Transparent Fault-Tolerant Java Virtual Machine

Roy Friedman* Alon Kama
Computer Science Department
Technion – Israel Institute of Technology
Haifa, 32000
Israel

Phone: +972-4-8294264
FAX: +972-4-8221128
Email: {roy,alon}@cs.technion.ac.il

December 3, 2002

Abstract

Replication is one of the prominent approaches for obtaining fault tolerance. In a distributed environment, where computers are connected by a network, replication can be implemented by having multiple copies of a program run concurrently. In cases where a copy on one of the computers crashes, the others may proceed normally and mask that failure.

Implementing replication as described above on commodity hardware and in a transparent fashion, *i.e.* without changing the programming model, has many challenges. Deciding at what level (hardware, operating system, middleware, or application) 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, such as I/O and environment differences (*e.g.* different clocks). Also, the minimization of overhead needs to be addressed, so that the performance is acceptable.

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).

Keywords: Fault Tolerance, Replication, Java, Virtual Machine

*Contact author.

1 Introduction

Active Replication is one of the prominent approaches for obtaining fault tolerance [10, 31]. Specifically, a replicated system involves running multiple copies of the same program concurrently such that if one of the replicas crashes, the others can mask the failure.

The main complication 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.

In this paper, we report on implementing fault tolerance at the virtual machine level of Java by deterministically replicating the execution on independently-failing processors, and provide a detailed performance analysis of our implementation. We show that despite the restrictions on thread and scheduler behavior, we achieve favorable execution results. The key challenges to be described herein are in guaranteeing deterministic execution and interfacing with an existing and complex virtual machine.

1.1 Fault Tolerance

As defined by [25], a *fault* is a defect occuring in a hardware or software component. This fault may cause certain algorithmic assumptions to no longer hold and will lead to an *error*, which is an incorrect behavior that is caused by the presence of a fault. The error, in turn, may lead to *failure*, which is defined as the inability of a system to perform its specified task or service.

The general aim is to avoid such failures when possible. A qualitative term to describe the ability of a system to perform properly is *dependability*. Dependability is partly based on the quantitative measures of *reliability* and *maintainability*. Reliability, denoted $r(t)$, is the probability that the system performs without interruption over the entire interval $[0,t]$, where

t represents time [22]. Reliability is often represented using "nines-notation", *e.g.* five-nines means that the system is reliable 99.999% of the time. If t is taken to be one year, than a system represented by five-nines promises to be reliable for all but five minutes out of that year.

Maintainability, denoted m(t), is the probability that the system can be restored after a failure within time t [22]. Maintainability requires that the state to which the system is restored will be just after the last state prior to the fault.

There are several ways to increase reliability. One such method is *fault avoidance* [25], which takes the view that it is possible to build fault-free software. This method focuses on the specification and verification of software, *i.e.* concentrates on the design phase. Another method is *fault removal*, which applies techniques to remove faults from existing software. In this method faults are analyzed ("debugged") and fixes are applied to the program so that at a future execution of the code, the failure will not re-occur.

The above methods can improve the reliability of a system but do not help to increase maintainability. In other words, these measures concentrate on removing faults but do not aid in recovering quickly after a fault occurs. Furthermore, they do not provide solutions to physical failures during runtime. There is no way to guarantee faultless executions, and thus there needs to be a way to deal with them during runtime. *Fault tolerance* is such a method. It focuses on preventing faults from causing failures. For a system to be fault-tolerant, it must be able to detect, diagnose, confine, mask, compensate and recover from faults [18].

Fault tolerance is important because it allows the execution of long-lived computations that exceed the system's *Mean Time Between Failures* (MTBF) measure (see [30] and references therein). In such a case, there is a high probability that the computation will not finish successfully, unless the system can recover from failures in a swift manner. Fault tolerance is also useful for preserving the continuity of a system, as well as in cases where a failure is prohibitively expensive, for example in applications such as air traffic control.

1.2 Implementation of Fault Tolerance

One of the primary methods of achieving fault tolerance in programs is by utilizing the concept of *Checkpoint/Restart* [16]. C/R is composed of two parts: the *checkpoint*, which is the periodic saving of the state of the program, and the *restart*, which is the recovery from a failure by continuing execution from the last saved checkpoint.

Using C/R has several advantages. First, it is a simple concept, and thus lends itself to being implemented at various levels, such as within the operating system or inside the application itself. There are implementations of C/R that are statically linked to a program as a partially transparent library [29]. Furthermore, checkpoint data may be written to a file on a shared network drive, thus providing for an easy method of recovery, potentially on a second machine of a similar configuration.

One of the main disadvantages in C/R concerns the trade-off between the frequency of checkpoints and the ability for speedy recovery upon restart. In order to maintain low runtime overhead, checkpoints should be taken infrequently so as to avoid work stoppage in what may be a prolonged snapshot of the system state. However, this may cause a restart to potentially last much longer, since the last checkpoint may now be much further apart from the point at which the fault occurred. Also, C/R by itself does not ensure that the reexecuted computation between the last checkpoint and the point of failure will be the same as before the failure.

Another method for providing fault tolerance is *active replication* [5, 13, 30]. Active replication is the term used to describe the coordination of a set of replicas on processors that fail independently of each other. If care is taken to ensure that the replicas keep the same state, then such a system can achieve near-negligible recovery times after a failure of one of the replicas (known as *Mean Time to Repair*), since the computation continues uninterrupted on the rest of the machines. Furthermore, a replication-based system can guard itself against a certain number of failures by maintaining enough replicas to hide such faults. A system is

called *t-fault-tolerant* when it can withstand $t-1$ independent faults and not fail [30].

