Transparent Fault-Tolerant Java Virtual Machine

Roy Friedman[*] Alon Kama
Computer Science Department
Technion – Israel Institute of Technology
Haifa, 32000
Israel
{roy,alon}@cs.technion.ac.il

Abstract

Replication is one of the prominent approaches for obtaining fault tolerance. Implementing replication on commodity hardware and in a transparent fashion, i.e., without changing the programming model, has many challenges. Deciding at what level to implement the replication has ramifications on development costs and portability of the programs. Other difficulties lie in the coordination of the copies in the face of non-determinism.

We report on an implementation of transparent fault tolerance at the virtual machine level of Java. We describe the design of the system and present performance results that in certain cases are equivalent to those of non-replicated executions. We also discuss design decisions stemming from implementing replication at the virtual machine level, and the special considerations necessary in order to support Symmetric Multi-Processors (SMP).

1. Introduction

Active Replication is one of the prominent approaches for obtaining fault tolerance [9, 29]. Specifically, a replicated system involves running multiple copies of the same program concurrently so that if one of the replicas crashes, the others can mask the failure.

One of the main complications in realizing fault tolerance by active replication is the fact that programs are usually not deterministic. In particular, a program's behavior is often influenced by external events, such as I/O, local clocks, and possibly other local environment aspects like process ID, scheduling, memory management, etc. Clearly, any attempt to provide a fault-tolerant infrastructure must handle this inherent non-determinism.

[*]This author is now affiliated with IRISA/INRIA, Rennes, France

The holy grail of replication-based fault-tolerance is finding a good balance between the transparency of replication to the application, the overhead that replication imposes, and the general tendency to use commodity hardware and operating systems. Transparency is vital in order to render legacy applications fault-tolerant, and in order to reduce the costs of developing and maintaining fault-tolerant systems. Alas, most users are only willing to pay a small performance penalty for the sake of fault-tolerance. Also, commodity hardware and operating systems are becoming a common requirement in order to reduce total costs (economy of scale) and to avoid falling behind the technology curve.

As discussed in [11], an interesting issue that is directly related to the above tradeoffs is at what level should replication be implemented. One option is to provide the replication logic in hardware. This achieves transparency and good performance, but incurs huge hardware development costs and complexity, and requires a new implementation for each realization of the target architecture. Another option is at the operating system level. However, operating systems are extremely complex, making it hard to provide correct replication semantics at this level. Moreover, the replication logic must be adjusted for each new version of the operating system. Yet another alternative is to expose the replication semantics to the application, possibly with the aid of a middleware, e.g., [5, 6, 8, 15, 17, 18, 19, 21, 26, 30, 31]. However, this complicates application design and maintenance, increases the probability of bugs since not all application developers are experts in fault-tolerance, and does not address legacy applications.

In contrast, the approach proposed by Bressoud and Schneider allows replicating a commodity-based system in a fully transparent manner by use of a hypervisor [11]. A *hypervisor* is a layer of software that implements a virtual machine having the same instruction-set architecture as the machine on which it executes. The hypervisor, thus, serves as an instruction-level interceptor, allowing to add functionality in a transparent manner. Implementing the replication logic at the hypervisor level addresses the main problems of the three alternatives presented above. The hypervisor-based approach incurs some performance overhead, but given its considerable benefits, it can be viewed as a viable solution.

In this paper, we report on FT-JVM, an implementation of fault-tolerance at the virtual machine level of Java, that applies similar techniques from the hypervisor-based approach to the JikesRVM [1, 4]. Our work only addresses non-malicious faults. We describe the design of the system, the alternatives we considered, and present a detailed performance analysis of the system we implemented. An interesting complication that we encountered was due to the multithreaded nature of Java, in particular when running on SMPs. The main challenge was how to solve the resulting non-determinism while maintaining good performance and processor utilization. Moreover, our work is unique in that we allow JIT compilation, yet do not modify the JIT or the compilation process in any way.

As virtual machine-based languages, such as Java and C#, become more common, our approach becomes compelling. In particular, much of the work in [11] was devoted to realizing a virtual machine, whereas in the case of virtual machine-based languages, it already exists. Moreover, by working at the virtual machine level, we only need to transfer non-deterministic events related to the application we wish to make fault-tolerant, rather than all events that occur in the computer, and thus obtain substantially lower replication overhead than [11]. Of course, relying on the virtual machine of one language has the limitation that it only applies to that language. On the other hand, Java has gained enough popularity to make it interesting, and the same solution works as is across multiple hardware architectures and operating systems. Microsoft's Common Language Runtime (CLR) provides a language independent virtual machine, making it a good candidate for future research.

Finally, for clarity of presentation, the discussion here assumes one primary and one backup. Yet, it can be extended trivially to any number of nodes n where one node is the primary and the others are backups. In this case, the backups are numbered, and in the event of a failure by the primary, one of the surviving backups is chosen deterministically, e.g., through a group membership service, to replace the failed primary.

2. System Model Assumptions

In this work, we made assumptions about the behavior of the underlying system and the structures of the applications that we aim to make fault-tolerant. For example, we assume fail-stop failures on the part of processors, where such failures are easy to identify and the failing processors are not Byzantine [24]. Furthermore, we assume that all processors are of a similar architecture, e.g., if the primary is a dual-CPU machine then the backup is also of that configuration. We rely on the replicas to house all the class files for the application.

