Living with Nondeterminism in Replicated Middleware Applications

Joseph Slember and Priya Narasimhan

Carnegie Mellon University, Pittsburgh PA 15213, USA
jslember@ece.cmu.edu, priya@cs.cmu.edu

Abstract. Application-level nondeterminism can lead to inconsistent state that defeats the purpose of replication as a fault-tolerance strategy. We present Midas, a new approach for living with nondeterminism in distributed, replicated, middleware applications. Midas exploits (i) the static program analysis of the application's source code prior to replica deployment and (ii) the online compensation of replica divergence even as replicas execute. We identify the sources of nondeterminism within the application, discriminate between actual and superficial nondeterminism, and track the propagation of actual nondeterminism. We evaluate our techniques for the active replication of servers using micro-benchmarks that contain various sources (multi-threading, system calls and propagation) of nondeterminism.

1 Motivation

Replication is a common technique used to build fault-tolerant, distributed systems. The idea behind replication is the creation and distribution of multiple, identical copies (replicas) of a component across a system so that the failure of a replica can be masked by the availability of the other replicas. Determinism is a fundamental property required in order for replication to work. A component is said to be deterministic if it contains no characteristics that could cause replicas to become inconsistent with each other. In other words, identical replicas, when started from the same initial state and supplied the same ordered sequence of input messages, should reach the same final state and produce the same output.

A simplistic, but effective, strategy is to disallow the use of any nondeterministic functionality within applications that are to be replicated – effectively, this forbids the use of multithreading, shared memory, local I/O, system calls, random numbers, timers, etc. This is, in fact, the approach adopted by industrial standards, such as Fault-Tolerant CORBA [14].

Clearly, this approach is unrealistic for real-world applications that wish to use all of these nondeterministic functions. Current approaches to handling nondeterminism, covered in Section 8, allow nondeterminism to exist within the application, but handle it transparently. Transparency has its accompanying benefits, but does not exploit application-level information that might facilitate the handling of nondeterminism. In addition, architecture/application programmers often need to able to exercise control and "want to worry about replica

M. van Steen and M. Henning (Eds.): Middleware 2006, LNCS 4290, pp. 81–100, 2006.

configuration, intervene in failure detection or enabling explicit synchronization between replicas" [21]. With this motivation, we have developed a program-analysis approach to handling all forms of nondeterminism (including system calls and multithreading) – this allows us to *exploit application-level insight in handling nondeterminism.* Active replication is the predominant replication style that falls prey to nondeterminism. Therefore, our techniques are focused on how to handle nondeterminism in architectures using active replication. However, our techniques are easily applicable to other replication styles as well.

Contributions of this paper: Our previous research [18] showed that program analysis could assist in handling one specific form of nondeterminism, namely, system calls, such as `gettimeofday`. In our enhanced approach, Midas, described in this paper, we handle all forms of nondeterminism, including multithreading and contaminated nondeterminism. More specifically, the contributions of this paper include the following:

- Taxonomy and technique that distinguishes between nondeterminism that is superficial (looks like a nondeterministic call, but its effects do not lead to replica divergence) vs. actual (effects do lead to replica divergence) – this allows us to be discriminating in that we only need to worry about *addressing the actual, and not the superficial, nondeterminism*;
- Tracking the *propagation (or "contamination") of nondeterminism* through the application code – this allows us to capture the effects of nondeterministic execution and variables on otherwise deterministic code;
- Design and empirical evaluation of various application-centric *performance-sensitive techniques that compensate for the nondeterminism* that we detect and track – these techniques range from re-executing the contaminated nondeterminism to transferring the entire application state.

2 Taxonomy of Nondeterminism

Program analysis allows us to identify the true causes in the divergence of replicated state. Application state can be classified into one of three mutually exclusive categories: pure nondeterminism, contaminated nondeterminism, and pure determinism.

1. Pure nondeterminism: This covers any function that is the originating source of nondeterminism and that affects the server's state. Examples include system calls such as `gettimeofday` or `random`, all inputs, and all `read` calls that change the server's state nondeterministically. An example is

```
for (int j = 0; j < 100; j++ ) foo[ j ] = random();
```

Shared state among threads also falls within this category. However, we treat shared state in a special way – each access of shared state by a thread is considered to be a separate source of nondeterminism. For example, consider a single shared variable between two threads; if each thread accesses this variable four times, then, there exist eight separate instances of pure nondeterminism. It is immaterial that these eight instances happen to involve the same variable. This

view of shared state among threads frees us from having to worry about thread interleaving or the actual point in time when the threads execute.

2. Contaminated nondeterminism: This covers state that has any dependency, direct or indirect, on an instance of pure nondeterminism. Contaminated state captures the effect of pure nondeterminism when it propagates to the rest of the application. In other words, the pure nondeterministic state marks the beginning of nondeterministic execution. Anything that the pure nondeterministic state then touches is contaminated. If there was no pure nondeterminism, then, there would be no contamination. An example is the contaminated variable `bar` that depends on the purely nondeterministic variable `foo`:

```
for (int j = 0; j < 100; j++ ) {
  foo[ j ] = random();
  bar[ j + 100 ] = foo[ j ]; }
```

3. Pure determinism: This covers state that has no dependency whatsoever on the identified pure nondeterminism. This category of state will always be consistent across all server replicas. Assuming that the values in `bar` are initialized to zero, an example is:

```
for (int j = 0; j < 100; j++ ) bar[ j ] = bar[ j ] + 10;
```

4. Superficial nondeterminism: This falls under the category of pure determinism, but might be misclassified if a transparent approach to handling nondeterminism were used. In this category, a nondeterministic call is executed, but the end-result does not affect the application's persistent state and does not contaminate the rest of the application, either. An example is:

```
int a = random(); b = 5; return b;
```

Here, variable `a` is nondeterministic, but its value does not affect the server's state. More realistic examples of superficial nondeterminism are not shown here due to lack of space. A significant source of superficial nondeterminism arises in multithreaded applications where threads do not share any variables and do not modify any persistent application state, or where the shared state is split up across the threads such that each thread has its own distinct piece of state.