Active replication as a method for fault tolerance also has its disadvantages. One is the increased cost of hardware for the duplication of effort. The redundancy of hardware and computation cycles presents a monetary as well as a maintenance cost. Another drawback is the overhead and complexity of maintaining consistency among the replicas. This includes the communication that must take place to ensure coordination, as well as the fact that the pace is dictated by the slowest replica.

However, active replication has one big advantage over C/R: if combined with *transparency* — the ability to mask failures by relying on replicas and maintaining the illusion of a single copy of the execution — then such a system is highly available for external requests. This is due to the fact that recovery costs from faults are minimal and can be invisible to a user, if the transition to $t-1$ replicas from the original t can be hidden.

Yet another methodology for replication is *primary–backup* [13], in which the processing is concentrated at one of the servers, referred to as the *primary*. Client requests are only handled by the primary, which processes the requests and sends the updated state to the rest of the replicas, which are referred to as *backups*. In essence, the backups only apply the state change that is received from the primary, thus reducing the redundancy of computation and potentially decreasing the processing time of the backups (under the assumption that a state change ensues the processing of a client request, and that it can be applied at a cost which is less than performing the original computation).

An important question to consider is at what level to implement replication. The general consensus dictates that implementing at lower levels (*e.g.* hardware or operating system) may increase speed, but at a formidable development cost (especially for hardware) and the lack of portability options. Implementing it at the application level may place an undue burden on the application's developer, because replication may involve complex coordination and this may not be correctly implemented. Furthermore, if replication is implemented at a lower level than

the application, then legacy programs that are not fault-tolerant may be run at this superior dependability level, without any programmatic changes.

This leads us to consider the possibility of not implementing active replication within a certain tier, but rather building it between existing levels. This concept is called the hypervisor approach. A *hypervisor* is a software layer that implements virtual machines. Such a virtual machine executes the same instruction-set as the level on which it resides. That is, the virtual machine acts as an interceptor between the two layers and communicates with them using the existing interface.

The hypervisor presents an ideal platform upon which to implement active replication. By its nature it is transparent to the layers above and below it, as well as to the user. It is more cost-effective than changing an existing and possibly complex layer. And if implemented at a high enough level, it can be highly portable.

Bressoud and Schneider [12] implemented a hypervisor to perform replication, and built it between the operating system and the hardware. Their method for achieving fault tolerance was by using a *semi-active replication* approach called *leader/follower* [7, 8, 22]. One processor is designated the *primary* and the others are designated *backups*. The user interacts only with the primary, *i.e.* sends input to it and receives output from it. The primary informs the backups of the input, and the backups' output is suppressed. If the primary fails, one of the backups is chosen to be the new primary and the computation continues.

The above implementation achieved replication at a slowdown factor of 2. The disadvantages of this method are the fact that it is not portable (the implementation in [12] is composed partly of assembly language for the HP 9000) and has the side-effect of replicating all processes on the machine, whereas we typically wish to replicate a specific application.

Our approach to replication borrows from Bressoud and Schneider in that we implement replication using leader/follower in a tier that sits between the static tiers of a computer [12].

Specifically, ours is a commonly-used middle tier that is easily replaceable: the Java Virtual Machine. This platform supports potentially numerous applications, due to the ubiquity of the Java programming language. Furthermore, JVMs have been written for many platforms, and some are highly portable. With this design, we achieve replication with dramatically less overhead than Bressoud and Schneider's implementation. This is because we apply techniques to communicate the minimum information necessary to guarantee deterministic execution only for the required processes.

1.3 Paper Roadmap

The rest of this paper is organized as follows. Section 2 outlines the system model assumptions. Section 3 presents the design of the FT-JVM. In section 4 we analyze the performance of this model. Section 5 describes some of the limitations of our model. Section 6 summarizes the relevant related work. Section 7 concludes this paper.

2 System Model Assumptions

In the design of the FT-JVM, 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 do not behave in a Byzantine manner [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 engine. 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.

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.

The environment variables that are accessed by the JVM are replicated from the primary to the backups. However, in the case where the primary fails and a backup with different values to those variables takes over, we assume that execution will proceed correctly. In other words, we assume that the underlying application will safely handle the change in 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 (the partially shaded modules indicate where changes were made to support fault tolerance; these changes will be discussed later). JikesRVM itself is mostly written in Java. As such, the VM executes its own Java code, much as 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 within the Java code. Both of these modules are written in C.