The Java applications that are executed by the FT-JVM should not contain native methods or invoke external commands using the `java.lang.Runtime.exec()` method that access non-deterministic data, as this information will not reach the replication mechanism. When there is access to a file on a disk, we assume that the file resides on a shared drive that is accessible to the replicas should the primary fail.

On SMPs, we assume no data races within the application. That is, we assume that all accesses to shared data are guarded by the Java synchronization constructs.[1] This assumptions is not required on uniprocessors.

Environment variables that are accessed by the JVM are replicated from the primary to the backup. However, if the primary fails and a backup with different values to those variables takes over, we assume that execution will proceed correctly. That is, we assume that the underlying application will safely handle the change in the environment variables' values.

3. Virtual Machine Based Replication Coordination

3.1. JikesRVM

The virtual machine we chose to extend is JikesRVM [1] (Jikes Research Virtual Machine, formerly Jalapeño [4]). JikesRVM is an actively developed, open-source, multi-platform (IA32 and PowerPC) virtual machine. Furthermore, it is built in a way that may be adapted to support interpreted languages other than Java, which means that our implementation could potentially support even more applications.

Figure 1 outlines the main components of this virtual machine. Partially shaded modules indicate where changes were made to support fault tolerance; these changes will be discussed later. The replication module is a new component, and will also be presented later. JikesRVM itself is mostly written in Java. As such, the VM executes its own Java code, much like a compiler is used to compile a new version of itself. The VM runs atop an executable "boot image" that contains the bare essentials for bootstrapping. Besides the native bootstrap code there exists a thin layer containing native "magic" methods that access raw memory or make direct system calls that cannot be performed from

[1] A *data race* is defined as a concurrent access to a variable that is not guarded by a lock.

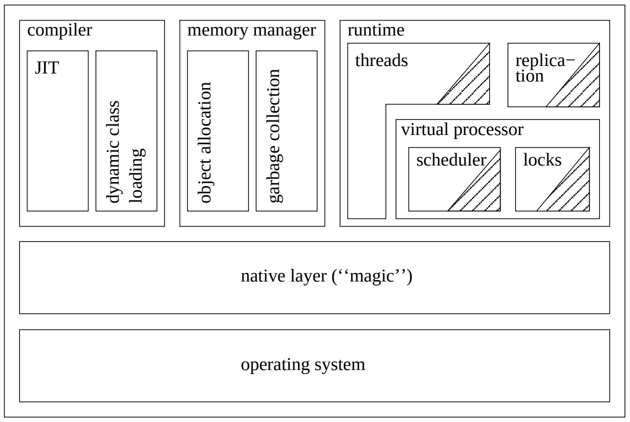

Figure 1. JikesRVM block diagram. Shaded components were modified by us.

within the Java code. Both of these modules are written in C.

The modules relevant to replication lie in the runtime subsystem. Here, we only discuss the components that are relevant to our work, namely the thread and scheduling mechanisms.

Java threads within JikesRVM are not mapped to system threads. Rather, they are multiplexed on “virtual processors”, which in turn are implemented as pthreads. Typically a virtual processor corresponds to a physical processor in the system so as to maximize the use of computational power. There is no difference between the core threads of the VM and the threads that it spawns for the application that is run on top of it; all lie in the same scheduling queues.

For technical reasons, JikesRVM’s threads are not fully preemptive. Rather, the just-in-time (JIT) compiler interjects “safe yield points” at the jumps into method prologues and the jumps to back-edges, such as the *else* of an *if-else* construct or the end of a *while* loop. There are two possible settings for deciding on which safe yield point a thread will be preempted. In the default configuration, a clock-driven timeout interrupt is used, and a yield will take place at the next safe yield point after such an interrupt is received. An alternate, deterministic configuration will use a counter to count the safe yield points encountered, and then yield after reaching a predetermined limit. We refer to such a sequence of yield points, which represents the maximum that a thread may proceed before being preempted, as a *time slice*.[2]

The thread scheduling algorithm is triggered when a thread encounters the quasi-preemptive yield that is described above, or in response to the invocation of the `yield` and `sleep` methods of `java.lang.Object` and the `wait` method of `java.lang.Thread`, or due to blocking I/O such as file or network access. The threads to be scheduled are stored in FIFO queues. These queues are associated with a specific virtual processor and there exists a load balancing mechanism to transfer threads between the processors. The scheduler checks for I/O availability using the `select` system call, and if no I/O-dependent threads can be scheduled, it turns to any of the ready threads. Otherwise it schedules the idle thread.

[2]In fact, what we need is deterministic scheduling. This ability of JikesRVM is simply a cheap way of obtaining it, since we do not need to log every scheduling decision.

Another place where the scheduler is invoked is upon lock contention. All higher-level VM locking mechanisms rely on the processor lock primitive. These processor locks are encapsulated as Java objects with a field that identifies the virtual processor (and therefore the thread) that owns the lock. If a lock cannot be acquired, the thread requesting that lock will yield.

3.2. Sources of Non-Determinism

There are many ways in which non-deterministic behavior can fork an execution. We set out to discover and disable them so that consistency may be maintained with minimal overhead.

