Improving Java Virtual Machine Reliability for Memory-Constrained Embedded Systems*

Guangyu Chen and Mahmut Kandemir
Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802, USA
{gchen, kandemir}@cse.psu.edu

ABSTRACT

Dual-execution/checkpointing based transient error tolerance techniques have been widely used in the high-end mission critical systems. These techniques, however, are not very attractive for cost-sensitive embedded systems because they require extra resources (e.g., large memory, special hardware, etc), and thus increase overall cost of the system. In this paper, we propose a transient error tolerant Java Virtual Machine (JVM) implementation for embedded systems. Our JVM uses dual-execution and checkpointing to detect and recover from transient errors. However, our technique does not require any special hardware support (except for the memory page protection mechanism, which is commonly available in modern embedded processors), and the memory space overhead it incurs is not excessive. Therefore, it is suitable for memory-constrained embedded systems. We implemented our approach and performed experiments with seven embedded Java applications.

Categories and Subject Descriptors

D.3.m [**Software**]: Programming Languages—*Miscellaneous*

General Terms

Reliability

Keywords

Java Virtual Machine, dual execution, transient error

1. INTRODUCTION

Java is increasingly being used in embedded environments [2, 3]. The success of Java technology in the embedded systems can be ascribed to several factors, including portability, safety, ease of programming, and mature developer community. Many non-trivial Java applications, such as web browsers and email processors, are popular on the small devices such as mobile phones and PDAs. In the near future, one can even see mission-critical applications (e.g., online transaction processing) running on mobile-phone-like devices. Following this trend, fault-tolerance will be an important issue for Java-enabled embedded environments.

*This work is supported in part by NSF Career Award #0093082 and by GSRC.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

A certain class of transient hardware faults in modern microprocessors are usually caused by cosmic rays to which we are exposed everyday [4]. As the feature size and the voltage of integrated circuits keep scaling down, the future microprocessors are becoming increasingly susceptible to transient hardware faults. Many solutions have been proposed for improving the reliability of computer systems against these faults. For example, dual-execution techniques enable us detect the transient errors in the datapath [9, 12]. However, such solutions require special hardware support, and thus, increase the overall cost of the system. Embedded systems are usually sold in huge quantities and thus tend to be more sensitive to the per device cost as compared to their high-performance counterparts. Consequently, the existing fault-tolerance solutions for high-end systems may not be attractive for low-cost embedded systems. Further, an embedded system may run a set of applications and not all of them may require fault-tolerance. Employing expensive hardware for just a few applications that need fault-tolerance may not be the best economic option.

Intuitively, a transient fault tolerant JVM can be designed as follows. First, we use two bytecode execution engine instances (E_1 and E_2) to execute the same copy of an application, and compare their states (including the state of each object in the heap[1], and the state of each register of the CPU) at certain points of the execution. A mismatch in these two states normally indicates an error. Second, from time to time, we take checkpoints by copying the current states of E_1 and E_2 to a reserved memory area so that we can roll back to the state of the previous checkpoint in case an error is detected. However, this approach has two major drawbacks. First, we need memory space to store three copies of the execution state, two copies for the two execution engine instances and one copy for the checkpoint. Consequently, compared to a JVM without transient fault tolerance, this approach increases the memory requirement of a Java application by 200%, which makes it unsuitable for memory-constrained systems. Second, comparing the states of each corresponding pair of Java objects incurs a heavy performance overhead.

The contribution of this paper is that we focus on embedded JVM as a case study and develop a scheme that allows E_1, E_2, and the checkpoint to share a significant portion of the objects in the heap. Our approach has the following advantages, which make it very attractive for a low-budget embedded environment. (1) By sharing objects, the memory overhead due to fault-tolerance mechanism is significantly reduced. (2) Performance overhead due to comparing the states of E_1 and E_2 is reduced since there is no need to compare the states of the objects that are shared by both the execution engine instances. (3) Our scheme does not require any special hardware support, except for the memory page protection mechanism, which is currently supported by the

[1]The stacks of Java threads are allocated in the heap memory.

