Using Program Analysis to Identify and Compensate for Nondeterminism in Fault-Tolerant, Replicated Systems *

Joseph G. Slember and Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3891
jslember@ece.cmu.edu, priya@cs.cmu.edu

Abstract

Fault-tolerant replicated applications are typically assumed to be deterministic, in order to ensure reproducible, consistent behavior and state across a distributed system. Real applications often contain nondeterministic features that cannot be eliminated. Through the novel application of program analysis to distributed CORBA applications, we decompose an application into its constituent structures, and discover the kinds of nondeterminism present within the application. We target the instances of nondeterminism that can be compensated for automatically, and highlight to the application programmer those instances of nondeterminism that need to be manually rectified. We demonstrate our approach by compensating for specific forms of nondeterminism and by quantifying the associated performance overheads. The resulting code growth is typically limited to one extra line for every instance of nondeterminism, and the runtime overhead is minimal, compared to a fault-tolerant application with no compensation for nondeterminism.

1. Introduction

Replication is a common technique used to provide fault-tolerance to distributed applications. With replication, multiple copies, or replicas, of the application's components are provided so that even if one replica crashes, other replicas can continue to provide service. Clearly, for replication to work, the replicas need to be identical, both in state and in behavior. When an application is replicated for fault-tolerance, the most important criterion is to keep the replicas consistent in state, even as they receive invocations, process invocations, and return responses. This requires delivering the same set of operations, in the same order, to all of the replicas assuming, of course, that the application is *deterministic*.

From a fault-tolerance viewpoint, an object is said to be deterministic if all of its replicas (either on the same processor or on different processors), when starting from the same initial state and executing the same set of operations in the same order, reach the same final state. It is this reproducible behavior of the object that lends itself well to reliability. If an object did not exhibit such reproducible behavior, one could no longer maintain the consistency of the states of its replicas.

Many distributed applications are written using middleware such as CORBA. For CORBA applications, fault-tolerant determinism can be achieved by forbidding the application's use of any mechanism that is likely to produce different results on different processors. This includes a long list of items, such as local timers, local I/O, hidden channels of communication (such as non-CORBA communication), multithreading, etc. Simple CORBA applications can clearly satisfy the notion of determinism from a fault-tolerance perspective. For more realistic applications, especially critical systems that deal with real-world (and inherently nondeterministic) entities, it is often not possible to prohibit the various forms of nondeterminism.

The recent Fault-Tolerant CORBA [18] (FT-CORBA) standard describes standardized fault-tolerant mechanisms and interfaces to be included in any CORBA implementation. Unfortunately, the FT-CORBA standard fails to address the non-trivial challenges of providing fault-tolerance to real-world, nondeterministic CORBA applications. In fact, the specification is explicit in stating its lack of support for nondeterministic applications:

> "If sources of nondeterminism exist, they must be filtered out. Multi-threading in the application or the ORB may be restricted, or transactional abort/rollback mechanisms may be used."

* This work has been partially supported by the NSF CAREER grant CCR-0238381 and General Motors Collaborative Laboratory at Carnegie Mellon University.

In addition, the heterogeneity of today's platforms forces FT-CORBA to mandate that all of the replicas of a CORBA object must be hosted on the same processor, operating system (version) and hardware. Otherwise, unforeseen nondeterministic effects can arise from any differences in the underlying platform that can cause replicas to diverge in state and/or behavior.

Some approaches to handling nondeterministic applications recommend the use of specific forms of replication (such as passive or semi-active replication); however, as we show later in this paper (Section 2.1), every replication style is vulnerable to the problems of nondeterminism. Yet other approaches recommend the use of a virtual-machine-like layer that is interposed between the operating system and the application; the layer overrides, and compensates for, nondeterministic system calls and returns identical results to all of the replicas of the application. It is our belief that the knowledge of the application's structure and functionality can provide for a better way to identify, and to handle, nondeterminism. Thus, using a technique that compensates for every form of nondeterminism, regardless of whether it is present in the application, might be overkill.

For CORBA-based systems, one of the authors of this paper had pioneered an earlier approach for compensating for multithreading in CORBA applications through the use of a special interceptor-based operation scheduler [17] that "watches" and schedules incoming invocations carefully, preserving replica consistency for multithreaded ORBs and applications. The transparency of this approach has the advantage that it allows for ease of use in deploying the system; the disadvantage is that the approach fails to exploit any application-specific information that could make it easier and more effective to compensate for nondeterminism.

The lessons that we learned with this previous transparent approach to nondeterminism have motivated the improved, and interdisciplinary, approach that we describe in this paper, where we apply static program analysis to distributed fault-tolerance. It is our current conviction that, for issues such as nondeterminism, (i) source code-level information is essential and useful, (ii) the application programmer/deployer should have the ability to choose how to compensate for the nondeterminism based on the resulting overhead, and (iii) involving the application programmer is an important step needed to educate programmers in developing distributed fault-tolerant applications.

The contributions of this paper include the details of the design and implementation of our novel program analysis-based approach to identify, and compensate for, nondeterminism in distributed CORBA applications. Our approach inserts code into the application to handle each instance of identified nondeterminism. There can exist multiple, diverse sources of nondeterminism: system calls, multithreading, asynchronous signals *etc.*; in this paper, to illustrate our technique, we target two specific sources of nondeterminism of the system-call kind, namely, local timers and random numbers.

We selected the `gettimeofday()` system call as an example of nondeterminism is because it is used in timer operations, timeouts and various other applications. In fact, the CORBA Time Service is often implemented using this call and is, therefore, nondeterministic. Thus, our approach to correcting time-based nondeterminism could be applied to providing a deterministic version of the CORBA Time Service. We picked random numbers as another instance of system-call nondeterminism to establish that our technique is independent of the specific system call.

In this paper, we also describe the runtime infrastructure that, together with the results of our compile-time analysis, provides for a way to cope with, and compensate for, application-level nondeterminism, without compromising replica consistency. In addition, we quantify the overheads of our approach for both active and passive replication styles, for different degrees of replication. While we target CORBA applications in this paper, our approach to handling nondeterminism in distributed replicated applications is independent of CORBA, and can be equally applied to non-CORBA applications.

The details of the MEAD infrastructure, which handles node-crash faults, message-loss faults and process-crash faults, are described elsewhere in the literature [12]. This paper, however, focuses primarily on how nondeterministic CORBA applications can be handled, given the existence of an underlying fault-tolerance infrastructure, such as MEAD, to handle all of the fault-tolerance issues (reliable ordered message delivery, checkpointing, fault-detection and fault-recovery) other than nondeterminism.

2. Background

Over the past thirty years, programming paradigms and techniques have developed significantly [23]. The first programs were entered into a shared computer system using punch cards; these programs were typically written using the sequential programming approach. Sequential programming control-flow provided all of the information needed at compile-time in order to determine the path of execution, and did not introduce any runtime nondeterminism. However, sequential programming was severely limiting to the programmer in terms of the applications that could be written; with time and the associated hardware advances, programming paradigms evolved to event-based and thread-based programming that allowed for far more versatility.