The value of this taxonomy, lies in its utility in compensating for nondeterminism. Only pure and contaminated nondeterminism need to be addressed for replica consistency – the other categories (pure determinism and superficial nondeterminism) can be disregarded. Thus, the compensation overhead will depend on the relative amounts of each category within an application.

3 Objectives

Our aim is to permit programmers to continue to create distributed applications that are nondeterministic (e.g., containing performance features such as multithreading) and yet allow these applications to be made fault-tolerant. Midas is independent of the target application and middleware and could be readily applied to any distributed, nondeterministic application.

In this paper, we exploit client-server middleware as the vehicle for exploring the issues underlying nondeterminism. In particular, we target CORBA C++ applications for the application of Midas. MEAD [12], the fault-tolerant middleware that we use, enables CORBA applications to be made fault-tolerant in multiple ways, including active, or state-machine, replication [17]. With the active replication of a server, every server replica receives and processes each request; every server replica also sends a response to the client, leading to duplicate responses that need to be filtered. The MEAD infrastructure performs this filtering and delivers only one response to the client, thereby masking the server's replication from the client. Clearly, for active replication to work, the server replicas must receive the same set of messages in the same order, which MEAD assures because it conveys messages over the underlying totally-ordered group communication system, Spread [3]. Active replication traditionally requires the supported application to be deterministic; however, we relax this requirement to allow MEAD to support the active replication even of applications containing nondeterministic features.

Midas' approach involves a synergistic combination of two aspects: *compile-time* knowledge with *run-time* compensation. By exploiting program analysis to isolate the possible places where nondeterminism can affect the system state or behavior, we then perform code transformations (that do not violate application semantics or expected functional behavior) to ensure consistent results across all of the replicas. We offer the programmer various options to deal with nondeterminism. A side-benefit of our analysis lies in its software engineering aspect. Because our program analysis tracks all live variables and their dependencies on detected nondeterminism, we can assess to what extent nondeterminism pervades the application. This information can be beneficial to the application programmer in understanding the trade-offs and deciding between various choices in compensating for nondeterminism.

Assumptions. Midas relies on having complete access to the application's source code, along with the ability to modify it prior to deployment. Specifically, we assume that we are allowed to modify the source code for the client, the server, and the IDL interfaces of all objects. Both the client and server source code must be available for analysis, although only the server is replicated. We also assume that all of the application state can be determined statically – thus, program analysis techniques that can handle dynamic state (e.g., dynamically allocated variables whose size is unknown at compile time) are outside the scope of this paper. Pointer-aliasing analysis is currently outside the scope of the techniques highlighted in this paper; our most recent work does incorporate advanced compiler techniques to handle dynamic memory and pointers.

For the purpose of this paper, and to describe how we handle *application-level nondeterminism*, we assume the deterministic, reproducible behavior of the operating system and the underlying middleware. While we make this simplifying assumption in order to demonstrate our approach to handling nondeterminism, we emphasize that Midas is general enough that we could apply it equally to the middleware/OS source-code and address their inherent nondeterminism as well, as describe in [19]. We also require homogeneous platforms, i.e., all of the

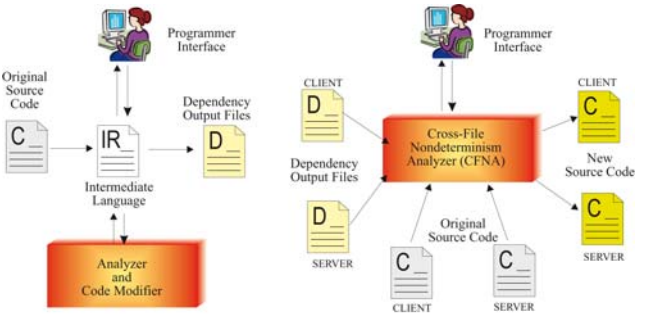


Fig. 1. Midas' program analysis framework for analyzing nondeterminism

replicas of the application must be hosted over identical hardware and operating systems; future versions of our approach will be extended to cover heterogeneous platforms. We assume an independent-failures model across distinct nodes and replicas, and aim to tolerate crash and communication faults.

4 Program Analysis Framework

To perform program analysis, we needed to convert the C++ CORBA application source-code into an intermediate format that is more suitable for program analysis. We first transformed our target C++ applications into C code using EDG [1], and then used the SUIF2 [2] compiler to transform the resulting C code into the intermediate representation. Conversion from C++ to C allows for easier analysis because it eliminates some complexities (e.g., object-oriented issues) that C++ introduces. It also allows us to leverage current compiler tools that expedite the transformation of C code into a workable, efficient intermediate form (referred to as an annotated parse-tree henceforth).

As shown in Figure 1, Midas' 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 that variable as potential (contaminated) nondeterminism. For each source file, the analyzer creates a dependency file that captures the nondeterministic behavior of the source code in that file. We then modify the original application source-code to insert specific code-snippets for the tracking and subsequent compensation of nondeterminism.

Enhancements to Analysis Framework. In our current program-analysis framework, we use SUIF to generate the initial abstract syntax tree (AST). All of the subsequent application analysis-passes are custom extensions to SUIF because of our specific needs in analyzing nondeterminism. For instance, our enhanced Midas framework supports thread analysis, as long as we can statically determine the entry, exit, and launch of all threads. In addition, we perform a complete dependency analysis to identify not only pure nondeterminism, but also the contaminated state that depends on it.

Some information is lost in the conversion from C++ to C, and we traverse the C++ code to mark up the SUIF-generated AST tree to fill in this information. The declaration of variables needs to be updated as scope is defined differently in C and C++, and this can affect the dependency chain between variables. For instance, in C++, the conditional block within an `if`, `while`, `do-while`, or `for` is considered to be a new scope, unlike in C. Another example of lost information relates to exception-handling code in `try-catch` blocks; `try-catch` blocks that form the top-level statements of functions, constructors, or destructors must be updated because they can affect the propagation of exceptions. Midas' current automated generation and insertion of code to handle our categorized nondeterminism includes:

- Tracking to assign unique identifiers to nondeterminism that is embedded within specific elements of a non-scalar data structure (e.g., nondeterminism that affects only one element of an entire array);
- Data structures to hold the variable-size state of the application;
- State-transfer operations (`get_state` and `set_state`) to copy state back and forth from the application into the appropriate data structures for transfer over the network;
- Execution that re-generates the contaminated state from the pure nondeterministic state, only if the latter has been transferred.