Clearly, clock-driven scheduling is a likely source of non-determinism, and thus we decided to use the deterministic scheduler. Another scheduler concern is the actual scheduling, or the choosing of which thread to run next. The scheduling routine must also be deterministic and uninfluenced by outside factors that may be different on the other replicas. This includes the points at which threads wait to receive input. Such input may arrive at a different point in the execution on all of the machines, and this can cause them to proceed at different speeds or even different paths entirely.

Due to external sources, one processor may have more load than another, and the load balance will probably be different between primary and backup. In a multithreaded environment where the threads access shared data, race conditions may ensue. These race conditions will of course ruin the deterministic behavior that we are striving for. On a uniprocessor, this problem can be solved using deterministic scheduling only. However, on an SMP, multiple threads may run concurrently, so deterministic scheduling is not enough. We deal with this by assuming that the program is “safe” in the sense that all accesses to shared data are synchronized. If so, the access order to all shared variables is governed by the order in which locks were obtained and released. We will forcefully duplicate this order on the backup VMs by allowing a thread to lock an object only in the order that occurred on the primary.

Finally there are the environment-specific attributes that are unique to each machine. The system clock, which measures time in milliseconds, is not assumed to be synchronized between all the systems, and this can lead to different behavior when the application relies on the clock, for example when choosing a random number. Other attributes may be local environment variables in Unix or the system variables in Windows, and identity attributes such as the host name or the machine's IP address.

3.3. Design of FT-JVM

Following are the changes we made to JikesRVM to support fault-tolerant execution.

First, we disabled the default configuration of preemptive switching that is led by clock timeouts, choosing the deterministic one where a yield is forced after encountering 1000 yield points. Yields that originate from within the application, occurring before the counter reaches its limit, are also honored, since they occur at the same place in both primary and backup.

Next, the scheduler's algorithm was altered. JikesRVM's original algorithm for scheduling was already deterministic: if available, schedule a ready I/O thread, else schedule a regular ready thread, etc. However, threads that wait for I/O may become ready at different points in their execution on different replicas, depending on when the I/O is available to the particular system. We decided to check I/O using `select` at certain predetermined intervals so that, when I/O has completed, a blocked thread will be reintroduced to the ready queues in the same position in both primary and backup.

To accomplish the above, we utilize the concept of *frames* [11], but adapt it to virtual machine-level multi-threaded SMP-based executions. A frame is an execution of several consecutive context switches; a frame ends either after a predefined number of context switches occur, or when there are no ready threads (i.e., all threads are waiting for I/O, locks, etc.).[3] Synchronous coordination between the replicas is made at a frame's preamble, so as to allow a consistent, deterministic execution throughout the frame. At this point it is verified that all replicas have completed the previous frame and that it is safe to proceed. When a thread tries to perform an I/O operation, it is blocked and is placed in a special I/O waiting queue. At the start of a frame, the primary replica, which is responsible for actual execution of the I/O, performs a local `select` on all file descriptors corresponding to threads in the I/O waiting queue. For each file descriptor that has completed its I/O task, a message is created that contains the file descriptor's ID as well as the content of that operation (in the case of input it is the data and a status code; in the case of output it is only the status code) and forwards this message to the backup replica. On the backup, instead of calling `select`, the incoming message buffer is consulted and the threads that are listed there are scheduled to run.

Similarly, when running on an SMP, during the execution of a frame on the primary, whenever a lock is acquired, we log the lock ID and the ID of the locking thread, and send this information to the backup. On the backup, our mechanism ensures that threads will obtain locks in the same order as in the primary. We emphasize that this is only done on SMPs and not on uniprocessors, in which deterministic scheduling is enough to ensure consistency.

The above description does not solve the problem of the timing in which these threads are reintroduced to the list of runnable threads. That is, a thread should not be allowed to run on the backup until all the non-deterministic information that it will depend on while executing in this frame has arrived there. Figure 2 illustrates our synchronization protocol for this situation. We force a one frame lag between primary and backup. In particular, we do not allow a new frame to begin until the previous one is complete. This guarantees that the relevant data will arrive at the backup before the beginning of the next frame. We signal to the backup that all data has been sent by sending an "end of frame" message.

Because the primary runs ahead of the backup, it may happen that data arrive at the primary but the primary fails before successfully sending this data to the backup. This exact same data may no longer be available, in which case the backup might end up reading different data, e.g., if the input file was modified in the meantime, or the read is from an external source beyond our control. We mitigate this risk by having the primary immediately send the data upon receiving it from the operating system. Furthermore, the size of a frame can be set upon startup in order to force more frequent synchronizations between primary and backup.

Another alteration to the JikesRVM functionality was in fetching environment-specific data such as the current time or the host name. We identified all such system calls made by the JVM and have the primary pass it to the backup in an asynchronous manner. The backup does not make the system calls and instead consults its queue of messages from the primary.

The *replication engine* is an independent component that we added to the JVM, given the task of coordinating messages between primary and backup. The replication engine is a VM thread and consists of two interfaces: a network interface to a peer JVM and an internal interface to collect or return the non-deterministic data to the other modules. Figure 2 illustrates the protocol that is used for synchronization between the primary and the backup. All data to be sent to the backup is transferred to the replication engine, which in turn packages it and sends it asynchronously. The messages

[3]This is an additional advantage of working at a level that allows recognizing idle state. In [11], frames always execute a fixed number of machine instructions, even if all these instructions are `nop` caused by threads waiting for I/O.