Event-based programming allows the execution of an entire program to be completely dependent on external stimuli, while thread-based programming allows two or more concurrent tasks to run the same application code at the

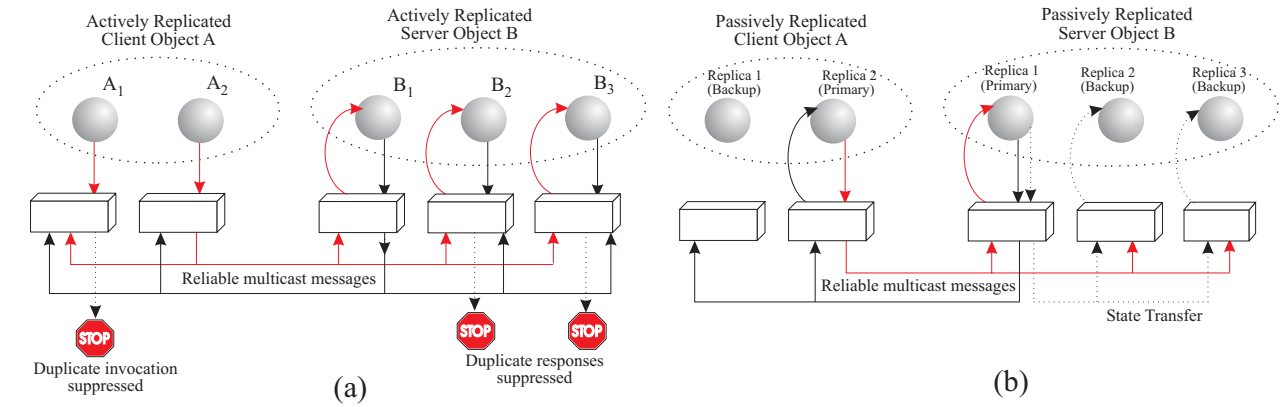


Figure 1. Different replication styles: (a) active replication and (b) passive replication.

same time. Clearly, both event-based and thread-based programming introduce the possibility of nondeterminism because we are no longer able to predict the control flow of such programs, especially in the face of asynchronous interrupts. Note that CORBA is an event-based and thread-based programming paradigm and is, therefore, susceptible to nondeterminism.

There clearly is a need for a compiler-level mechanism to deal with nondeterminism. In the absence of such a mechanism, the typical solution to compensating for nondeterminism would effectively undo the advances in programming languages by transforming (by forcibly serializing) thread-based programs to result in sequential programs. Furthermore, this transformation approach has its drawbacks: the resulting overheads are often not trivial and the transformations might not always be possible. Thus, handling nondeterminism by reverting to sequential programming might be infeasible (if the specific form of nondeterminism can not be serialized away) and expensive (if the associated overheads are high).

Our alternative approach of using a compiler-based mechanism to handle nondeterminism is attractive in not requiring modifications to the programming style, and in potentially not incurring high overheads and in being able to handle various forms of nondeterminism.

2.1. Replication Styles

Nondeterministic behavior is an inevitable and challenging problem in the development of fault-tolerant systems, regardless of the replication style employed. Figure 1(a) shows active or state-machine replication [22], where every replica performs every operation. Figure 1(b) shows passive replication [10], where only one replica, the primary, performs every operation, and periodically synchronizes its state with its backups; one of the backups takes over if the primary fails.

For active replication, determinism is crucial to maintaining the consistency of the states of the replicas of the object. Passive replication is often perceived to be the solution for nondeterministic applications. There is some truth in this because, with passive replication, invocations are processed only by the primary, with the primary's state being captured and then used to assign the states of the backup replicas. If the primary fails while processing an invocation, any partial execution is discarded, and the invocation is processed afresh by the new primary. Because the state updates happen only at one of the replicas, namely, at the primary, the results of any nondeterministic behavior of the replicated object can be completely contained, without compromising the replica consistency of the object.

However, there do exist situations where passive replication does not compensate for non-determinism. This can happen when the nondeterministic behavior of a passively replicated object is not contained because its behavior "leaks" to other replicated objects in the system. Consider a multi-tiered (or nested) application where a client invokes the primary server replica, which then invokes some other server based on some nondeterministic decision (*e.g.*, for load balancing, the primary randomly chooses one of *n* servers to process a request). If the primary fails after issuing the invocation, there is no guarantee that the new primary will select the same server as the old primary; thus, the system could now be in an inconsistent state because the old and the new primary replicas have communicated with different servers, both of whose states might be updated differently.

Unless we can ensure that the old and new primaries do not affect external components differently for the same operation, passive replication does not always resolve nondeterminism. For passive replication to work around nondeterministic behavior, there should be no persistent effect (*i.e.*, no lingering "leakage") due to the nondeterministic, partial execution of an invocation due to the failure of the primary.

Semi-active and semi-passive replication styles are no cure for nondeterminism, either. Should a fault occur in the primary before the nondeterministic information is relayed to the backups, several issues can arise. The backups will need to roll-back to the previous deterministic checkpoint. Additionally, if any nondeterministic information was propagated just before the primary failed, then, the nodes that received this information will also need to be rolled back. Additionally, semi-active or semi-passive replication can be regarded as a form of serialization because the backups must wait to receive the execution path of the primary, and any nondeterminism that resulted from it, before they can proceed. Furthermore, to avoid the "lag", the primary must wait for the backups to complete this update, thereby slowing down the overall execution of the replicated process.

3. Design Objectives

Rewriting the middleware, or implementing a virtual machine, to order execution and ensure consistency between replicas is not the objective of our work. We opt instead to exploit program analysis to develop a solution to the problem of nondeterminism. We wish to permit programmers to continue to create applications that are nondeterministic; we aim to support different solutions that either eliminate the nondeterminism, or leave it in the code but quantify the overhead of compensating for it, without loss of consistency. In this manner, programmers can employ any programming paradigm of their choice, but are made to understand/appreciate the side-effects of any nondeterminism that they might introduce.

We have developed an approach that allows nondeterminism to exist within application code, but uses program analysis to isolate the possible places where nondeterminism can affect the system state or behavior. Then, we perform relatively simple code transformations to ensure consistent results (as perceived at the client side) across all of the server replicas. This is different from forcing each replica to be deterministic so that each creates the correct result in the same way.

We identify nondeterminism at compile-time by using program analysis techniques on the application's source code. We compensate for nondeterminism also through program analysis, but by employing separate compile-time and run-time components. Our approach is deliberately not transparent to the programmer so as to allow him/her the opportunity to eliminate the nondeterminism from the application, and not transparent to the deployer so as to allow him/her to tune the amount of nondeterminism to eliminate, based on the associated overheads.