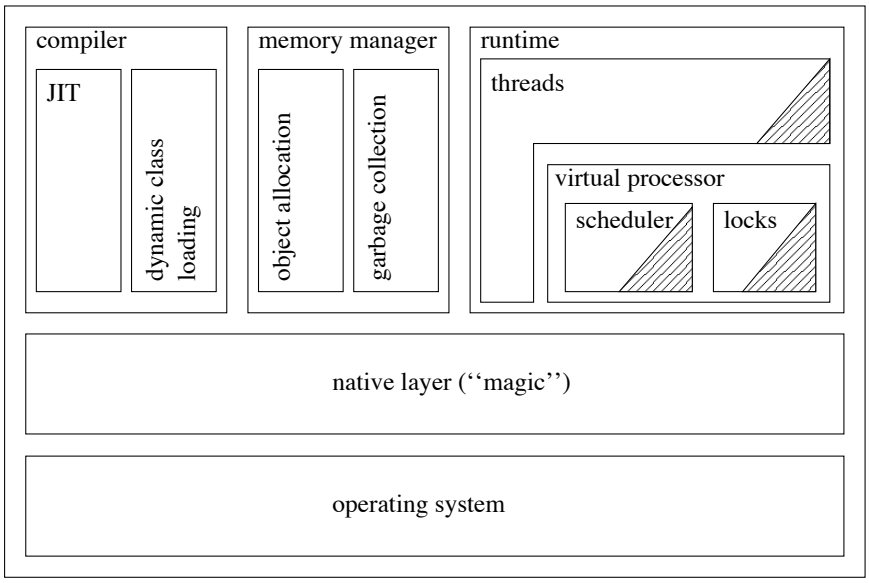

Figure 1: JikesRVM block diagram

The modules relevant to replication lie in the runtime subsystem. We will not discuss many of its components, such as object layout, dynamic type checking, and exception handling, as they are not relevant to our work. Instead we concentrate on 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*.

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 such I/O-dependent thread cannot be scheduled, it turns to any of the ready threads. Otherwise it schedules the idle thread.

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 that owns the lock. If a lock cannot be acquired, the thread requesting that lock will yield.

3.2 Sources of Non-Determinism

Our primary objective is to maintain a consistent execution between the replicas. In case one of the replicas fails, another replica that needs to mask this failure must have the same state as the failed one. This requires control over non-deterministic points in the execution that can cause a branch in progress and leave the replicas in a different state. 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 the least amount of overhead.

The first focus point for non-deterministic behavior is the thread scheduler. If clock-driven context switches are enabled then threads may perform a variable amount of work, depending

upon when the clock was polled. This can result in a difference in execution of one segment, demarcated by two neighboring safe yield points, for a given time slice.

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.

The problem is even more acute on a SMP. Due to external sources, one processor may have more load than another, and the load balance will probably be different between primary and backups. 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. We deal with this by assuming that the program is "safe", that is, all accesses to shared data are synchronized. If so, there is a total ordering in the access to locks by the threads on a SMP. 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 which 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 or 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 modified the preemptive context switching. It is preferable to preempt threads so that waiting threads are not blocked for unreasonable periods of time. Therefore we disabled

the default configuration of preemptive switching that is led by clock timeouts, choosing the deterministic version 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.

Second, 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, where it is certain that the I/O has completed and the thread may proceed in its execution.

In order to accomplish the above, we utilize the concept of *frames* [12]. A frame is a consecutive number of context switches. 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. Then, at the start of the frame, the primary replica, which is responsible for actual execution of the I/O, performs a local `select` on the file descriptors of threads that are currently awaiting the completion of I/O operations. For each of the file descriptors that have completed their 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 backups replicas. On the backups, instead of calling `select`, the incoming message buffer is consulted and the threads that are listed there are scheduled to run.

The above description still 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 has arrived at the backup. Figure 2 illustrates our synchronization protocol for this situation. We force a lag between primary and

backups by having the primary run one frame ahead. 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 backups 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 backups, it may happen that data arrive at the primary and that the primary fails before successfully sending this data to the backups. This data may be unrecoverable and can degrade the transparency of the fault tolerance mechanism. 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 backups.

Another alteration to the main JikesRVM functionality was in fetching environment-specific data such as the current time or the host name. We identified all system calls made by the JVM to retrieve this type of data and have the primary pass it to the backups in an asynchronous manner. The backups do not make the system calls and instead consult their 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 backups. 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 backups. All data to be sent to the backups is transferred to the replication engine, which in turn packages it and sends it asynchronously. The messages are sent over TCP and thus are guaranteed to arrive in the order in which they were sent.[1] When the end of the frame is reached, messages are concurrently sent by the primary to all backups. When the backups reach their own end of

[1]Currently, we only tested the system with two nodes. For a larger number of nodes, it may be better to use a group communication toolkit [10].

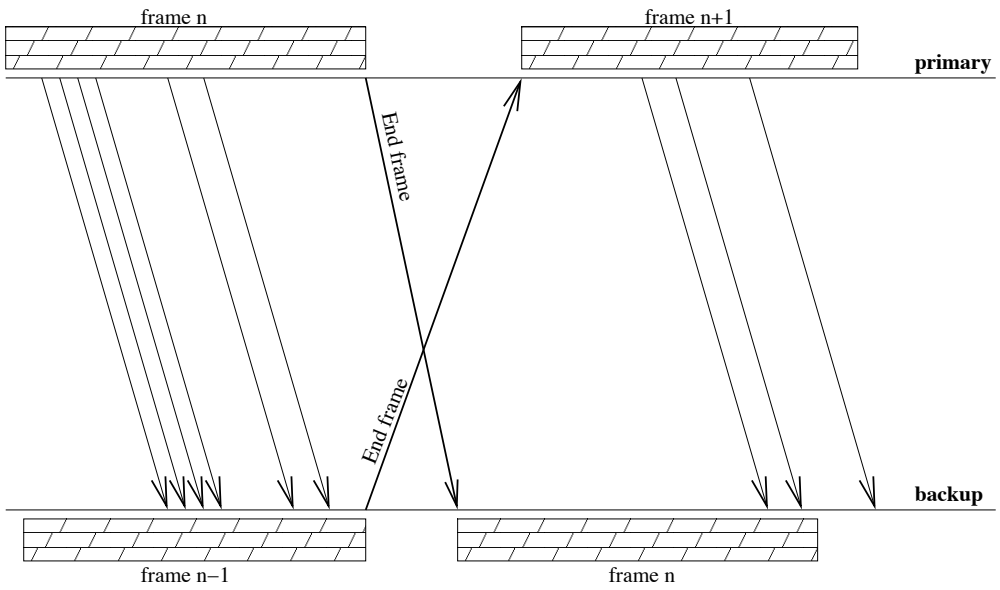

Figure 2: End of frame synchronization protocol

frame, they send a similar message to the primary. The backups proceed to the next frame after receiving the primary's message; the primary waits until receiving all the messages from the backups.