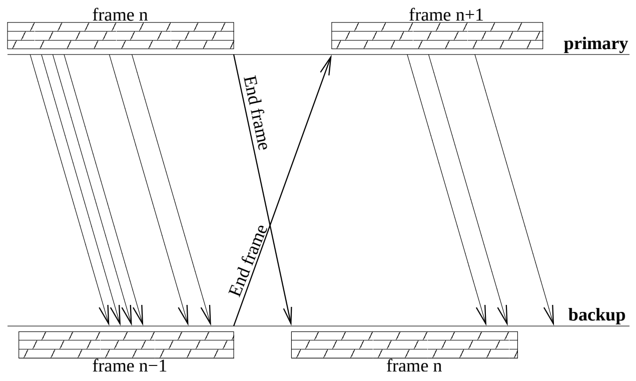

Figure 2. End of frame synchronization protocol. Messages containing I/O information are generated at the beginning of a frame, yet messages regarding lock acquisitions and environment-specific data are sent throughout the execution.

are sent over TCP and thus are guaranteed to arrive in the order in which they were sent.[4] When the end of a frame is reached, a message is sent from the primary to the backup. When the backup reaches its own end of frame, it sends a similar message to the primary. The backup proceeds to the next frame after receiving the primary's message; the primary waits until receiving the end-of-frame message from the backup.

The replication engine on the backup is also responsible for detecting the failure of the primary or backup. This is done by monitoring the network connection shared between the primary and the backup and by exchanging heartbeats. When the connection fails or the heartbeat timer expires, then a general failure of the primary is assumed, and the backup switches to recovery mode. In this mode, the current frame is executed to completion, as all the necessary information from the primary has already been received.[5] If the backup fails, the primary simply continues its execution without synchronizing with the backup.

We maintain that all the changes described above fulfill the necessary conditions for replica determinism, subject to the system model assumptions described in Section 2. To illustrate this point, we present an informal step-by-step description of the execution of an application.

An application is said to begin when the first thread is scheduled to run. If non-deterministic events do not occur, then the application thread proceeds in the same manner on both primary and backup. Yields are taken at the same point in the execution, either explicitly by a call from within the application or implicitly by way of the quasi-preemptive nature of the scheduler. If new threads are spawned by the main thread, then their creation will occur at the same point in the execution on all replicas, and the new thread will be added to the end of the queue, to be later scheduled at the same point on all replicas.

If a thread must wait for input that may arrive at an unknown future time, then it is moved to a special queue to be checked upon frame starts, which occur at predefined points in the execution. If, upon checking, a thread receives the input it was waiting for, then it is scheduled to run by the primary. Notice of this is sent to the backup so that the thread's respective replicas are also started at this point.

Any external data collected by a thread on the primary, whether by performing an I/O operation or by fetching information from the system environment, is sent to the backup. The backup simulates the operation and instead returns whatever data was found by the primary.

When the JVM is run on an SMP, newly-created threads are moved to either of the virtual processors in a round-robin (deterministic) fashion. In cases where two threads contend for a shared lock, then the order in which the lock was obtained on the primary is recorded and relayed to the backup. When a backup thread repeats this execution, it will be allowed to acquire a lock only if the log indicates that this thread is next in line to acquire the lock. If not, then a short delay is enforced, after which the lock acquire is retried.

3.4. Implementation Hurdles

We faced many difficulties in maintaining determinism because of internal JVM operations that are beyond our control. These operations are invoked by the application threads and are run synchronously by those threads, so their run cost is part of the allotment of the thread during its time slice. Some of these internal operations are non-deterministic in their nature, or acquire global system locks and we do not control the access to these global locks. Both of these behaviors affected the application thread's state and resulted in inconsistent runs.

For example, the JIT compiler uses a global hash table that is locked during any compilation. Since the primary and backup machines need not be identical, JIT compilation may occur at different times on them. This necessitated a method by which the thread switch counter of an application thread would be saved prior to such operations and subsequently restored after the operation completed. Another example is the different behavior of the primary and backup when processing non-deterministic data. The primary must take actions to send this data to the backup in addition to performing the operation, while the backup simply needs to look up the corresponding message in its packet queues. Re-

[4] For a larger number of nodes, it may be better to use a group communication toolkit [9].

[5] In the case of multiple backups, a simple leader election protocol will determine the new primary. All backups will connect to the new primary, which will then begin the next frame.

call that the scheduler bases its scheduling decisions on the number of operations performed by each thread. Since here the same thread does different tasks on the primary and the backup, then unless care is taken, the scheduler might make different preemption decisions. The same technique of saving counters, as well as disabling preemption, was used to correct this behavior.

The disabling of preemption had an oft-occurring side-effect of deadlocks with threads that acquired a global lock to internal VM code and now could not release the lock. Besides access to the JIT, another example is access to standard output. This led to judicious checks in the cases where preemption was to be disabled, to make sure that the operation would complete.