The difference between our approach and related work is that we offer the programmer the option of how to deal with the nondeterminism based on the latency overheads as well as code growth. In this paper, as a proof of concept, we have chosen to address one form of nondeterminism (those arising from system calls) to target and demonstrate how our approach/technique solves the problem. Our ongoing research (outside the scope of this paper) focuses on extending our approach to other, more complicated forms of nondeterminism as well.

4. Assumptions and Limitations

Our approach, while providing significant advantages, requires some assumptions in order for it to be practically implementable and to produce results. We list some of these assumptions, and the limitations of our approach that follow from these assumptions.

While program analysis allows application programmers to continue to use the majority of programming techniques and styles that they are accustomed to, some practices currently prove to be difficult for our program analysis to handle. Inheritance (a feature supported by many programming languages), especially multiple inheritance, endows objects with multiple levels of attributes, each constituting a potential source of nondeterminism. If the application employed inheritance, we would need a significant amount of compile-time and run-time information to keep track of each object and its inherited attributes and to determine how nondeterminism can matriculate through the hierarchy. Although it is possible for us to do this, it can result in significant overheads, and is outside the scope of this paper. Polymorphism is similar to inheritance because the exact function that is called or the event that is initiated depends entirely on the type of the object; however, the overhead for us to handle polymorphism might not be as large because type information is typically known at compile time.

Identifying and eliminating all sources of nondeterminism is a challenging problem, especially for distributed applications. While we feel that our approach is generic enough and has the ability to provide solutions to many sources of nondeterminism, we have chosen to focus on one specific type of nondeterminism (those arising from system calls, such as `gettimeofday()`) in this paper in order to justify our approach and to demonstrate its benefits. Other types of nondeterminism, such as multithreading, asynchronous signals $etc.$, form the focus of our ongoing research, and are outside the scope of this paper because they require relatively more complex nondeterminism-compensation mechanisms.

In addition, we make the following assumptions:

- The CORBA application is replicated with replicas running on different processors in a distributed, asynchronous system. No assumption is made about the relative speeds of the processors.

- We assume that we have access to application source code, and that we are allowed to modify the client and server source-code , as well as the IDL interfaces of all objects. This assumption could pose a problem in the case of proprietary software or for systems where multiple versions of code might co-exist.
- We assume that every other layer in the architecture (including the ORB and the operating system) is deterministic and that only the application exhibits nondeterminism. We recognize that this assumption is not realistic in the long run – however, this paper is a proof of concept of our approach to handling nondeterminism, although only at the application level. We intend to apply our technique recursively to ORB and infrastructure source code to handle nondeterminism comprehensively.
- Processes/processors fail independently of each other.
- Our current fault model encompasses node-crash, process-crash and message-loss faults.

The capabilities of the underlying MEAD infrastructure are transparent, including, for instance, the reliable message delivery, ordering and replication mechanisms.

5. Approach

Current and previous research practices in eliminating nondeterminism do not analyze the interactions between the different parts of the client and server application code. With our approach, the client-side and server-side code are jointly analyzed, and modified, as a single unit. Data-flow and control-flow analyses allow us to determine the instructions that execute in the server and that produce a nondeterministic result which can be either returned to the client or used in future updates to the server state. The results of this nondeterministic execution need to be propagated back to the client, which can then store the specific results in a special structure that represents the nondeterministic state of the server. Thus, if one of the server replicas fails, a new replica can have the nondeterministic portion of its state initialized/overwritten through the "cached" nondeterministic snapshot (of the previous failed server's nondeterministic state) stored at the client.

5.1. Program Analysis

5.1.1. Preparation for Program Analysis. To make program analysis possible, we first need to convert the input source code into an intermediate format. The CORBA test applications that we used were written in C++, which required us to use EDG [1] to convert the C++ code to C, and then to use SUIF2 [3] compiler to convert the resulting C code into an intermediate representation amenable to program analysis. As show in Figure 2, our analyzer makes multiple passes through each intermediate file, and highlights the sources of nondeterminism in the code. For instance, a pass that discovers a nondeterministic call will annotate the return value of that call and then track the variable holding the return value as a potential source of nondeterminism. A subsequent pass will attempt to establish whether this variable, or any other data that it affects, is used in an invocation to a server; in that case, there exists the possibility of nondeterminism propagating to another node in the distributed system. For each source file, our analyzer creates a dependency file that captures the nondeterministic behavior of that source code. Since our current implementation only targets the application's source code, we do not need to modify either the ORB or the operating system. The list of steps that are performed in the preparation stage are as follows:

- *Source-code conversion from C++ to C*: We do this for several reasons. Conversion to C allows for much easier analysis because it eliminates the complexities (*e.g.*, object-oriented issues) that C++ introduces. Additionally, we are able to take advantage of current compiler tools that expedite the transformation of C code into a workable, efficient intermediate form (annotated parse tree).
- *Data-flow pass*: At least one pass is needed over the tree to determine and fully recognize all of the variables and their characteristics. "Def-use" chains are significantly important in this part of the analysis; determining where a variable is defined, used and redefined is invaluable for ascertaining when and where nondeterminism can occur. We do not perform pointer-aliasing analysis because this is complicated and beyond the scope of this paper. We note, however, that there exist advanced compiler techniques that we intend to exploit in the future to resolve some of these issues.
- *Nondeterminism-analysis passes*: We designed our approach to be modular. For each type of nondeterminism, we perform a separate pass through the code. While some passes could potentially overlap in the analyses that they perform, this is relatively inconsequential because it is performed at compile-time. The additional overhead at compile-time is far outweighed by the decreased complexity and ability to control specific modifications and analysis.

  As mentioned before, in this paper, we target the type of nondeterminism introduced by functions such as `gettimeofday()` or `random()`. This pass determines where in the code such functions are called; we also identify and store the variables assigned to the return values of these nondeterministic functions. Beyond the prior simple data-flow pass, we now need to perform further analysis on which variables are as-