The replication engine on the backups is also responsible for detecting the failure of the primary. This is done by monitoring the network connection shared between the primary and each of the backups 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. 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.

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 backups. 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 FIFO 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 backups 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 backups. The backups simulate the operation and instead return whatever data was found by the primary.

When the JVM is run on two processors, any 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 natural locking order that occurred on the primary is recorded and relayed to the backups. 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 this lock. If not, then a short delay is enforced, after which the lock acquire is retried. By that time the other processor will have scheduled the appropriate thread that is destined to acquire the lock first.

3.4 Implementation Hurdles

In implementing the FT-JVM, we encountered significant obstacles that were overcome only after some effort. First off, we started using JikesRVM while it was still actively developed. For the first several months, until a newer version was released, it took about 25 minutes for a compilation cycle, which slowed down our progress. Also, the platform can only be debugged using a special debugger that lacks many basic features and does not support SMP executions.

We also faced difficulties in maintaining determinism because of internal JVM operations. 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. Unfortunately, some of these internal operations were non-deterministic in their nature, or acquired global system locks that could not be released when the thread was preempted. Both of these behaviors affected the application thread's state and resulted in inconsistent runs.

An example of the above behavior can be found in the JIT compiler, which uses a global hash table that is locked during any compilation. 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 backups when processing non-deterministic data. The primary must take actions to send this data to the backups in addition to performing the operation, while the backups simply need to look up the corresponding message in their packet queues. These disparate operations have a different "cost", which again may affect the thread's state upon preemption. The same technique of saving counters, as well as disabling preemption, was used to correct this behavior.

The disabling of preemption had an oft-occuring 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, so as to make sure that the operation would complete.

3.4.1 Garbage Collection-related problems

Another 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 backups because of their different execution nature. The existing behavior was to force a yield of the thread that requested the allocation. This creates an inconsistent execution. 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 a 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 parallel. 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 idle queue that kept 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 would continue as before.

The garbage collector that was used in our implementation was the mark-sweep collector. It is possible to use other collectors in JikesRVM, for example the concurrent GC that does not require halting any application threads. We have not tested this garbage collector and

therefore cannot report on any performance improvement that it might offer. However, we anticipate that such a collector would require further synchronization between threads on the allocation of memory from a global heap. Such contention may introduce non-deterministic yields such as the ones described previously, and would force us to disable preemptions when accessing certain critical sections.

4 Performance Analysis

We tested the modified JVM on dual-processor 1.7GHz Intel Xeon machines, with 1GB of RDRAM. The machines ran Red Hat Linux 7.2 with kernel version 2.4.18. They were connected via a 100Mbps Ethernet network. The results described herein were achieved by running the benchmark programs multiple times while varying the "number of context switches per frame" parameter of the JIT compiler. 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. Intuitively, 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 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, with such tests as Fast Fourier Transform, Pi approximation, and sparse matrix multiplication. The replicated version of the JVM was able to equal the performance of the non-replicated version when the frame size was widened enough. Note that for these tests, 10000 context switches take about 1 second. For a frame of size 10000 context switches, we achieved very close results to the unmodified JVM.