Data-Flow Passes. We perform multiple passes over the annotated parse tree. The first set of passes identifies all of the persistent state within the server code. Ultimately, this represents the only state that might be affected by nondeterminism and the state that we need to worry about for consistent server replication. The second set of passes identifies the pure nondeterminism within the application; these passes find and mark nondeterministic system calls, inputs, I/O, etc. Shared state between threads is initially considered as potentially nondeterministic, and another pass is made to discover all accesses to this shared state; these accesses are then marked as pure nondeterminism. Subsequently, these accesses are treated as sources of nondeterminism in their own right, and effectively constitute state. `def-use` chains (that determine where a specific variable is defined, and where it is used or assigned to another variable) are then calculated for all marked pure nondeterministic variables – this represents the first phase of dependency-tracking.

Control-Flow Passes. The next phase involves evaluating all of the possible execution paths that the server code might take. We determine the order of variable assignments along a particular control path, and for every discovered control path, we link together the `def-use` chains that we determined in the previous data-flow phase. This allows us to calculate dependencies of every variable for every possible execution path. Carrying this argument forward, we can now mark as contaminated nondeterminism all of the state that depends on the pure nondeterministic state identified in the data-flow phase. This is recursive – as we mark more contaminated state, we need make further passes to determine if there are further dependencies on this newly discovered contaminated state. We

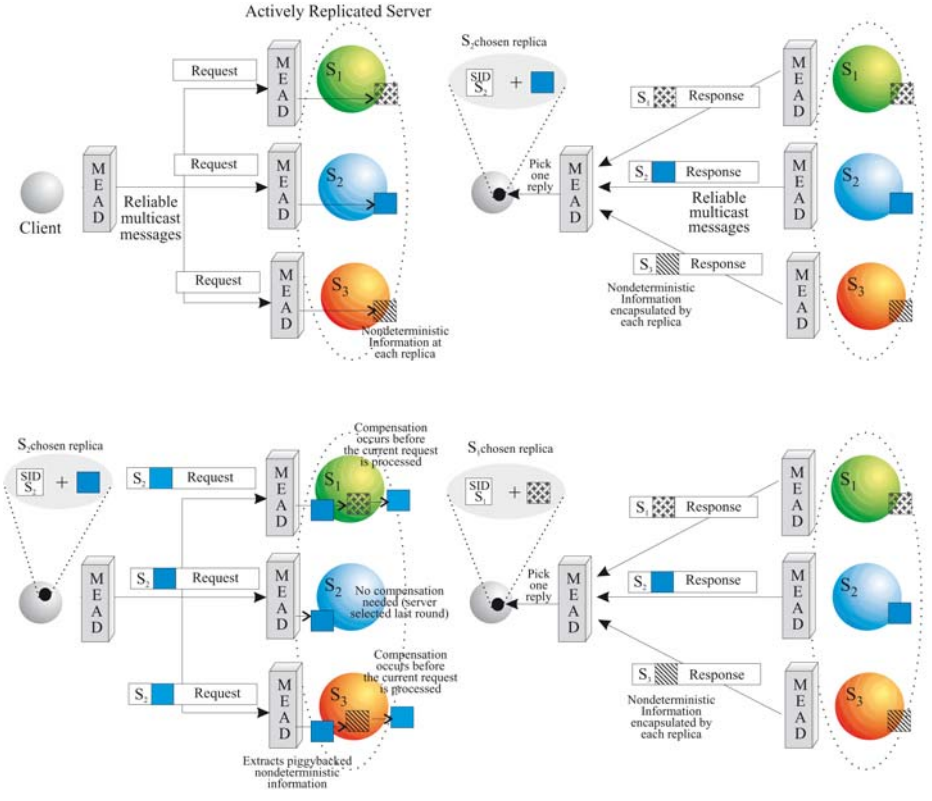


Fig. 2. Underlying approach for Midas' various compensation techniques. The techniques differ in the nature/amount of the information passed back and forth between the client and the server, and in the actual compensation work done on the server-side.

perform an exhaustive search of the server source-code to ensure that all such contaminated state is found. All persistent state that remains unmarked at the end of the control-flow phase can be considered as pure determinism.

5 Midas' Compensation Approaches

During our compile-time phases, we insert the compensation and state-transfer code snippets that will actually execute at runtime within the application. In this section, we describe how and when these code-snippets accomplish the compensation. For the remainder of the text, we assume that the server is actively replicated.

In our approach, the client is an integral participant in the compensation of the server's nondeterminism. Consider any two consecutive requests from the client to the replicated server, as shown in Figure 2. Each server replica piggybacks the relevant information(this information is specific to the technique described

below) about its nondeterminism to the client in its response to the first request. Then, this information, piggybacked onto the second request, is echoed by the client to all of the server replicas so that they can perform individual, local compensation actions *before* they begin to process the second request. All of the piggybacked nondeterministic information, as well as its associated transfer and compensation code, is generated by our compile-time phase, without burdening the application programmer.

We emphasize here that the server replicas do not need to be in lock-step synchronization in order to do this – each replica proceeds asynchronously to service its incoming, totally-ordered requests and to return responses. Thus, through the runtime execution of our inserted compensation snippets, each replica is rendered logically identical with its peers before it starts to process any new request from the client; between requests, the server replicas (if each's internal state is inspected individually) might, in fact, be divergent in state. However, this out-of-band divergence does no harm because it does not compromise the fault-tolerance of the application. If a replica fails or is recovered, it will simply be rendered consistent with the others at the *start* of the next new request. In Section 7, we address how this divergence becomes an issue when multiple clients are involved, with each controlling some part of the compensation.