3.4.1. Garbage Collection-related problems An additional area that required special attention is the garbage collector. GC is triggered when an allocation request cannot be fulfilled, and this may occur at different points for primary and backup because of their different execution nature. The existing behavior was to force a yield of the thread that requested the allocation. Since the yield might happen only on the primary and not on the backup, or vice versa, then unless care is taken, different scheduling decisions could occur in both of them. To solve the problem, the yielding thread would be moved to the head of the queue, to be rescheduled immediately after the GC thread completes, and have its original state restored so as to hide the unanticipated yield.

The above solution did not work smoothly when FT-JVM was executed on an SMP. This is because it relies on the assumption that the yielding thread would run only after the GC thread was completed, since the GC was supposed to be non-preemptive. However, when multiple processors are involved, each processor maintains a GC thread and those threads work in conjunction. Therefore, the GC thread on one processor would yield until all its peer threads were active. This situation caused the application threads to be scheduled before their time, causing further memory requests that led to unnecessary reinvocations of garbage collection. This situation drastically reduced performance. Our fix was to disable all access to queues during all phases of garbage collection, except for the queue that keeps only the idle thread. We made sure that the idle thread did not request any memory allocations so that it would not interfere in the collection of garbage. Only after the GC threads completed on all processors would the thread queues be reactivated and the scheduling resume as before.

4. Performance Analysis

We tested the modified JVM on dual-processor 1.7GHz Intel Xeon machines, with 1GB of RDRAM, running Red Hat Linux 7.2 (kernel version 2.4.18), connected via a 100Mbps Ethernet network. The results described herein were obtained by running the benchmark programs multiple times while varying the "number of context switches per frame" parameter of the FT-JVM. This variable dictates how often the primary synchronizes with the backup, waiting until the previous frame's completion on the backup before proceeding to the next frame. When this parameter is set to a low value it forces the synchronization to occur more frequently, thereby adding communication overhead. On the other hand, when this variable is assigned a high value, synchronizations are few and far between, but there is greater data loss when the primary fails. In this section we present results that show that even a frequent synchronization, for example every second, can achieve results that are competitive with non-replicated executions.

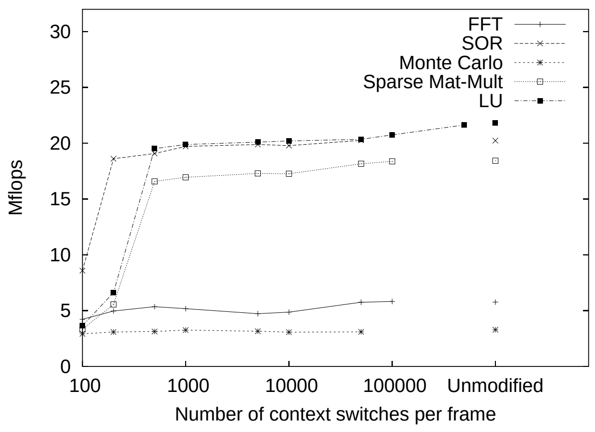

Figure 3. SciMark Java benchmarks

Figure 3 shows the performance of the system while running the SciMark 2.0 benchmark suite [2]. This benchmark performs a low-I/O, compute-intensive exercising of the JVM, such as Fast Fourier Transform, π approximation, and sparse matrix multiplication. The replicated version of the JVM was able to equal the performance of the non-replicated one when the frame size was large enough. For a frame of size 10000 context switches, which takes about 1 second to complete, we achieved very close results to the unmodified JVM.

In contrast to the CPU-intensive SciMark benchmark, Figure 4 shows the _201_compress benchmark from the SPECjvm98 suite [3], which performs LZW compression on a large file, an I/O-bound operation. Here the amount of data that is passed from primary to backup becomes evident, as short frames force the waiting for all the data to arrive before proceeding to the next frame. The performance loss is less than a factor of 2, even with very short frames. However, another I/O-bound benchmark in the suite (_209_db) does not fare so well. Figure 5 shows its results. This I/O-intensive benchmark differs from the former in the large