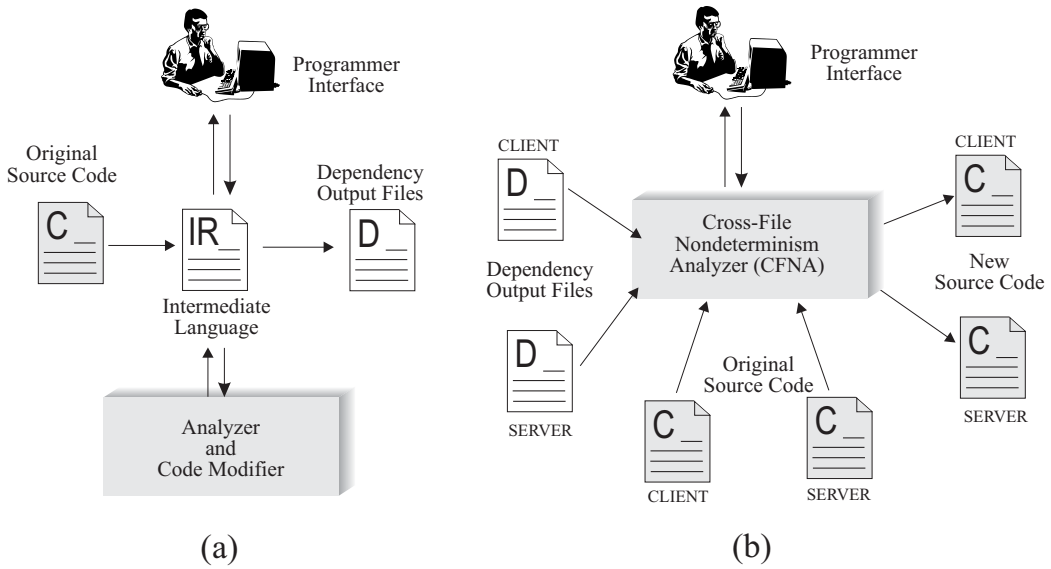


Figure 2. Our program analysis framework.

signed to others, in order to ascertain the transference and proliferation of data. This information is needed to determine how nondeterminism spreads to other parts of the application code.

- *Control-flow passes*: From the previous passes over the tree, we know where nondeterminism exists, and how it can propagate through the code. However, an application take multiple paths as it executes; thus, we now need to evaluate all of the possible execution paths. Control-flow information, combined with the previously gathered information, brings together all that we need for completing the preparation phase. For instance, if a variable x is assigned nondeterministic information, and x is then assigned to another variable y somewhere else in the code, it is possible that y can subsequently become nondeterministic, after the assignment. In this pass, we determine if such an order of assignments between variables exists along a particular execution path.

5.1.2. Correlation Analysis For Nondeterminism. The next stage of program analysis involves multiple passes over each dependency file to determine the interaction between the different components of the application. As shown in Figure 2, all of the dependency files are input into our Cross File Nondeterminism Analyzer (CFNA), where the majority of our compile-time compensation is accomplished. One challenge here was to maintain linking information that might be used in different libraries to form a chain of how information is passed within the application.

For each specific instance of nondeterminism, the CFNA inserts data structures to hold the nondeterministic information; for example, the CFNA creates a `struct` to hold the return value of `gettimeofday()`. This `struct` is then prepended to each message that outputs/propagates the nondeterminism. Additionally, the recipient of the message, whether it is a client or a server, must also store the nondeterministic information locally, thereby creating the `struct` also at the receiving side. Because we can find the source and the destination of the nondeterminism, we can determine where the CFNA should automatically insert the compensation code-snippet into the application source-code, namely, just after the system call that is nondeterministic. For example, if `gettimeofday()` is called on each invocation of a replicated time-server, if there exists clock-skew across the nodes hosting the replicas, then, it is important that to compensate for the times; in this case, the compensation code-snippet performs the clock-skew offset. Once the CFNA has modified all of the source files appropriately, the modified application is compiled to produce a version with compensation for nondeterminism.

5.2. Runtime Compensation for Nondeterminism

The run-time part of our approach is responsible for piggybacking the extracted, run-time nondeterministic information onto messages exchanged between the client and the server in the application. In Figure 3, we show a CORBA client communicating with a three-way actively replicated CORBA server. The MEAD infrastructure forms the fault-tolerant layer that enables reliable ordered delivery of the application's messages.

In Figure 3(a), the client sends a request to the replicated server and receives three distinct replies. Assume that the server executes a nondeterministic function before respond-

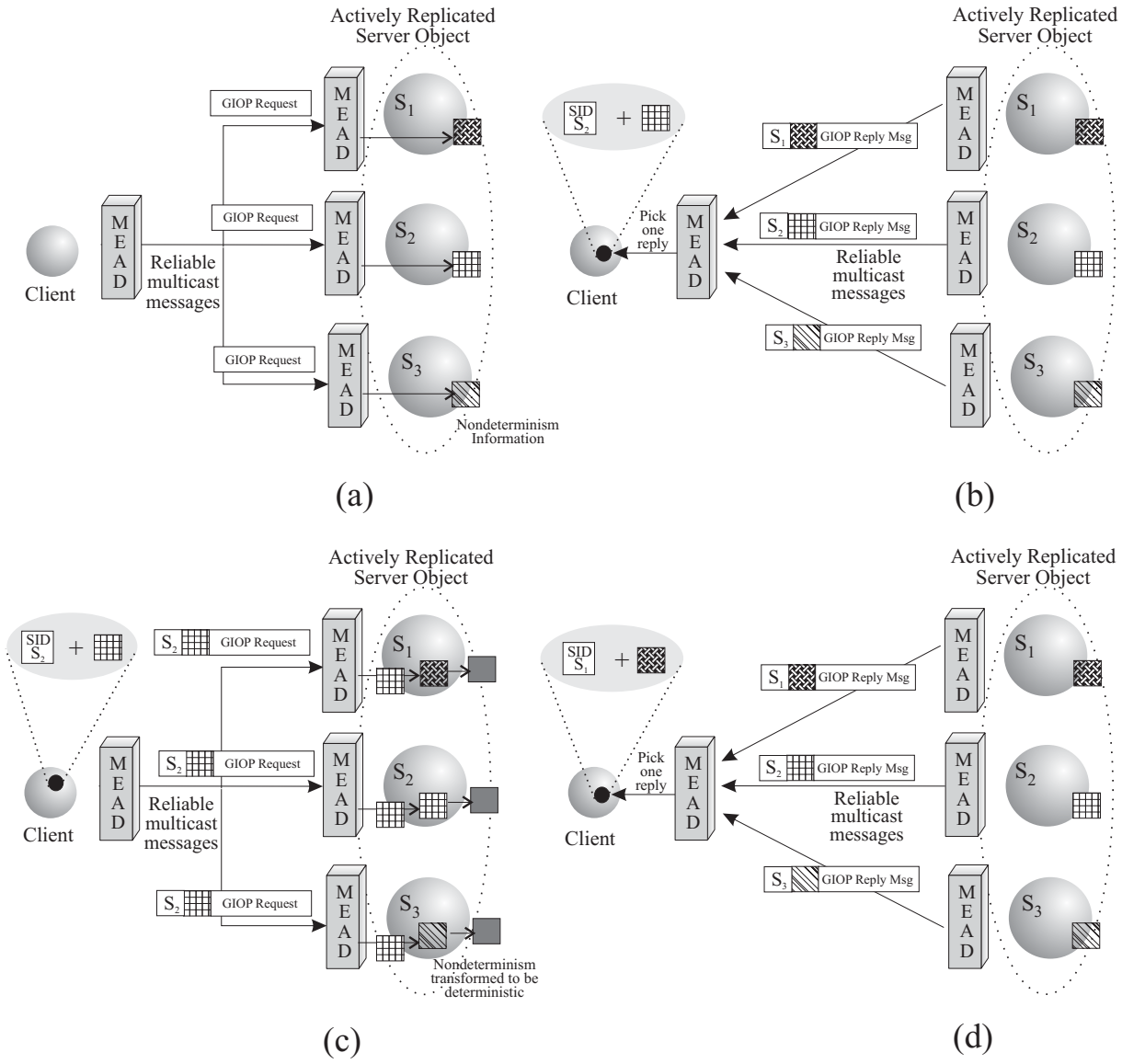


Figure 3. (a) Client invokes nondeterministic replicated server. (b) Server replicas respond to client with prepended nondeterministic values (shown shaded). (c) Client invokes nondeterministic server again, this time prepending the received nondeterministic values to the new invocation. Each replica compares the prepended SID to its own, and executes compensation snippet, if there is a mismatch. (d) Server replicas respond to client with new nondeterministic values prepended to the responses.