All of our performance-sensitive compensation techniques undergo two rounds of client-server interaction for compensation, as shown in Figure 2. However, they differ in the amount and nature of compensation work done at the server replica and the amount/kind of relevant information transferred back and forth between the client and the server replicas. While all of our techniques are common in exploiting program analysis, the range of choices allow an application programmer to make an application-centric, performance-sensitive choice in compensating for nondeterminism. The techniques described below can be broadly classified as:

- Transfer of state, or the `transfer-*` techniques: `transfer-ckpt`, `transfer-diff-ckpt`, `transfer-contam` and `transfer-contam-track`;
- Re-execution of code, or the the `reexec-*` techniques: `reexec-contam` and `reexec-contam-track`.

In Figure 3, we depict the decision process that an application/system developer would undergo in order to decide among the various techniques.

5.1 Full-Checkpoint Transfer (`transfer-ckpt`)

After processing each request, every replica marshals its entire state (checkpoint) and passes this state, along with its response, to the client. The client accepts the first response[1], stores the identifier of the corresponding (selected) replica

[1] The client always sees only one response from the entire set of replicas because MEAD delivers the first-received response from the replicated server and suppresses the other responses. The replica whose response makes it first to the client is called the *selected replica*in the processing of the client's *next* request. The selected replica can vary from one request to the next, and is not dictated by the client or the server.

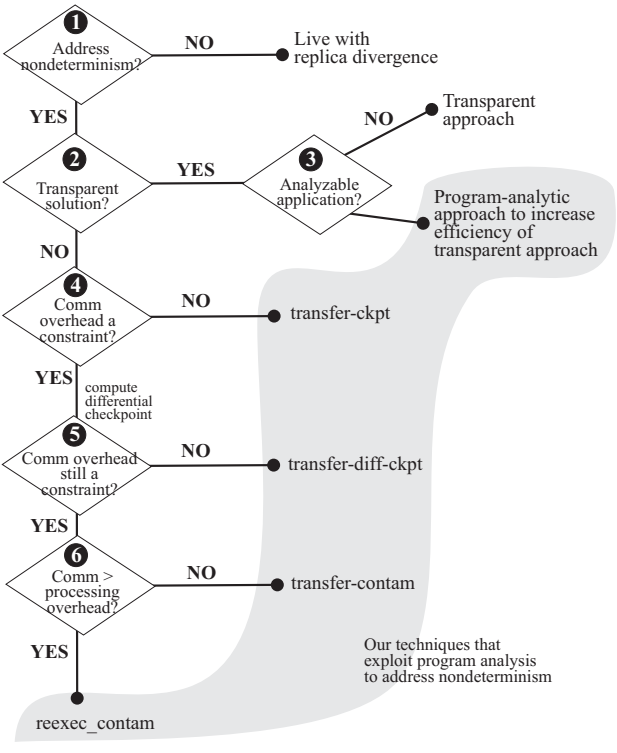


Fig. 3. Decision tree for determining appropriate technique for handling nondeterminism in an application. (1) Yes: Nondeterminism must be dealt with; No: Either no nondeterminism or can live with potential replica divergence. (2) Yes: Application code cannot be modified or designer prefers a transparent approach; No: Application code can be modified. (3) Yes: Pure deterministic code can be highlighted by program analysis, enabling a more efficient transparent technique that addresses only actual, and not superficial, nondeterminism; No: Program analysis cannot be performed on application source code, requiring a transparent approach that unnecessarily handles even superficial nondeterminism. (4) Yes: Communication overhead is an issue and a more efficient technique must be found. Only the state that has changed needs to be handled; No: Communication overhead is not a constraint and `transfer-ckpt` technique can be used. (5) Yes: Communication overhead is still a constraint and further analysis is required; No: Communication overhead is within reason and `transfer-diff-ckpt` can be used. (6) Yes: Communication overhead is a greater constraint than processing overhead, and `reexec-contam` can be used; No: Processing overhead is a greater constraint than communication overhead and `transfer-contam` can be used.

and that replica's state. On its next request to the server, the client piggybacks this saved information (including the checkpoint of the selected replica).

Each receiving replica examines this information to see if it was the selected replica, at the client-side, for the previous request. The selected replica does

not need to compensate and can proceed with processing the current request; a replica that was not selected by the client in the previous round must apply the piggybacked checkpoint before proceeding to service the current request. Thus, these checkpoints are passed back and forth between the client and the server to ensure replica consistency. Effectively, the compensation is as if a new replica was started and a fresh checkpoint was transferred to it, except that, in our case, the checkpoint is funneled through the client in its next request.

5.2 Differential-Checkpoint Transfer (`transfer-diff-ckpt`)

We instrument the application code in all of the places where the processing of a request might modify its state. Clearly, not all of these potential state-change points might actually be executed when the server processes a request. At runtime, only the actually executed change-points are captured and the associated state (called a differential checkpoint) transferred to the client. The remainder of the technique is similar to `transfer-ckpt`. Compared to `transfer-ckpt`, we have increased static code growth due to the additional instrumentation. There should be a slight increase in runtime server-side latency due to the additional scaffolding code required to track variables. This technique performs best when the scaffolding latency is outweighed by the benefit in communication latency obtained with transferring the differential checkpoint vs. the full checkpoint.

5.3 Transfer Contaminated-Nondeterminism (`transfer-contam`)

The `transfer-ckpt` and `transfer-diff-ckpt` techniques do not discriminate between actual and superficial nondeterminism. In the `transfer-contam` technique, each server replica piggybacks only its actual nondeterministic state (both pure and contaminated) back to the client.

Based on the output of our data-flow and control-flow analyses, we create a server-side `struct` that holds the pure and contaminated nondeterminism within each replica. Because this `struct` needs to be marshaled over the standard middleware protocol, we need to augment the IDL interface specifications of the server so that this nondeterministic `struct` contains (and serves as) the return value of the server's methods and is also an input parameter to the server's methods – this allows us to piggyback the nondeterministic `struct` onto messages passed back and forth between the client and the server. The remainder of the algorithm is similar to the `transfer-ckpt` technique, except that the client-side and the server-side extract, copy, and piggyback the nondeterministic `struct` instead of a checkpoint.