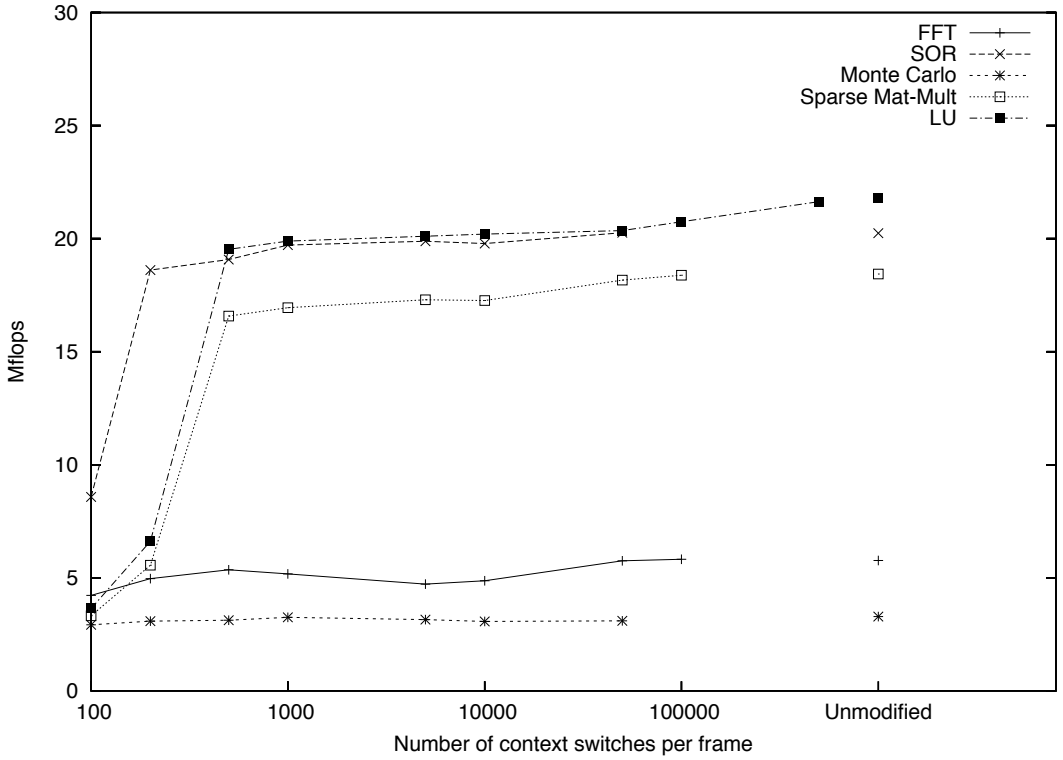

Figure 3: SciMark Java benchmarks

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 backups 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 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 another of SPECjvm98's benchmarks suite. This benchmark (_205_raytrace) yielded similar