ing to the client. Each active server replica processes the request, and prepends the nondeterministic results, along with a unique server identifier `SID`, to the response that it returns to the client. Each server replica stores its nondeterministic information locally, until the next invocation, as shown in Figure 3(b). In the figure, the client picks the first received response from the replicated server (in this case, from replica $S2$) and stores the prepended information. In Figure 3(c), in the client's subsequent request to the replicated server, the client prepends the nondeterministic information extracted from the previous reply. In Figure 3(d), each server replica receives the request and compares its identifier to that transmitted with the prepended header. If there is a match (as in replica $S2$), then, the invocation proceeds normally. However, where there is a mismatch ($S1$ and $S2$), the server replicas run the compensation code after processing the invocation. For example, in the case of `gettimeofday()`, the compensation would

```
CORBA::Long Time_impl::get_cycles()
        throw (CORBA::SystemException) {
    time_t time_now = time(0);
    struct tm * time_p = gmtime(&time_now);
    time_p->tm_hour += (24 + this->time_zone_st);
    time_p->tm_hour %= 24;
    long cycles =
        (((time_p->tm_hour - this->past_tod.hour)*3600) +
        (time_p->tm_min - this->past_tod.minute*60) +
        (time_p->tm_sec - this->past_tod.second)*18000000);
    return cycles;
}
```

Figure 4. *getcycles()* function containing nondeterminism.

```
TimeTransfer::NonDetStruct
    Time_impl::get_cycles_nondet_corr(
        const TimeTransfer::NonDetStruct & nd_pass)
    throw (CORBA::SystemException) {
    time_t time_now = time(0);
    struct tm * time_p = gmtime(&time_now);
    TimeTransfer::NonDetStruct tod;
    tod.sid = this->sid;
    tod.time = time_p;
    if(this->sid != nd_pass.sid) {
        int sec_diff =
            ((nd_pass.hour - this->past_tod.hour)*3600) +
            (nd_pass.minute - this->past_tod.minute*60) +
            nd_pass.second-this->past_tod.second);
        tod.cycles =
            (((((tod.hour - this->past_tod.hour)*3600) +
            (tod.minute - this->past_tod.minute*60) +
            (tod.second-this->past_tod.second)- sec_diff))
                    *18000000);
        this->past_tod = time_p;
        return tod;
    } else {
        tod.cycles =
            (((tod.hour - this->past_tod.hour)*3600) +
            (tod.minute - this->past_tod.minute*60) +
            (tod.second-this->past_tod.second)*18000000);
        return tod;
    }
}
```

Figure 5. *getcycles()* function with compensation for nondeterminism. The code could be optimized so compensation is only one line. The nondeterminism is also stored from the previous invocation.

involve a local CFNA-generated code-snippet (see Figure 4 and Figure 5), typically just a single line of code, executing to compute the offset between the prepended nondeterministic information and the locally computed one. Each server replica then replies with its result for the invocation, along with the nondeterministic information that it generated locally. Each replica's local "cache" of nondeterministic information is also updated. This back-and-forth propagation and compensation of nondeterminism occurs on every invocation from the client to the replicated server.

5.3. Optimizations

For every instruction that influences the server state in such a way that it might not be possible for the different replicas to reproduce the same result, the associated data variables and values must be sent back to the client. Doing this

```
int a = random();
int b = a + 5;
b = b * b;
c = b * 3.14;
/* Checkpoint is inserted after assignment of ''c'',
because ''c'' captures any prior nondeterminism
due to the random() function call. */
```

Figure 6. Inserting checkpoints appropriately.

for every system call would incur very high overheads. Instead, we can adopt a checkpoint strategy. For instance, if we know that variable *a* stored a random number that was used in calculations through several lines of code, but was then never used afterwards, we could insert a checkpoint after the last use of *a*, as shown in the snippet in Figure 6.

We only need to send back *b* and its value to the client since *a* is never used again; *c* fully encapsulates the nondeterminism due to *a*. This does not entirely eliminate overheads; inserting checkpoints based on usage scope of a variable can impact overheads, especially if variables are used for only a few lines of code. This is likely since the average length of a basic block is typically only three to four instructions [2], and it is also possible for one instruction to span an entire function or even the entire program. Clearly, global variables represent such a worst-case scenario. Therefore, a different checkpointing approach needs to be adopted because we are building a distributed fault-tolerant system, and allowing programmer interaction. During its implementation, we can ask the programmer to set the level of fault tolerance, including such information as the maximum tolerable downtime (recovery time), *i.e.*, the longest time between a replica's failure and a new replica being made operational. Once this specific information is known, we can take advantage of it to provide for a smarter checkpointing strategy. The use of worst-case execution time analysis [13] plays a major role in our checkpointing scheme. Assuming that communication costs are fixed, we analyze both client and server code to compute the worst-case execution time between different points in the code. Based on this knowledge, we know the execution time from any point in the execution path to any other. We can decompose this path into segments, corresponding to the maximum recovery time specified by the programmer, and then insert checkpoints at the end of each segment. To further eliminate nondeterminism due to heterogeneous hardware platforms, we can specify time in cycles, rather than in seconds. Specifying time in seconds does not work in distributed systems where clock speeds differ across machines (the execution time of different replicas will then depend on the speed of their respective processors). We favor using the number of cycles for a task's execution because this value is likely to be identical even on heterogeneous hardware. This technique is processor independent. However, the actual worst-case execution time will depend on the clock speed of each processor. Therefore, in a heterogeneous system, we would recommend running the checkpointing analysis on each ma-

chine that is different from the others. Those machines that have the same configuration can maintain the same checkpoints.

MEAD's fault-tolerance infrastructure provides for a recovery mechanism through checkpoints on the server-side. If a replica fails, the new replica initializes itself to the most recent checkpoint, and then awaits nondeterministic information from the client. The new replica updates its local variables and compensates for nondeterminism. Once state is completely restored, the replica can begin normal operation.