5.4 Reexecute Contaminated-Nondeterminism (`reexec-contam`)

We insert prepared portions of code that can be executed to re-generate the contaminated nondeterminism, if provided the pure nondeterminism (i.e., the origin of the contamination) as an input. In `reexec-contam`, every receiving

server replica extracts the piggybacked nondeterministic `struct`, as in `transfer-contam`. As with all of the other techniques, the selected replica for one request has no compensation work to do for the next request. On the other hand, each of the remaining (non-selected) replicas for a request performs compensation, before processing the next request, by first setting the pure nondeterministic part of its state to the received nondeterministic `struct`, and then re-executing the inserted code-snippets to regenerate the corresponding contaminated nondeterminism. At the end of this compensation, each replica is consistent and is ready to process the current request.

Compared to `transfer-contam`, the `reexec-contam` technique should incur lower communication overheads due to the reduced amount of nondeterministic state being piggybacked back and forth; however, the tradeoff is that runtime latency is increased by the reexecution of the compensation snippets at the server side. Also, `reexec-contam` requires more compile-time analysis and source-code modification to the server-side than `transfer-contam`. This is because additional control-flow passes are needed to isolate the code that encapsulates the contaminated nondeterministic state. The client-side code is the same as in `transfer-contam`.

Obviously, reexecution is justified when the compensation overhead is outweighed by the communication overhead of the `transfer-*` techniques.

5.5 Incorporating Tracking (`transfer-contam-track`, `reexec-contam-track`)

The complexity of the data structures that constitute application state, along with the way these structure are accessed or referenced, affects how we track changes in that application's state. The nondeterministic `struct`s that we create for compensation purposes must be flexible and able to hold a dynamic amount of information, ranging from no state all the way to a full checkpoint. We use the CORBA `sequence` type for this purpose because it can hold, and marshal over the wire, a dynamic amount of information.

If state variables are all scalar types (e.g., `int a`), then, there is no need for tracking. However, if data structures are more complex or non-scalar (e.g., `int a[10000]`) , then, additional information might be needed to track which of the member items of the non-scalar structure have changed.

To cover the worst possible case, we identify each piece of state with an additional identifier. This identifier can be used to directly reference its associated piece of state. For example, if `int a[500]` is a part of the state, then another shadow array of the same size is created to hold the indexes of array `a[]`. If only one value in the array `a[]` changes at runtime, the shadow array tracks the change and allows us to know which index in `a[]` changed. The additional compile-time work to support tracking is minimal because it involves creating sequences of `long`s to hold all the identification information to reference non-scalar types.

5.6 Additional Clarification

The above techniques encapsulate all of the nondeterminism that is present in a distributed application. However, nondeterminism might be introduced if different replicas of the same server talk to different external servers. In other words, we assume that a replicates server receives the same messages in the same order using totally ordered multicast. Therefore, consistency is maintained and nondeterminism is handled properly in the above techniques.

Midas' techniques will handle *all* nondeterminism that is present in an application. This, however, can present a problem if the nondeterminism is built into the application for a specific reason and, therefore, should not be compensated for. In order to allow for nondeterminism to exist in the application without being compensated for, it is possible for a programmer to mark parts of code or variables that Midas would consider deterministic and, therefore, would not handle by its compensation techniques. Additionally, we could allow the programmer to specify when and/or what replicas responses would be used for the compensation. This would allow for greater control for the application programmer and for more flexibility in the architecture. However, this is outside the scope of this paper, even though the implementation would be relatively straight-forward.

The main idea behind using program analysis to handle nondeterminism is to target only the nondeterminism that actually causes replica divergence. Thus, it should not result in higher overheads than other transparent approaches, such as full-state transfer. While it is possible that an application will be strife with nondeterminism and, therefore, will involve significant overhead on Midas' part, this overhead should not exceed that of a basic transparent approach.

6 Experimental Evaluation

Because our techniques are non-transparent, the overheads that we incur should be directly proportional to the amount of actual nondeterminism that exists within the application, e.g., if only 5% of the application is actually nondeterministic, our compensation overheads should be incurred only for that portion of the application. We also note that the runtime overheads and behavior of MEAD will undoubtedly influence our runtime overheads. Where possible, we distinguish between MEAD's performance and our compensation performance.

We conducted our experiments using the Emulab distributed environment [22], with a homogeneous test-bed of nodes that each run the RedHat 9 Linux, 2.4.18 kernel operating system on a 850MHz processor, 256KB cache, and 512MB RAM over a 100 Mbps LAN. We use MEAD version 1.5 that uses Spread version 3.17.3 as its group communication protocol. In our experiments, we do not load the nodes with any other running programs other than MEAD, Spread, our micro-benchmarks, and the native OS utilities that typically run on each node. Each replica runs on a separate node.

We evaluate a number of metrics (communication overhead, compensation overhead, server-side processing time, and round-trip time) under fault-free conditions.

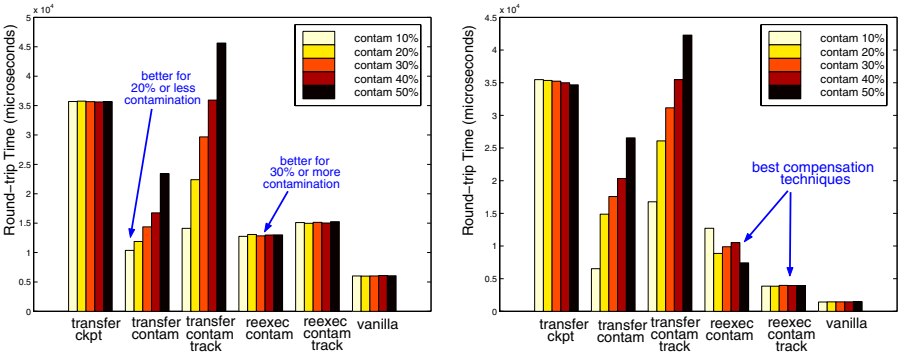
Table 1. Description of the various micro-benchmarks