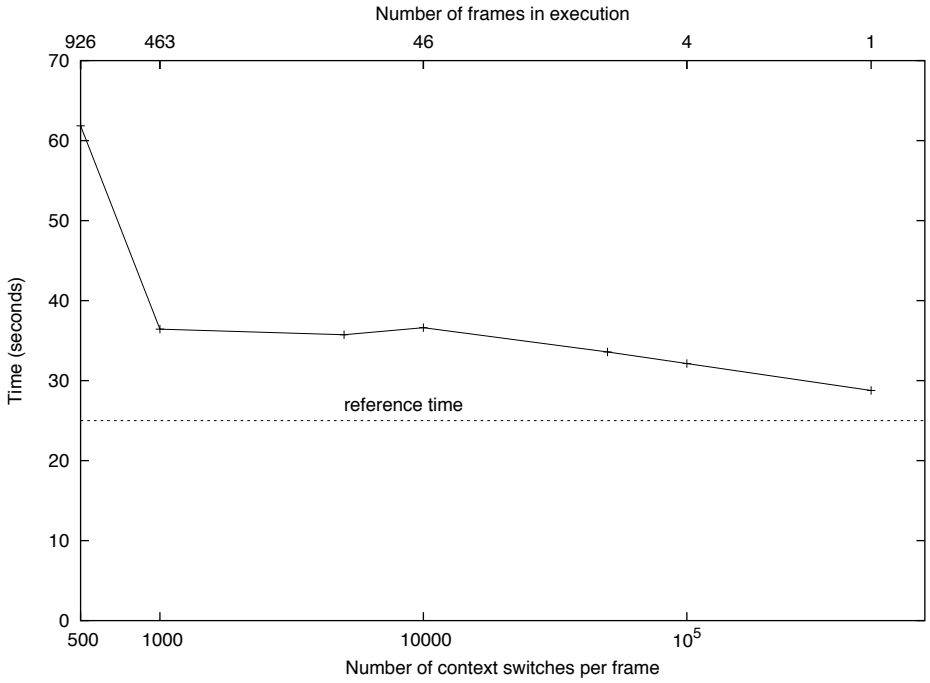

Figure 4: LZW compression (I/O-bound) benchmark

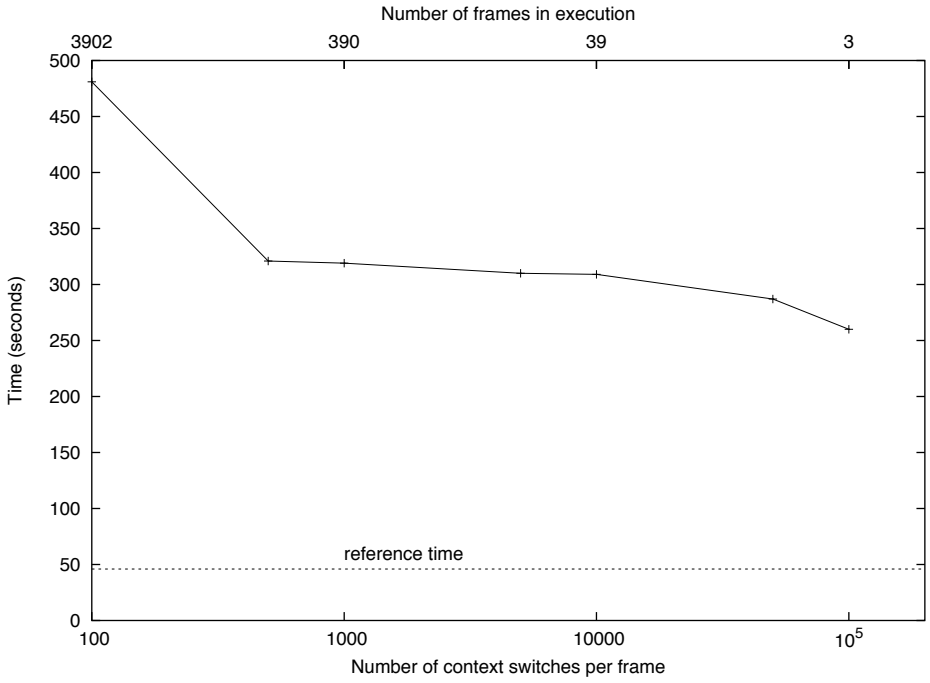

Figure 5: Address book (very I/O intensive) benchmark

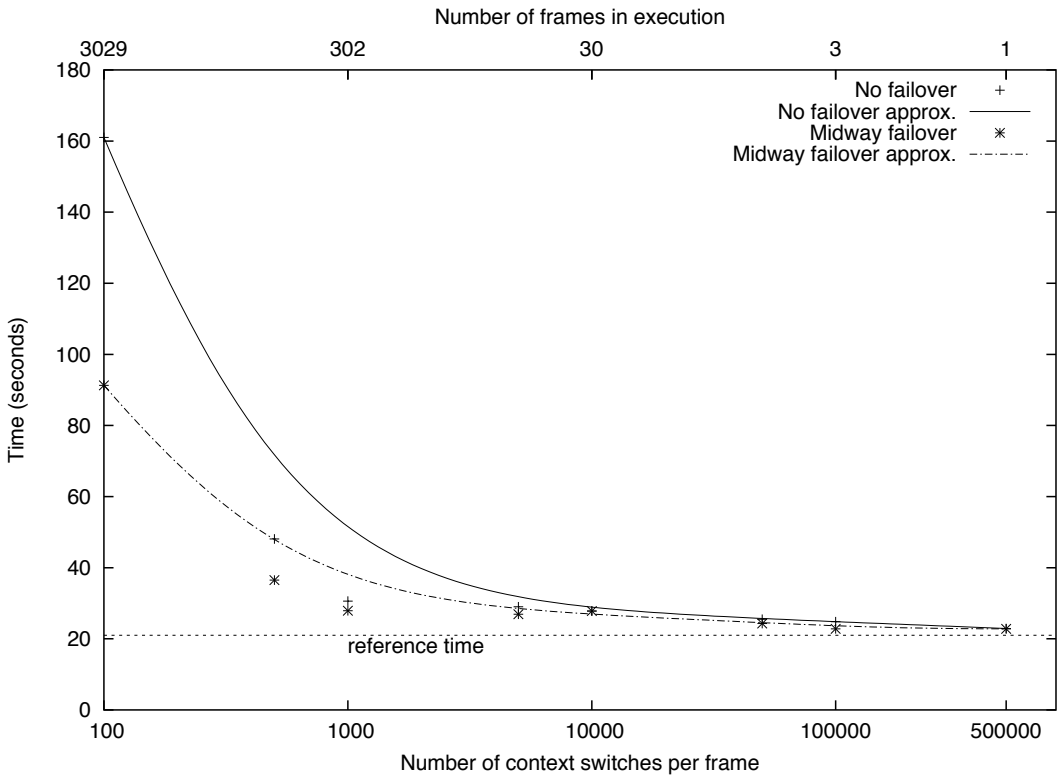

Figure 6: Ray-tracing benchmark with a forced failover of the primary in mid-execution

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 similarly 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.

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 7. The application run time shows the portion of the time dedicated

to executing the target application. The send queue bottleneck measures the time between 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 from the chart that the type of an application has an effect on the performance of replication. Specifically, an application that generates many I/O operations will burden the replication engine with many messages that lead to a larger delay in sending them to the backups. 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 is increased.

Figure 8 illustrates the performance of a ray-tracing benchmark test running on two processors. This raytracer (_227_mtrt) utilizes two threads so as to maximize concurrency. As can be seen, running the 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 backups.

We would like to point out that each of the applications of the SciMark benchmarks were executed separately, instead of the more common batch execution. This was done in order to circumvent a technical condition that creates frames of significantly different sizes, for example in the case where a short frame follows a very large frame. The primary cannot move past the short frame until it receives messages from the backups, which are still executing the large frame. Therefore the primary will lie idle for a period of time that roughly equals the size difference between the two frames.

This variance in the sizes of frames can occur in JikesRVM and is due to the way it counts