6. Empirical Evaluation

In order to evaluate our approach, we ran several sets of experiments to target various metrics to measure the overhead and scalability of our approach for different replication styles.

Test-bed. We used a test-bed of seven Intel x86 machines. Each machine is a dual processor Pentium III running at 900MHZ with 512MB RAM running RedHat Linux 9.0. We employed the Spread (v3.17.01) group communication protocol to provide the reliable ordered delivery guarantees required for consistent replication by MEAD.

Test application. Our CORBA test application was built using ACE and TAO (ACE version 5.2.8 and TAO version 1.2.8). The server calculates how much work, in terms of the number of cycles, that has elapsed since the last invocation. In a normal application, such a `get_cycles()` method would perform a `gettimeofday()` and subtract that time from the previous time of invocation. The result of `get_cycles()` is obtained by multiplying the time difference by the speed of the processor, in order to return a `long` type holding the result, as shown in the first code snippet.

The second snippet of code demonstrates our approach's transformation of the source code. The input parameters have been modified to include the nondeterministic time-of-day `struct` from the client. The return value has changed from a `long` to a `struct` to hold the result, as well as the value of time used in the calculation. An `if-then` statement is added to the code after the `gettimeofday()` call. The current server replica's identifier, SID, is compared to the input parameter SID. If they match, then, the program executes normally. If there is a mismatch, then, the highlighted snippet that computes the offset between the nondeterministic input value and the current value of time is executed, with its results being used in the `get_cycles()` calculations.

Our approach incurs two distinct types of runtime overhead (i) increased computation, because of additional work done at each server replica on each nondeterministic invocation, and (ii) increased bandwidth, because of the prepended nondeterminism headers passed with every GIOP request and reply message.

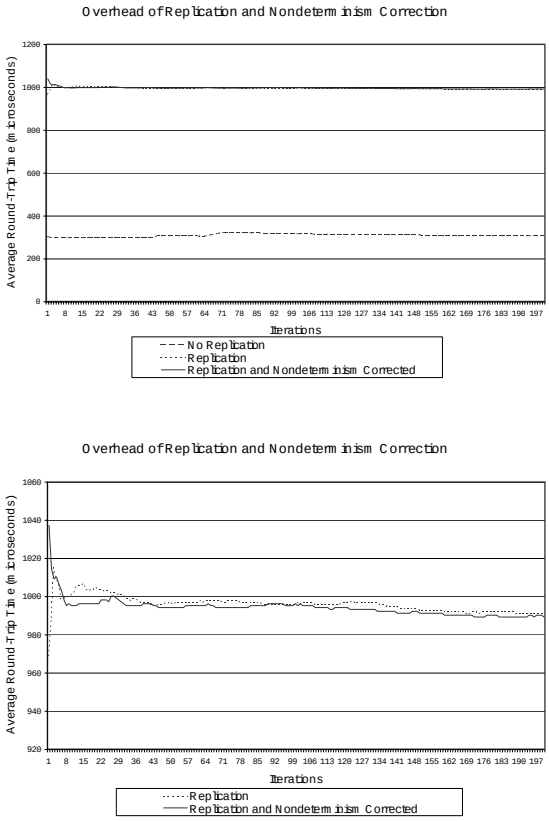


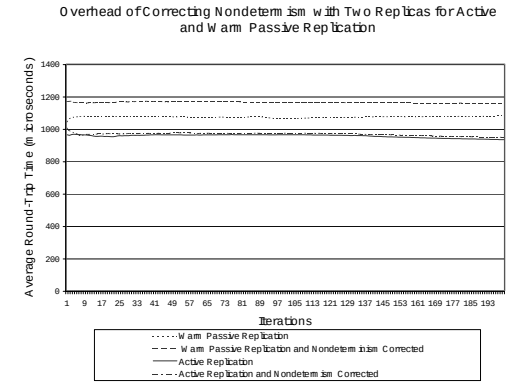
Figure 7. *Top* - Round-trip times of configurations 1, 2 and 3 with only one replica server. *Bottom* - Round-trip times for configurations 2 and 3 only.

Performance Metrics. While we could target the evaluation of our compile-time framework (through metrics such as code growth), the runtime evaluation is of most interest in a distributed fault-tolerant system. For the sake of completeness, in our simple example, the increase in footprint size is minimal, less than one percent. The correction code adds but one additional line of code, and a struct to store the nondeterminism and return it to the client.

Measurement of runtime overhead. To portray an accurate picture of our overheads, we evaluated the following three configurations:

- Configuration 1: Client with unreplicated server
- Configuration 2: Client with replicated server with no compensation for nondeterminism
- Configuration 3: Client with replicated server with compensation for nondeterminism

Overhead of replication and compensating for nondeterminism. Figure 7 displays the round-trip times of configurations 1, 2 and 3 with only one server replica.



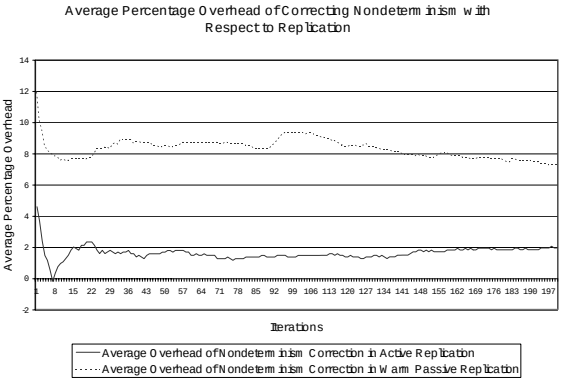


Figure 8. *Top* - Round-trip times of configurations 2 and 3 with two server replicas for both active and warm passive replication styles. *Bottom* - Average percentage overhead for both replication styles.

The bottom graph in Figure 7 provides a closer look at configurations 2 and 3. The average round-trip time for configuration 1 is 310 μs. Using replication increases the round-trip time to 994 μs (*i.e.*, an additional 684 μs) for configuration 2. Compensating for nondeterminism increases the round-trip time to 996 μs (*i.e.*, an additional 2 μs) for configuration 3. The overhead of compensating for nondeterminism is minimal with only one replica because the correction snippet never needs to execute when the same replica (the sole replica in the system, in this case) is always used. The additional overhead averages 0.2% more than without compensating for nondeterminism.

Overhead of compensating for nondeterminism with respect to replication styles. Figure 8 displays the round-trip times of configurations 2 and 3 with two server replicas for both active and warm passive replication. The average round-trip time for configuration 3 is only 15 μs more than that for configuration 2 under active replication. However, compensating for nondeterminism under warm passive replication adds, on average, an additional 89 μs to configuration 2.