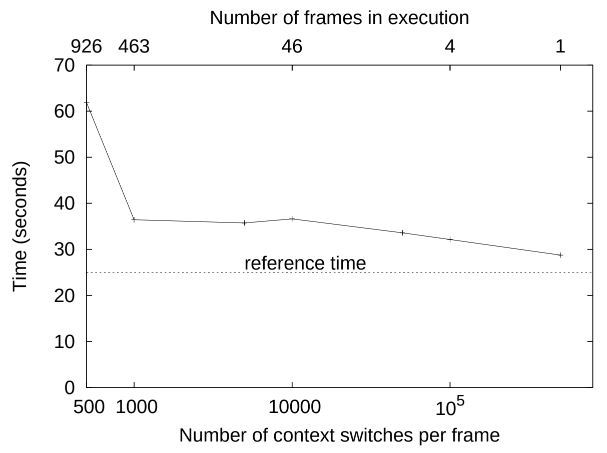

Figure 4. LZW compression (I/O-bound) benchmark

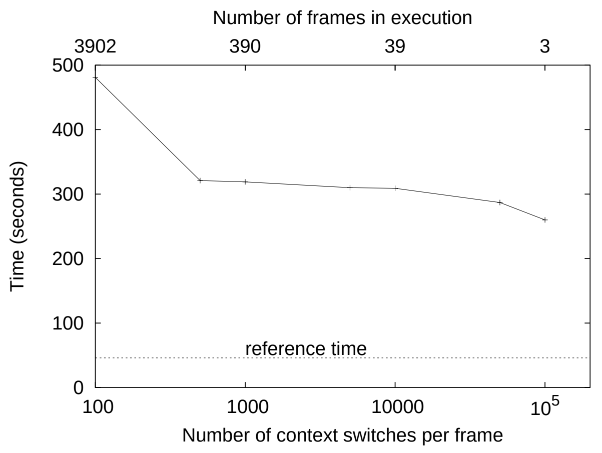

Figure 5. Address book (very I/O intensive) benchmark

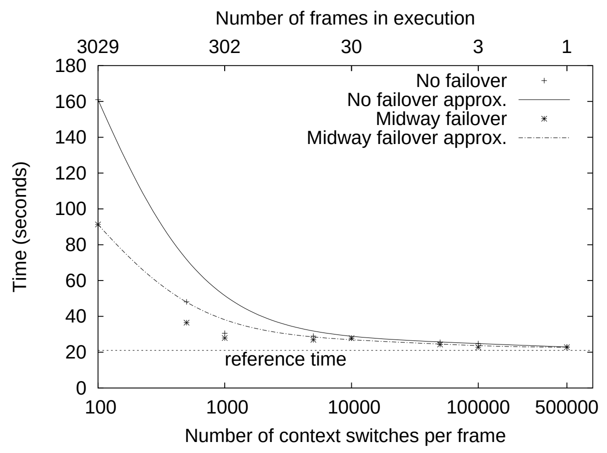

Figure 6. Ray-tracing benchmark with a forced failover of the primary in mid-execution

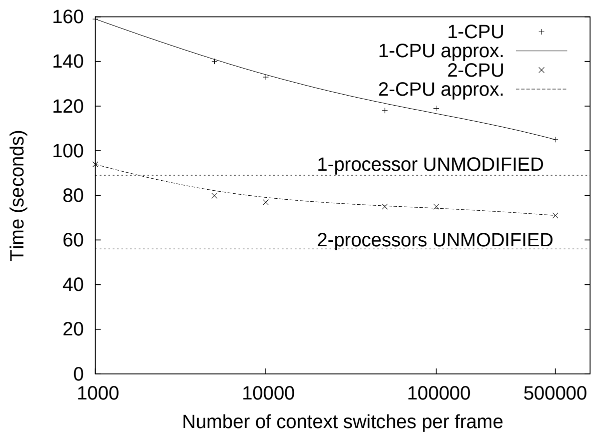

Figure 7. Multithreaded ray-tracing benchmark, two threads on SMP

number of times that it accesses the disk: about 95000 packets signifying such events are sent by the primary to the backup.

We profiled the performance of the FT-JVM using a ray-tracing application, which is also part of the SPECjvm98's benchmarks suite. This benchmark (_205_raytrace) yielded similar results to the SciMark benchmark, as can be seen in Figure 6. Here we also show the results obtained by manually aborting the primary midway through the execution. The only backup replica continued the execution as primary and had no new backup with which it had to communicate. Subsequently, its execution took less time since the second half of the execution proceeded in a manner similar to an unreplicated execution. Regardless of the value of the "context switches per frame" parameter, the primary's failure was detected by the backup and recovery was begun within less than one second.

Figure 7 illustrates the performance of a ray-tracing benchmark test on two processors. This raytracer (_227_mtrt) utilizes two threads so as to maximize concurrency. As can be seen, running FT-JVM on both processors yields better results than single-processor runs, despite the possibility for contention and the overhead of marshaling lock information to the backup.

In order to gain insight into what portions of replication cause more overhead, we profiled three of the benchmarks. These are ray-tracing, which is compute intensive, LZW compression, which is I/O-bound, and the address book (database), which is very I/O intensive. The results are illustrated in Figure 8. The application run time shows the portion of the time dedicated to executing the target application. The send queue bottleneck measures the time be-

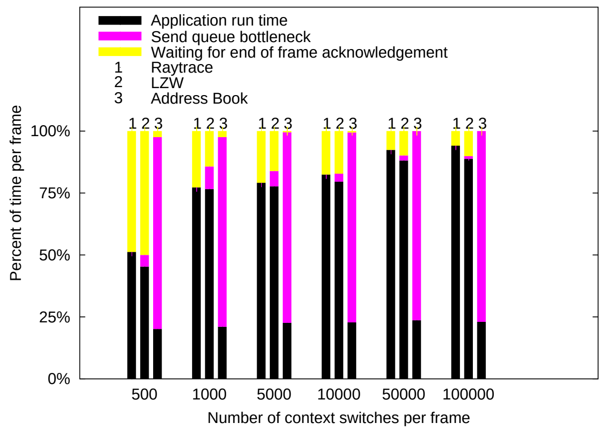

Figure 8. Cost breakdown in replicated execution of various applications

tween an end of frame message being generated and when it is actually sent. This shows the backlog of messages that have yet to be sent to the backup. The third indicator shows the delay between the time at which the end of frame is sent by the primary and the time at which the acknowledgment is received. This indicator shows the lag in the backup: if it is very small then the backup is the one that completed the previous frame before the primary.