Compensation technique	`no_sha` micro-benchmark	`sha` micro-benchmark
`vanilla` (baseline)	Replicas are nondeterministic and inconsistent; no compensation performed	Same as `no_sha`, except that a 20-byte digest is computed and stored at each replica at the end of each request
`transfer-ckpt`	Entire checkpoint piggybacked on each server's reply to the client, compensation according to Section 5.1	Same as `no_sha`, with digest considered part of the checkpoint and piggybacked on each server's reply
`transfer-contam`	Pure and contaminated nondeterminism piggybacked on each server's reply to the client, compensation according to Section 5.3	Same as `no_sha`, with digest considered part of the contaminated nondeterminism
`transfer-contam -track`	Same as `transfer-contam` above, but with tracking enabled	Same as `transfer-contam` above, but with tracking enabled
`reexec-contam`	Pure nondeterminism piggybacked on each server's reply to the client, contaminated nondeterminism re-generated through re-execution, compensation according to Section 5.4	Same as `no_sha`, with digest needing to be re-computed as a part of the re-execution
`reexec-contam -track`	Same as `reexec-contam` above, but with tracking enabled	Same as `reexec-contam` above, but with tracking enabled

6.1 Micro-benchmarks

We have developed two micro-benchmarks to compare our various compensation techniques. The two micro-benchmarks are identical in many ways. They both constitute a two-tier application, i.e., with a single client and a single replicated server. Both micro-benchmarks use multi-threading with homogeneous threads (to simplify experimentation), identical code at each of the server replicas (except for the fact that each replica stores a unique, hard-coded server_id SID), and identical initial state to start out with. The difference is that the `sha` micro-benchmark involves the computation of a 20-byte digest, and therefore, requires significantly more processing time at the server-side, as compared with the `no_sha` micro-benchmark. The two micro-benchmarks are compared in Table 1. The `sha` version is used to give an example of an application that has increased reexecution time.

Each micro-benchmark contains an array of 10,000 `long`s that represents its state. Pure nondeterminism involves generating a random number and assigning it to one of the elements in the array. Contaminated state is subsequently created by performing arithmetic on the random number and assigning the result to



(a) `sha` benchmark (b) `no_sha` benchmark

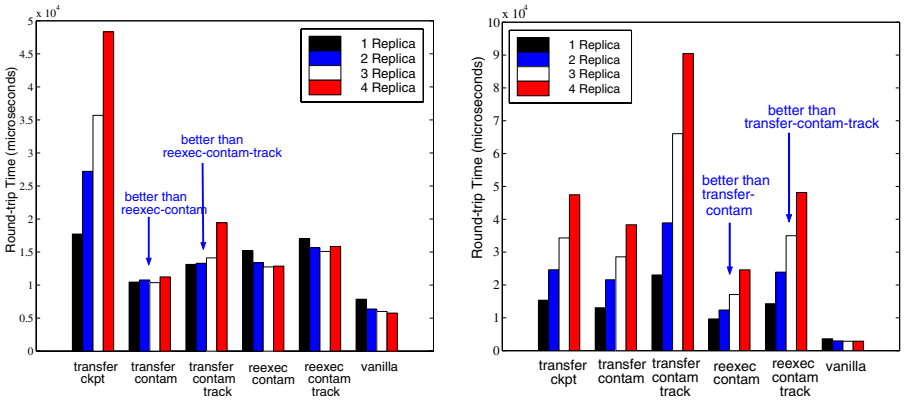
Fig. 4. Compensation approaches with varying amount of contaminated state for 10% pure deterministic state for the two micro-benchmarks. The cross-over between the `transfer-contam` and the `reexec-contam` is visible for both the tracking and the no-tracking cases.

another element in the array. The server state is changed in 15 different ways: varying the pure nondeterminism to 10%, 30% and 50%. For each value of pure nondeterminism, we vary the amount of contaminated nondeterminism to 10%, 20%, 30%, 40% and 50%. For each of the 15 state combinations, we evaluate each of our five techniques: `transfer-ckpt`, `transfer-contam`, `transfer-contam-track`, `reexec-contam` and `reexec-contam-track`. Note that we can compare all of the techniques for a given $x\%$ of nondeterminism. However, we cannot fairly compare a single technique for $x\%$ vs. $y\%$ of nondeterminism because these represent two entirely different applications (while the % of nondeterminism varies, the application is, in fact, functionally different). The `vanilla` case simply serves as a baseline for performance comparison. We also vary other parameters, such as the number of replicas (1–4), amount of multithreading (2–6 threads), and amount of state (100, 1000 and 10,000 `long`s).

6.2 Empirical Observations

Varying amount of contamination. Graph 4(b) shows the effect on the round-trip time of increasing the amount of contaminated nondeterminism within the `no_sha` micro-benchmark. The amount of pure nondeterminism for these results is fixed at 10%, and 3 replicas are used. Because pure nondeterministic state is handled identically across all of our various techniques, the graph demonstrates how each technique handles an increase in contaminated state.

The `transfer-ckpt` technique shows a fairly constant round-trip time regardless of the amount of contaminated state. The processing time increases slightly across all techniques because additional work is done due to the increased amount of contaminated state. However, the processing time is relatively small compared to the communication overhead of passing the entire state of back and forth.



(a) `no_sha` benchmark

Fig. 5. (a) `transfer-contam-*` techniques are better than `reexec-contam-*` techniques for increasing number of replicas for the `sha` micro-benchmark for 10% pure deterministic state and 10% contaminated state. (b) `reexec-contam-*` techniques are better than `transfer-contam-*` techniques for increasing number of replicas for the `no_sha` micro-benchmark for 50% pure deterministic state and 30% contaminated state.