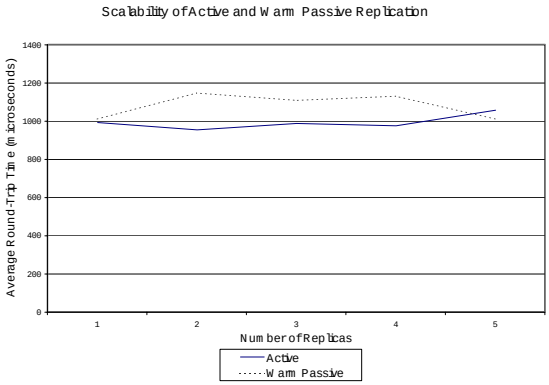


Figure 9. Round-trip times of configuration 3 for both replication styles, with 1-5 replicas.

The overhead of compensating for nondeterminism under active replication is, on average, 1.6%. The overhead is a little higher, 8.3%, under warm passive replication. This increase is predominantly due to additional message passing when checkpoints are processed in warm passive replication. The recovery snippet adds additional overhead for active replication with two replicas because the correction snippet always runs at one of the replicas. Figure 8 also shows the average percentage overhead over time for both active and passive replication styles.

Overhead of compensating for nondeterminism for increasing degree of replication. Figure 9 displays the round-trip times of configuration 3 for both replication style with 1-5 replica servers.

The average overhead of warm passive $vs.$ active replication fluctuates between 20% and 16.2% for two through four replicas. When adding a fifth replica, active replication performs worse than passive replication.

Warm passive replication has higher round-trip times compared to active replication. The overhead of checkpointing in warm passive replication drives down its performance compared to active replication. If the cost of compensating for nondeterminism was exponential or anything but constant, we would see the active replication times increase dramatically as we added replicas. This is not the case, however, and demonstrates that our approach scales well with the number of replicas.

6.1. Additional Test Example - Random Numbers

We chose to perform experimental evaluation on one other form (but similar) of nondeterminism: random numbers. A random number generator is significantly representative of the class of nondeterminism that originates from system calls that return unpredictable values. Simply starting the generators at the same seed will not work due to order of execution problems that crop up with multithreading. If a

replicated server uses the results of a random number generator in returning a response to a client's request, then, then each replica's reply will be different. Several different algorithms (*e.g.*, encryption schemes, prime number evaluators) use random number generators in this code. We target this form of nondeterminism to demonstrate that our approach works for more than just `gettimeofday()`.

Prior solutions would require coordination among the replicas to transfer the nondeterministic information, *i.e.*, the random number, used in the processing. One replica would have to compute the random number and then transfer it to the remaining replicas. Only when all replicas receive the information could the processing resume. This lock-step operation significantly increases the overhead as stated before.

For our experiments, we generated a random number that we used to offset characters in a string[1]. Each client and server has a starting random number to perform the first communication. When the client first makes a requests to the servers, it encrypts the information, a string, by multiplying each character by this random number. The server is able to decrypt it because it has the same random number. The server then performs a calculation, either addition or subtraction (so that we can maintain consistent overhead), based on the decrypted value. The result is returned along with a newly generated random number that can be used in the following round of communication.

Using program analysis, we are able to easily determine where the application generates random numbers, and how the nondeterminism spreads as the application executes. We insert code at the client and server side to compensate for nondeterminism. For every invocation, the client prepends the random number returned from the previous invocation as well as the ID of the server. Each server, when receiving an invocation, removes the server ID and checks it with its own. If the two are the same, then, processing continues. Otherwise, the server must decrypt the new random number with the old one, and use that instead.

The results are similar to the `gettimeofday()` experiments. The average round-trip time for one client with one unreplicated server was 406 μs. With replication, an average round-trip time grows to 1105 μs. Adding compensation for nondeterminism increases the round-trip time by a further 3 μs. This is due to the additional overhead of verifying the server ID. Since the same server is always used, the random number never needs to be decrypted. Thus, the overhead of compensating for nondeterminism is about 0.27%. This is representative of passive replication because only one replica is being used.

Active replication scales very similar to the `gettimeofday()` test example. With two replicas, the round-trip time increases by a further 203 μs (leading to a total round-trip time of 1311 μs. This is rise is slightly less that 18%. Three replicas register a slight decrease in round-trip time of 1290 μs, four replicas exhibit a round-trip time of 1294 μs and finally five replicas have a round-trip time of 1275 μs. These results show that our solution scales well and does not incur too much overhead.

1 While we recognize that this is a rather naive encryption algorithm, the point of this exercise is to illustrate nondeterminism in the presence of random numbers.

7. Future Directions

Currently our approach focuses only on single sources of nondeterminism, *i.e.*, our solutions will work if the nondeterminism is not multi-dimensional. For example, if an application is multi-threaded and also calls `gettimeofday()` at different places throughout the program path, then, our current solution would need to be enhanced to deal with multiple sources of nondeterminism simultaneously. We believe that we can employ the same approach, except that we would need to propagate our nondeterminism-encapsulating structures across tiers. Other future directions involve the creation of a standard nondeterminism metric. We currently measure the overhead of our approach through performance metrics such as code growth, memory usage, power consumption, or speed degradation. However, there is no metric for measuring just how much nondeterminism has been eliminated from an application. This metric could be very useful for large-scale applications to judge how much nondeterminism exists within them.

8. Related Work

Considerable research efforts have been expended in designing and implementing practical systems that employ strategies to enforce replica determinism, or to circumvent specific sources of nondeterminism. Some of the issues surrounding replica consistency and multithreading have been addressed for fault-tolerant systems that are not based on CORBA.

Gaifman[15] targets the nondeterminism that arises in concurrent programs due to external environment interaction. This technique supports only passive replication style because the backups lag behind the primary in order to ensure consistency. The technique is transparent, at runtime, to the user, but the application is actually modified by the transformation method that handles the nondeterminism. The source code is actually analyzed in order to identify nondeterminism, albeit only for nondeterminism due to multithreading.

TCP tapping [19] is used to capture and forward nondeterministic execution information to replicas. Passive replication is only supported due to the fact that the replicas must gain information from the primary after all the work is done. The approach is transparent to the application and programmer, but involves setting up routing tables to snoop on the client-to-server TCP stream, with the aim of extracting any

nondeterminism output by the primary. Once the backup receives information about the primary's nondeterminism, it can choose either to overwrite its state with that of the primary's or await a checkpoint update from the primary.

The Multithreaded Deterministic Scheduling Algorithm [16] aims to handle nondeterministic multithreading by providing for two levels of queues, internal and external, that together enforce consistency. The external queue contains a sequence of ordered messages received via multicast, while each internal queue is targeted at thread dispatching, with an internal queue for each process that spawns threads. The approach is transparent and offers a unique way of solving the nondeterminism that stems from multithreading.

Frolund and Guerraoui propose a theory of replication, X-Ability [14], based predominantly on the execution history resulting from the previous invocation histories. The approach is not necessarily transparent to the programmer because the proposed correctness criterion must be followed to ensure consistency. The advantage of this approach is that it is independent of the replication style being used.

