] D. E. Lowell and P. M. Chen, "Discount checking: Transparent, low-overhead recovery for general applications, Tech. Rep. CSE-TR-410-99, November 1998.

[29] The Apache Software Foundation, "BCEL: Byte Code Engineering Library," http://jakarta.apache.org/bcel.

[30] ——, "Regexp," http://jakarta.apache.org/regexp.

[31] D. Lea, "Collections," http://gee.cs.oswego.edu/dl/classes/collections.

[32] C. Fetzer and K. Högstedt, "Self*: A component based data-flow oriented framework for pervasive dependability," in *Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Jan 2003.