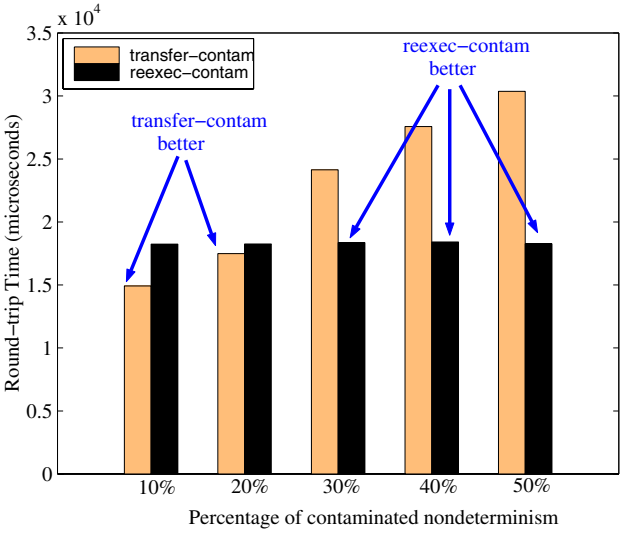


Fig. 6. Cross-over between the `transfer-contam` and the `reexec-contam` techniques for the `sha` micro-benchmark for 30% pure deterministic state

The `transfer-contam-*` techniques show a linear increase in round-trip time with increased amount of contaminated state. This is because the communication overhead is proportional to contaminated state. Note that `transfer-contam-track` has the higher overheads of the two because more information is

being passed by the replicas. Also, `transfer-contam-track` becomes worse than `transfer-ckpt` when more than 50% of the state is nondeterministic.

There is very little change in the round-trip time of the `reexec-contam-*` techniques with increased contaminated state because the communication overhead dominates over re-execution time. Again, `reexec-contam-track` has the higher overheads of the two. We observe that the `reexec-contam-*` techniques are better than their `transfer-contam-*` counterparts.

Figure 4(a) shows the effect on the round-trip time of increasing the amount of contaminated nondeterminism within the `sha` micro-benchmark. The amount of pure nondeterminism for these results is fixed at 10%, and 3 replicas are used. Note that the sha1 algorithm has a significant amount of processing time; this is readily visible when comparing these results with their `no_sha` counterparts.

The same trends are seen as in Figure 4(b). The most interesting observation here is due to the fact that communication overhead does not dominate processing time. For instance, with 10% and 20% contamination, `transfer-ckpt` appears to have lower overheads. Once contamination reaches 30% or more, `reexec-contam` once again displays lower overheads. This is because the increased processing time outweighs the communication overhead for lower amounts of contaminated state.

Varying degree of replication. In Figures 6.2 and 5(a), the amount of pure and contaminated nondeterminism is constant, but the number of replicas is varied. Figure 6.2 shows the `sha` micro-benchmark for 10% pure nondeterminism and 10% contaminated nondeterminism. Figure 5(a) shows the `no_sha` micro-benchmark for 50% pure nondeterminism and 30% contaminated nondeterminism. Note that, for every additional replica, the communication load increases because all of the replicas send their nondeterministic state, along with their responses, to the client.

In Figure 6.2, all of the techniques. except for `transfer-ckpt`, demonstrate a minimal increase in round-trip time with increased number of replicas. This is because, apart from `transfer-ckpt`, which sends the entire state over, the other techniques only deal with 10% pure and 10% contaminated nondeterminism. Because the communication overhead is relatively lower due to the small amount of nondeterministic state, `reexec-contam` performs worse than `transfer-contam` technique, except in the 4-replica case where the communication overhead overcomes the re-execution time. Thus, the number of replicas, along with the amount of transferred state, can dictate which technique is appropriate for a given application.

Figure 5(a) demonstrates lower processing time with higher communication overhead. As in the previous case, the tracking counterpart of a technique adds more overhead than its corresponding no-tracking version. Here, `reexec-contam` is always better regardless of the number of replicas. In fact, with an increased number of replicas, the relative performance of the `reexec-contam` technique becomes markedly better.

Trade-offs. Figure 6 shows the round-trip time for the `sha` micro-benchmark with the amount of pure nondeterminism fixed at 30% for 3 replicas, and with the amount of contaminated state varying from 10-50%. We focus only on the

performance of the `reexec-contam` and the `transfer-contam` techniques. The `reexec-contam` technique shows relatively no change as contaminated state increases because of the overwhelming communication overhead and the low processing time. The `transfer-contam` technique demonstrates a linear increase in overhead with respect to the amount of contaminated state. This graph clearly shows the cross-over between the two techniques, demonstrating that no technique works for all cases to provide the best performance. Many factors, including the number of replicas, the amount of contaminated state, the communication overhead, the processing overhead, etc., need to be weighed in deciding which technique is appropriate. Figures 6.2 and 5(a)also support our insights about the trade-offs between re-execution vs. the transfer of contaminated state, based on the relative amount of communication overhead and processing time.

Code growth. Code growth is inevitable in our technique. The `transfer-ckpt` technique will typically have the least code growth because it is perform simple checkpointing. The `transfer-contam` technique is next in code growth; `transfer-contam-track` will have even larger code growth. The `reexec-contam` will likely have the largest code growth of the all of techniques, because of the inserted compensation snippets. However, we note that `reexec-contam` will have smaller code growth if the amount of contaminated state as large and the re-execution snippets are small. Thus, while code growth matters and should be considered, using it as a metric for comparison might be subjective since it depends on the application's functionality.

7 Future Work

We note that our current implementations of the `transfer-*` and `reexec-*` techniques leave much room for optimization, but efficiency considerations form a part of our ongoing investigation. Multi-tier applications and nested end-to-end requests introduce increased complexity in handling nondeterminism, especially with actively replicated tiers. The propagation of nondeterministic state is no longer contained at the client or at any one tier. We need to handle any nondeterministic state or execution that propagates to other tiers. This is especially evident when a failure occurs during an end-to-end request, resulting in some of the replicas at every tier becoming inconsistent. Multiple clients can also complicate the techniques described in this paper because each client is an active participant in the back-and-forth compensation of nondeterminism, and we would then require coordination across clients or some alternative way of ensuring consistency across multiple clients. Both multi-tier and multi-client fault-tolerant architectures are part of our ongoing research on the scalable compensation of nondeterminism, but remain outside the scope of this paper.

8 Related Work

Gaifman [10] targets nondeterminism that arises in concurrent programs due to environmental interaction. This technique involves backup replicas lagging

behind the primary to ensure consistency. The technique is transparent to the user, but the application is actually modified by transformations that handle multithreading. The Multithreaded Deterministic Scheduling Algorithm [11] aims to handle multithreading transparently by providing for internal and external queues that together enforce consistency. The external queue contains a sequence of ordered messages received via multicast, while each internal queue focuses on thread dispatching, with an internal queue for each process that spawns threads. Basile [5] addresses multithreading using a preemptive deterministic scheduler for active replication. The approach uses mutexes between threads and the execution is split into several rounds. Because the mutexes are known at each round, a deterministic schedule can be created. This approach does not require any communication between replicas.