In [25], two approaches for maintaining replica consistency are presented. The first approach is primarily based on Ada 95, while the second approach deals with a piecewise deterministic computational model. The key observation is that it is feasible to handle nondeterminism by having a primary replica that actually executes all nondeterministic events, with the results being propagated to the backups. Furthermore, this propagation does not need to occur at every nondeterministic event, but can be logged and sent to the backups at an observable, deterministic event.

The Delta-4 project employs passive [11] or semi-active replication [4] to overcome the problems associated with nondeterministic replicas. However, as we point out in Section 2.1, no replication style fully solves the problems associated with the replication of multi-tiered nondeterministic applications.

For systems that must meet real-time requirements in addition to exhibiting fault-tolerance, the replicated data must be both consistent and timely. The fault-tolerant real-time MARS system [20, 21] requires deterministic behavior in highly responsive automotive applications which exhibit nondeterminism due to time-triggered event activation and preemptive scheduling. Replica determinism is enforced using a combination of timed messages and a communication protocol for agreement on external events.

In [24], a technique is employed to track and record the nondeterminism due to asynchronous events and multithreading. While the nondeterminism is not eliminated, the nondeterministic executions are recorded so that they can be replayed to restore replica consistency in the event of rollback.

In the SCEPTRE 2 real-time system [7], nondeterministic behavior of the replicas also arises from preemptive scheduling. The developers of SCEPTRE 2 acknowledge the limitations of both active and passive replication of nondeterministic "capsules" for the purposes of ensuring replica consistency. Semi-active replication is used, with deterministic behavior enforced through the transmission of messages from a coordination entity to the multiple backup replicas for every nondeterministic decision taken by a designated primary replica. The messages force the backup replicas to override their own decisions.

The Transparent Fault Tolerance (TFT) system [8] enforces deterministic computation on replicas at the level of the operating system interface. TFT handles nondeterministic system calls by interposing a soft- ware layer between the application and the operating system. The object code of the application binaries is edited to insert code that redirects all nondeterministic system calls to a layer that returns identical results at all replicas of an object. Bressoud and Schneider's approach, Hypervisor-Based Fault-Tolerance [9], creates a virtual simulator in order to ensure that all nondeterministic data is consistent across all replicas. This is accomplished by using a simulator to run all environmental instructions, and then forcing all of the replicas to obtain the same result. This approach assumes that the virtual layer does not itself introduce any nondeterminism. Additionally, all of the replicas are simultaneously operational and required to maintain the same state.

A more recent effort [6] to address nondeterministic multithreading comprises a preemptive deterministic scheduler for an actively replicated multithreaded server. A shared state is assumed between all threads of the same replica. The approach uses mutexes between threads and the execution is split into several rounds. Because the mutexes are known at each round, the authors state that a deterministic schedule can be created. This is a unique approach with the added benefit of not requiring any communication between replicas.

Most of the previous work aims to be transparent so that the programmer and end-user are never aware of how the nondeterminism is handled. While transparency has its benefits, insights that can be gained from the application itself are not fully utilized to provide a more effective strategy.

9. Conclusion

Through program analysis, we identify the sources of nondeterminism, determine how they propagate through an application, and calculate the effects that they have on state. Armed with this information, we then create correction code that is inserted into the application source code. Our runtime support infrastructure uses this code, along with storing and transferring nondeterministic information, in order to maintain consistency across server replicas. Our approach is deliberately not transparent to the programmer, to the user, or to the application code. As a result, we are able to fine-tune which and how nondeterminism is compensated for, and offer trade-offs to the user and programmer. Our approach does not impose significant code growth or run-time overhead regardless of the number of replicas or replications styles.

References

[1] http://www.edg.com/.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[3] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. *The Basic SUIF Programming Guide*.

[4] P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, D. Seaton, N. Speirs, and P. Verissimo. The Delta-4 extra performance architecture (XPA). In *Fault-Tolerant Computing Symposium*, pages 481–488, Newcastle, UK, June 1990.

[5] C. Basile. Active replication of multithreaded applications. Technical Report CRHC-02-01, University of Illinois at Urbana-Champaign, 2002.

[6] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *International Conference on Dependable Systems and Networks*, pages 149–158, San Francisco, CA, June 2003.

[7] S. Bestaoui. One solution for the nondeterminism problem in the SCEPTRE 2 fault tolerance technique. In *Euromicro Workshop on Real-Time Systems*, pages 352–358, Odense, Denmark, June 1995.

[8] T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *International Conference on Fault-Tolerant Computing*, pages 128–137, Munich, Germany, June 1998.

[9] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. In *ACM Transactions on Computer Systems*, pages 90–107, Munich, Germany, February 1996.

[10] N. Budhiraja, F. Schneider, S. Toueg, and K. Marzullo. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*. ACM Press - Addison Wesley, 1993.

[11] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *Symposium on Reliable Distributed Systems*, pages 43–50, West Lafayette, IN, October 1998.

[12] T. Dumitraş and P. Narasimhan. An architecture for versatile dependability. *DSN Workshop on Architecting Dependable Systems*, Florence, Italy, June 2004.

[13] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gubstafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *STTT (Software Tools for Technology Transfer) Special Issue on ASTEC*, 2001.

[14] S. Frolund and R. Guerraoui. X-ability: a theory of replication. In *ACM Symposium on Principles of Distributed Computing*, pages 229–237, Portland, Oregon, July 2000.

[15] H. Gaifman, M. J. Maher, and E. Shapiro. Replay, recovery, replication, and snapshots of nondeterministic concurrent programs. In *ACM Symposium on Principles of Distributed Computing*, pages 241–55, Montreal, Canada, August 1991.

[16] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *IEEE Symposium on Reliable Distributed Systems*, pages 164–173, Nurnberg, Germany, October 2000.

[17] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *IEEE Symposium on Reliable Distributed Systems*, pages 263–273, Lausanne, Switzerland, October 1999.

[18] Object Management Group. Fault Tolerant CORBA. OMG Technical Committee Document formal/2001-09-29, September 2001.

[19] M. Orgiyan and C. Fetzer. Tapping TCP streams. In *IEEE International Symposium on Network Computing and Applications*, pages 278–289, Cambridge, MA, October 2001.

[20] S. Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems*. PhD thesis, Technical University of Vienna, Vienna, Austria, April 1994.

[21] S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, vol. 49, no. 2, pages 100–111, 2000.

[22] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pages 299–319, 1990.

[23] R. W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Publishing Company, 2001.

[24] J. H. Slye and E. N. Elnozahy. Support for software interrupts in log-based rollback-recovery. *IEEE Transactions on Computers*, vol. 47, no. 10, pages 1113–1123, 1998.

[25] T. Wolf. *Replication of Non-Deterministic Objects*. PhD thesis, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, November 1988.