It is evident that the type of application affects the performance of replication. Specifically, an application that generates many I/O operations burdens the replication engine with many messages that lead to a larger delay in sending them to the backup. On the other hand, in CPU-bound applications, e.g., ray-tracing, the time spent in the primary is mainly the waiting to receive an acknowledgment to the end of frame message that was sent to the backup. This cost becomes less significant when the number of context switches per frame grows.

5. Limitations

As noted in Section 2, we make several assumptions about the system model and the structure of the applications to be replicated, which enable us to guarantee the consistency of execution and the correct recovery upon a failure. However, given the nature of applications in use today, some of our assumptions can be seen as too rigid. For example, in order to support consistent runs on an SMP, we assume that all shared data is protected by monitors and that the application is free of data races. In the cases where this is not so, our model fails to enforce a consistent state in the replicas, which may potentially lead to inconsistencies upon a failure.

Another source of non-determinism can come from native methods that are called by the Java application and are executed outside the scope of the JVM. We have no control over the progress of such executions, nor can we guarantee that actions performed on the primary will execute the same way on the backup. This scenario is addressed and partially solved in [27].

Our implementation does not currently replicate open sockets and does not transparently recreate network connections in the new primary following a failure. This limitation, however, can be overcome by sending more control information to the backup, such as TCP frame information. The means to accomplish this have been studied in [14, 16, 33].

Files that were opened by the primary before a failure must also be re-opened by the new primary. FT-JVM does not currently support this, but the implementation would be straight-forward: information on all open files, whether opened physically by the primary or virtually by the backup, will be recorded by the VM. The basic elements of the file descriptor, such as the seek location, would be updated upon every state change of the file, such as reading, writing, or seeking. Then, upon recovery, all previously-opened files would be reopened and the seek position would be restored. However, this technique will not work in cases where the writes to a file are different between primary and backup due to environmental differences. An example is if an application dictates that a string should be written to a file at a location that is dependent on the computer's name (perhaps in a text file that sorts the data alphabetically). If the primary with a host name A writes to the file at its head and then fails before the end of the frame, then the new backup (with host name Z) will repeat the write but will instead add the string at the tail of the file. This results in inconsistent behavior.

6. Related Work

The common methods for implementing fault tolerance by replication are the *replicated state machine approach* [29] and the *primary–backup approach* [12]. Both refer to the replication targets as providers of *services* that are invoked by client requests. The former model has the client requests sent to all replicas for processing, while the latter has clients interfacing only with the primary, and only state changes are sent to the backups. Active replication provides near instant recovery, while primary–backup is more resource efficient.

When state changes are complex and cannot be sent to the replicas by a primary server in a digest form, the active replication approach is more appropriate. In this case, all replicas individually process incoming requests. Such implementations focus on the need for deterministic execution, so that all executions will be alike. This is achieved either by transactional means [20] or by forcing a sequential order of requests [28]. However, these examples reduce the effectiveness of multithreaded designs and do not deal with executions on an SMP.

Another related field is the deterministic replay of executions [13, 22], where an execution is recorded in a manner similar to checkpoint/restart but with an emphasis on preserving the same ordering of execution when later replayed. This technique is typically used for debugging or visualization but can be used in the context of fault tolerance to execute on a secondary machine after the failover of the primary server.

Our work focuses on implementing replication at the virtual machine level, i.e., below the application but above the operating system. In the past, replication was typically built into the hardware [7, 32] as well as in the operating system [25] and middleware [5, 9, 21, 23].

The basis for our work is the hypervisor approach, detailed in [11]. Of fundamental difference is the fact that we significantly reduce communication overhead by only sending information relevant to the particular application that is being replicated, rather than for all processes on the computer. Furthermore we filter and send only the events that are required for maintaining the determinism of the replicas, rather than reporting on all interrupts or asynchronous events. For example, we concentrate on events that pertain to the application threads and ignore such events that involve threads that are internal to the virtual machine. Finally, we can end frames early if all threads are waiting for I/O, whereas in [11] frames always execute the same number of instructions.

The Voltan application environment creates fail-silent processes by replicating processes [10]. Voltan eliminates all sources of non-determinism and compares the results of computation on both replicas, shutting down a replicated process if the two replicas disagree. Voltan wraps all non-deterministic system calls so they are only being executed on one replica and their results are pushed to the other, similar to what we do. Voltan also provides hooks for the application to register handlers for application based non-deterministic functions, and wraps the communication with the outside world as well. The main differences between Voltan and our work is that we implement replication at the virtual machine level, and therefore can be more transparent to (pure Java) applications. We also handle thread scheduling of multithreaded applications, including on SMPs. On the other hand, Voltan includes checks for shutting down replicas whose state diverges, helping overcome Byzantine failures.

Another approach to implementing fault tolerance within a JVM was developed independently of our effort and reported in [27]. That work explores ways of controlling non-deterministic events, either by recording the order of lock acquires or the sequence of thread scheduling decisions in a multi-threaded environment. They also provide hooks that help control non-determinism that may arise out of calls to native methods, in which the JVM cannot control or directly observe the methods' actions and side-effects. Experimental results based on modifying Sun's Java Development Kit (JDK) 1.2 on Solaris were also reported in [27].