Delta-4 XPA's semi-active replication [4] addresses nondeterminism through a hybrid replication style that employs primary-backup replication for all nondeterministic operations and active replication for all other operations. In SCEPTRE 2 [6], nondeterminism arises from preemptive scheduling. Semi-active replication is used, with deterministic behavior enforced through the transmission of messages from a coordination entity to backup replicas for every nondeterministic decision of the primary's. Similarly, Wolf's piecewise deterministic approach [23] handle nondeterminism by having a primary replica that actually executes all nondeterministic events, with the results being propagated to the backups at an observable, deterministic event.

The fault-tolerant real-time MARS system requires deterministic behavior [16] in highly responsive automotive applications that are nondeterministic due to time-triggered event activation and preemptive scheduling. Determinism is enforced using a combination of timed messages and a communication protocol for agreement on external events.

X-Ability [9] is based predominantly on the execution history resulting from previous invocation. The approach is not necessarily transparent to the programmer because the proposed correctness criterion must be followed for consistency. The advantage is that it is independent of the replication style. Slye et al. [20] track and record the nondeterminism due to asynchronous events and multithreading. The nondeterministic executions are recorded so that they can be replayed to restore replica consistency in the event of rollback.

The Transparent Fault Tolerance (TFT) system [7] enforces deterministic computation on replicas at the level of the operating system interface. The object code of the application binaries is edited to insert code that redirects all nondeterministic system calls to a software layer that returns identical results at all replicas. Hypervisor-based fault tolerance [8] involves a virtual machine that ensures that all nondeterministic data is consistent across replicas. A simulator executes all environmental instructions, and then requires system-wide lock-step synchronization on this execution.

TCP tapping [15] captures and forwards nondeterministic execution information from a primary to other replicas. The backup replicas gain information from the primary after it has done the work. The approach is transparent, but involves

setting up routing tables to snoop on the client-to-server TCP stream, with the aim of extracting the primary's nondeterministic output. Zagorodnov et al. [24] target nondeterminism that is inherent to service protocols used by network servers. The solution involves the interception of I/O streams of replicas, and the appropriate handling of input and output streams.

9 Conclusions

We present Midas, a new approach, for living with nondeterminism in distributed, replicated applications by exploiting static program analysis on the application's source code, along with the runtime compensation of nondeterminism. We identify the sources of nondeterminism within the application, discriminate between actual and superficial nondeterminism, and track the propagation/contamination of nondeterminism within the application.

We describe two different techniques, one that involves the reexecution of contaminated nondeterministic code and another that involves the transfer of checkpoints or nondeterministic state. We can support even the active replication of nondeterministic applications in this manner. Our empirical evaluation involves various performance-sensitive techniques for distributed middleware micro-benchmarks that contain various sources (multi-threading, system calls and contamination) of nondeterminism.

Acknowledgements

We gratefully acknowledge the feedback that we received on early drafts of this paper from John Wilkes, Dan Siewiorek and Greg Ganger. This work has been partially supported by the NSF CAREER grant CCR-0238381.

References

1. http://www.edg.com/.
2. G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. *The Basic SUIF Programming Guide*.
3. Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *The International Conference on Dependable Systems and Networks*, pages 327–336, New York, NY, June 2000.
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, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *The International Conference on Dependable Systems and Networks*, pages 149–158, San Francisco, CA, June 2003.
6. 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.

7. T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Fault-Tolerant Computing Symposium*, pages 128–137, Munich, Germany, June 1998.
8. T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1), pages 90–107, Feb. 1996.
9. S. Frolund and R. Guerraoui. X-ability: A theory of replication. In *Principles of Distributed Computing*, pages 229–237, Portland, OR, 2000.
10. H. Gaifman, M. J. Maher, and E. Shapiro. Replay, recovery, replication, and snapshots of nondeterministic concurrent programs. In *Principles of Distributed Computing*, pages 241–255, Montreal, Canada, Aug. 1991.
11. R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Symposium on Reliable Distributed Systems*, pages 164–173, Nurnberg, Germany, October 2000.
12. P. Narasimhan, T. A. Dumitraş, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: Support for real-time fault-tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.
13. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *Symposium on Reliable Distributed Systems*, pages 263–273, Lausanne, Switzerland, Oct. 1999.
14. Object Management Group. Fault Tolerant CORBA. OMG Technical Committee Document formal/2001-09-29, September 2001.
15. M. Orgiyan and C. Fetzer. Tapping TCP streams. In *IEEE International Symposium on Network Computing and Applications*, pages 278–289, Cambridge, MA, Oct. 2001.
16. S. Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems*. PhD thesis, Technical University of Vienna, Vienna, Austria, Apr. 1994.
17. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
18. J. G. Slember and P. Narasimhan. Exploiting program analysis to identify and sanitize nondeterminism in fault-tolerant, replicated systems. In *Symposium on Reliable Distributed Systems*, pages 251–263, Florianopolis, Brazil, Oct. 2004.
19. J. G. Slember and P. Narasimhan. Nondeterminism in ORBs: The perception and the reality. In *Workshop on High Availability of Distributed Systems*, Krakow, Poland, September, 2006.
20. J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Fault-Tolerant Computing Symposium*, pages 250–259, Sendai, Japan, June 1996.
21. W. Vogels, R. van Renesse, and K. Birman. Six misconceptions about reliable distributed computing. In *ACM Special Interest Group on Operating Systems, European Workshop*, Sintra, Portugal, Sept. 1998.
22. B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002.
23. T. Wolf. *Replication of Non-Deterministic Objects*. PhD thesis, Ecole Polytechnique Federale de Lausanne, Switzerland, Nov. 1988.
24. D. Zagorodnov and K. Marzullo. Managing self-inflicted nondeterminism. In *Hot-Dep, International Conference on Dependable Systems and Networks*, pages 323-328, Yokohama, Japan, June 2005.