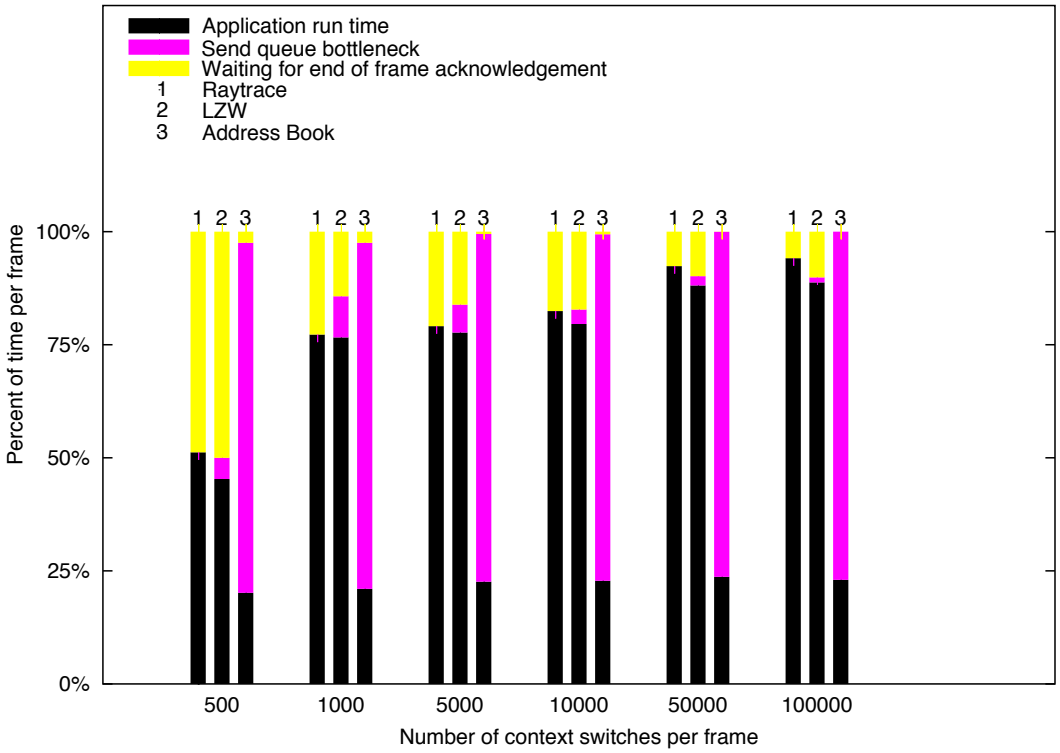

Figure 7: Cost breakdown in replicated execution of various applications

the safe yield points, where passing a certain number of them will trigger a context switch. These yield points relate to the program's structure: a tight loop or a fragment of code that makes many method calls will have many yield points, while a code fragment that does not contain many jumps will have few. Therefore, in the case of JikesRVM's deterministic thread switching, code that is of uniform structure will have uniform time slices, while code with drastically different structure will exhibit a variance in the time that a thread will run.

The above behavior is exhibited in the execution of the SciMark benchmarks. The suite is composed of different programs that are run in succession. This property resulted in one test running within a long frame while another test began within the following short frame. The second test performed poorly because its run contained a long pause before continuing to the next frame. Consequently its performance result was not reflective of its true performance on

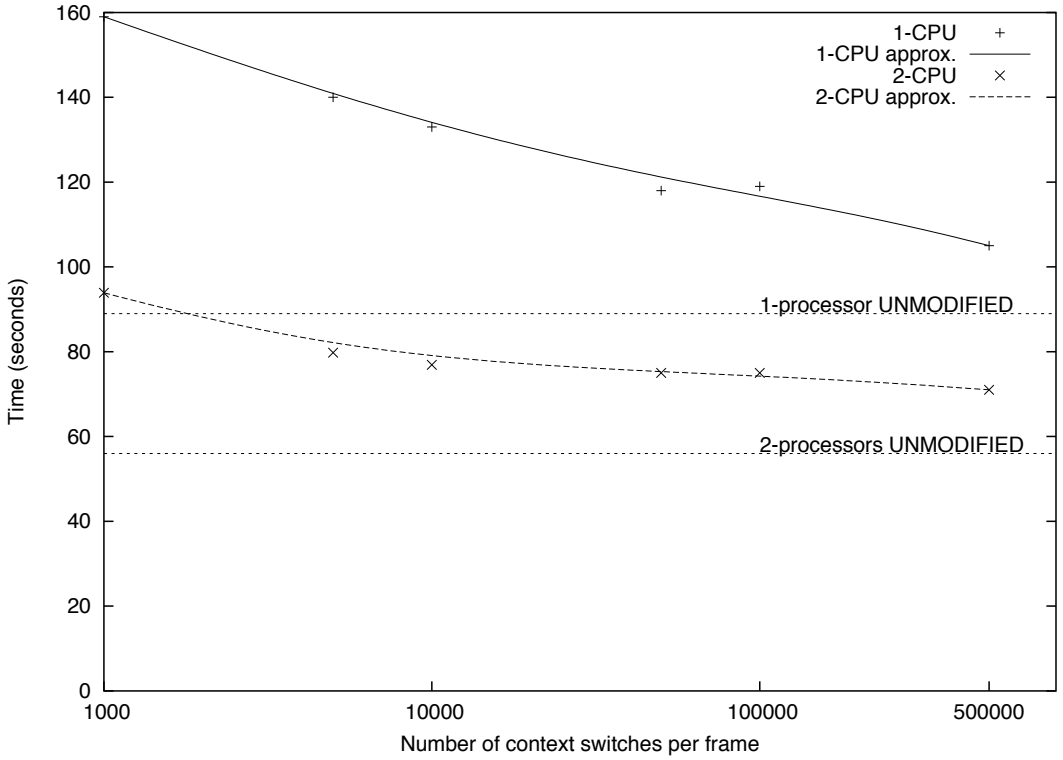

Figure 8: Ray-tracing benchmark, 2 threads on SMP

the JVM when executed individually. The true performance can be conveyed when each test in the suite is executed individually.

A solution to this problem is to change the mechanism by which a frame boundary is decided upon, from the currently used coarse grain counter to a more fine gained one, e.g., based on counting individual instructions. However, such a design should consider and take steps to control the overhead of bookkeeping and coordination that would be involved.

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 SMP, we assume that all shared data is protected by semaphores 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 do not have control over the progress of such executions, nor can we guarantee that actions performed on the primary will execute the same way on the backups. 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 backups, such as TCP frame information. The means to accomplish this have been studied in [15, 17, 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 straightforward: information on all open files, whether opened physically by the primary or virtually by the backups, 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 backups due to environmental differences. An example is an application dictates that a string should be written in a file at a location that is dependent on the computer 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.

Finally, as is evident from the performance measurements, our implementation incurs high overhead for applications that read very large amounts of data, and in particular, database applications. For such applications, another solution is needed.

6 Related Work

The use of replication as a technique for implementing fault tolerance has been extensively used. The common methods rely on the *replicated state machine approach* [30] or the *primary–backup approach* [13], which 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.