Our approach differs from the above in several ways. First, our JVM uses a deterministic scheduler, which eliminates the overhead of keeping track of thread scheduling decisions. Our approach can work with the JIT compiler without modifying it, whereas the work of [27] cannot. Consequently, all our results where obtained with JIT, while in [27] the results are for bytecode executions only. We specifically addressed issues related to running the JVM on an SMP, which entails a higher overhead of synchronization but a greater utilization of computational resources. The execution time overhead imposed by our technique (with frames of at least one second) is also smaller than [27]. This could result from tighter synchronization between the primary and backup in [27], and the fact that we obtain consistent scheduling without logging scheduling decisions (see Section 3). Finally, in our implementation the primary and backup simultaneously execute the program, whereas the above-mentioned approach uses a "cold" backup, in which recovery information is simply logged. In the case of a failure, our system recovers faster because in the latter approach the backup commences execution from the beginning of the program or from a recent checkpoint. On the other hand, the recovery process in [27] is sometime more accurate than ours, in the sense that if all output functions are *testable* or *idempotent*, they ensure correct *exactly-once* semantics. Our work can only guarantee this for idempotent output functions (but in our case, only the functions called within the last time frame in which the primary fails, which is typically less than a second long, are vulnerable to this).

7. Conclusions

In this paper, we report on implementing fault tolerance at the virtual machine level of Java by applying similar techniques from the hypervisor-based approach to the JikesRVM. We describe the design of the system, the alternatives we considered, and the performance obtained. The main challenge was how to solve the resulting non-determinism while maintaining good performance and processor utilization.

References

[1] Jikes Research Virtual Machine. http://www.ibm.com/developerworks/oss/jikesrvm/.
[2] Scimark 2.0. http://math.nist.gov/scimark2/.
[3] SPEC JVM98. http://www.specbench.org/jvm98/.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[5] Y. Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1995.

[6] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. Technical Report UBLCS–94–15, Department of Computer Science, University of Bologna, June 1994. Revised January 1995.

[7] J. Bartlett, J. Gray, and B. Horst. Fault Tolerance in Tandem Computer Systems. In A. Avižienis, H. Kopetz, and J. Laprie, editors, *The Evolution of Fault-Tolerant Computing*, volume 1 of *Dependable Computing and Fault-Tolerant Systems*, pages 55–76. Springer-Verlag, 1987.

[8] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.

[9] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, December 1996.

[10] D. Black, C. Low, and S. Shrivastava. The Voltan Application Programming Environment for Fail-Silent Processes. *Distributed Systems Engineering*, 5(2):66–77, June 1998.

[11] T. Bressoud and F. Schneider. Hypervisor-base Fault Tolerance. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[12] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison-Wesley, 2nd edition, 1993.

[13] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pages 48–59, 1998.

[14] O. P. Damani, P. Y. Chung, Y. Huang, C. Kintala, and Y. M. Wang. One-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proc. of the 6th World Wide Web Conference*, April 1997.

[15] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. Technical report, Computer Science Department, University of Newcastle, Newcastle upon Tyne, United Kingdom, August 1994.

[16] R. Friedman, K. Birman, S. Keshav, and W. Vogels. Reliable time delay-constrained cluster computing. United States Patent 6,393,581, May 2002.

[17] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. In *Proc. of the 15th Symposium on Reliable Distributed Systems*, pages 140–149, October 1996.

[18] R. Friedman and A. Vaysburd. Fast Replicated State Machines Over Partitionable Networks. In *Proc. of the 16th Symposium on Reliable Distributed Systems*, October 1997.

[19] M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.

[20] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proceedings of IEEE Symposium on Reliable Distributed Systems, SRDS'00*, pages 164–173, October 2000.

[21] I. Keidar and D. Dolev. Totally Ordered Broadcast in the Face of Network Partitions. Exploiting Group Communication for Replication in Partitionable Networks. In D. Avresky, editor, *Dependable Network Computing*, chapter 3, pages 51–75. Kluwer Academic Publications, 2000.

[22] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic Replay of Distributed Java Applications. In *Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium*, pages 21–228, 2000.

[23] L. Lamport. The Part-Time Parliament. *IEEE Transactions on Computer Systems*, 16(2):133–169, May 1998.

[24] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[25] D. Major, G. Minshall, and K. Powell. An Overview of the NetWare Operating System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 355–372, 1994.

[26] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A Toolkit for Building Fault-Tolerant Distributed Application in Large Scale. Technical report, Department d'Informatique, Ecole Polytechnique Federale de Lausanne, July 1995.

[27] J. Napper, L. Alvisi, and H. Vin. A Fault-Tolerant Java Virtual Machine. In *International Conference on Dependable Systems and Networks*, pages 425–434, June 2003.

[28] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Proceedings of the IEEE Symposium for Reliable Distributed Systems*, pages 263–273, October 1999.

[29] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[30] R. van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[31] P. Veríssimo, L. Rodrigues, and J. Rufino. Delta 4 - A Generic Architecture for Dependable Distributed Computing. In D. Powell, editor, *ESPRIT Research Reports*. Springer Verlag, November 1991.

[32] S. Webber and J. Beirne. The Stratus Architecture. In *21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 79–85, June 1991.

[33] V. C. Zandy, B. P. Miller, and M. Livny. Process Hijacking. In *Proceedings of the Eighth International Symposium on High Performance Distributed Computing (HPDC'99)*, pages 177–184, 1999.