The above models do not specify how to deal with concurrent requests and the resulting multithreaded environment, which may lead to race conditions. A replicated state machine requires a total ordering on all client requests to the replicas. Also, the model functions under the assumption that state changes can be succinctly communicated to the replicas and that such changes can be easily applied by them. It does not address side-effects such as accessing a file that is shared by all replicas.

For the cases where 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 will 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 [19] 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 a SMP.

Another related field is the deterministic replay of executions [14, 21], 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 [9, 32] as well as in the operating system [26] and middleware [6, 10, 20, 23].

The basis for our work is the hypervisor approach, detailed in [12]. 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.

The Voltan application environment creates fail-silent processes by replicating processes [11]. Voltan eliminates all sources of non-determinism and compare 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, similarly 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 of 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. Furthermore the JikesRVM uses JIT, which was not available for the Solaris JDK used in [27]. We specifically addressed issues related to running the JVM on a SMP, which entails a higher overhead of synchronization but a greater utilization of computational resources. 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 more reliable 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). Both works together serve as a proof of concept for the viability of implementing fault-tolerance at the virtual machine level.

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 de-

scribe 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.

As virtual machine based languages such as Java and C# become more common, our approach becomes compelling. In particular, much of the work in [12] was devoted to realizing a virtual machine, whereas in the case of virtual machine based languages, it already exists. 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. Although untested, our extension of the virtual machine should work as-is across the multiple hardware architectures and operating systems that JikesRVM supports, including heterogeneous replication.

7.1 Future Work

We would like to pursue the issue of load balancing of threads on an SMP while maintaining deterministic behavior. This will aid in the case of a performance degradation where one processor on a dual-processor machine (we'll assume this is on the primary) is burdened by CPU-intensive processes that are external to the JVM. If such external overhead does not exist on the peer backup, then the backup will needlessly sit idle, waiting for the primary's process that must compete for scheduling slots. It may be possible to offload certain threads from the slow processor onto a "virtual processor" that will run on the second (presumably more free) processor, together with the existing JVM process. Care must be taken so that the scheduling of the threads, now distributed over two processors, will remain deterministic and fully reproducible on the backup machine.

Another area that we would like to study is the synchronization protocol between primary and backups, where the model is expanded to support multiple backups. Currently the primary must wait for all backups to complete the previous frame before continuing to the next frame.

There exists a tradeoff between waiting for all the ACKs to arrive, in which case all the backups are in a consistent state, and waiting only for the first ACK. The latter option may allow for faster processing, at the risk of falling out of sync and dynamically reducing the t-fault-tolerant claim of the system.

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 Benchmarks. http://www.specbench.org/osg/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):194–211, 2000.

[5] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proc. of the 2nd International Conference on Software Engineering*, pages 562–570, San Francisco, CA, 1976.

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

[7] J.S. Banino and J-C Fabre. Distributed Coupled Actors: A CHORUS Proposal for Reliability. In *3rd International Conference on Distributed Computing Systems (ICDCS-3)*, pages 128–134, Miami, FL, 1982.

[8] P.A. Barrett, A.M. Hilborne, P.G. Bond, D.T. Seaton, P. Veríssimo, L. Rodrigues, and N.A. Speirs. The Delta-4 Extra Performance Architecture (XPA). In *20th International Symposium on Fault Tolerant Computing Systems (FTCS-20)*, pages 481–488, Newcastle Upon Tyne, U.K., 1990.

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

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

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

[12] 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.

[13] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. In Sape Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison-Wesley, 2nd edition, 1993.

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

[15] 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.

[16] E. N. Elnozahy, L. Alvisi, Y.M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[17] Roy Friedman, Kenneth Birman, Srinivasan Keshav, and Werner Vogels. Reliable time delay-constrained cluster computing. United States Patent 6,393,581, May 2002.

[18] Walter L. Heimerdinger and Charles B. Weinstock. A Conceptual Framework for System Fault Tolerance. Technical Report CMU/SEI-92-TR-33, Carnegie Mellon University, Pittsburgh, PA, USA, October 1992.

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

[20] Idit Keidar and Danny 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.

[21] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic Replay of Distributed Java Applications. In *Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium*, pages 21–228, 2000.

[22] Hermann Kopetz and Paulo Veríssimo. Real Time and Dependability Concepts. In Sape Mullender, editor, *Distributed Systems*, chapter 16, pages 411–446. Addison-Wesley, 2nd edition, 1993.

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

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

[25] Jean-Claude Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *15th International Symposium on Fault Tolerant Computing Systems (FTCS-15)*, pages 2–11, Ann Arbor, MI, USA, 1985.

[26] Drew Major, Greg Minshall, and Kyle Powell. An Overview of the NetWare Operating System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 355–372, 1994.

[27] Jeff Napper, Lorenzo Alvisi, and Harrick Vin. A Fault-Tolerant Java Virtual Machine. Technical Report TR-02-56, The University of Texas at Austin, Department of Computer Sciences, May 2002. Also, in Supplement of the 2002 International Conference on Dependable Systems and Networks, Student Forum, June 2002, pages A1–A3.

[28] L. E. Moser P. Narasimhan 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] J. S. Plank, M. Bech, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing Under UNIX. In *Usenix Winter 1995 Technical Conference*, pages 220–232, New Orleans, January 1995.

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

[31] Fred B. Schneider. The state machine approach: a tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.

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

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