MMU (Memory Manage Unit) of many embedded processors. (4) Our scheme is flexible in the sense that E_1 and E_2 can be scheduled either dynamically or statically. In the dynamic scheduling approach, each execution engine instance is implemented as a system thread that is scheduled by the operating system. E_1 and E_2 synchronize with each other using the synchronization APIs provided by the operating system. This approach is suitable for multiprocessor environments where two execution engine instances can run in parallel to achieve high performance. On the other hand, the static scheduling approach uses a single system thread to run both the execution engine instances. In this case, the programmer of the JVM inserts scheduling points in the code of the execution engines to explicitly switch contexts between E_1 and E_2. This scheme is suitable for the single-processor environments or for the environments where the operating system does not support multi-threaded execution. When using the static scheduling, the synchronization overhead is reduced since, at any time, there is only one thread running.

The two execution engine instances (E_1 and E_2) synchronize with each other at checkpoints to compare their states. Our approach can detect any type of transient error that causes the states of the two execution engine instances differ from each other. To recover from the errors by rolling back, we assume that the memory system (including MMU and the main memory) is reliable, which can be achieved by using well-known error detection and correction codes.

The rest of this paper is organized as follows. The next section presents the implementation of our transient fault tolerant JVM in detail. In Section 3, we study the memory space and performance overheads caused by our JVM using seven embedded Java applications. Section 4 discusses the related work. Finally, in Section 5, we conclude the paper.

2. IMPLEMENTATION

2.1 Object Model

Figure 1 shows the object model for our transient fault tolerant JVM. In this model, each object has a two-word header, containing the size and ID of the object. The ID is also be used as the default hash code for the object [10]. Each object is referenced by an entry of the object table, which is indexed by the object ID. The reference fields of the objects contain the IDs of the target objects, instead of direct pointers. A drawback of such indirect object references is that we need an extra dereferencing when accessing the contents of an object, and that the object table incurs some space overhead. The main advantage is that the objects can be moved freely without updating the reference fields in other objects, and this is why we adopt this object representation in our implementation.

In our implementation, almost all the data structures, including the user created objects, the code of the Java applications, the execution stacks of the Java threads, and the other data structures internal to the implementation of the JVM, are allocated in the heap. Many objects are accessed very frequently. Accessing these objects using the indirect reference each time incurs performance/power overheads. To address this problem, we employ pre-dereference optimization to avoid the indirect reference overheads for the most frequently accessed data structures. For example, the execution stacks of Java threads are the most frequently accessed data structures in the heap. Each execution engine instance maintains a pointer to the execution stack of the thread that is currently being executed. To avoid the cost for indirect reference at each access to the execution stack, whenever a thread is scheduled, we load the actual address of the execution stack of the scheduled thread into a register, i.e., we pre-dereference the indirect pointer to the execution stack. When the memory block for the execution stack of the scheduled thread is moved during garbage collection, we need to update the register that contains the address of the current execution stack. Similarly, we pre-dereference the pointer to the memory block that contains the bytecode of the method that is currently being executed to avoid the indirect reference overhead in fetching bytecode instructions.

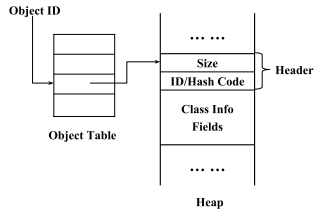


Figure 1: Object model for our transient fault tolerant JVM.

2.2 Object Sharing

Our approach reduces the memory space overhead by allowing the two execution engine instances and the checkpoint to share as many objects as possible. In this section, we explain this sharing mechanism. Figure 2 depicts the architecture for our transient fault tolerant JVM. Our JVM maintains a heap that is physically divided into pages. Each page can be either in the read-only mode or writable (read/write) mode. Writing to a read-only page triggers a memory protection exception. Each execution engine instance has its own object table. Each entry in each object table contains a pointer to an object in the heap. The heap is logically divided into three subheaps: a global subheap (H_g) and two local subheaps (H_1 for E_1 and H_2 for E_2). The size of each subheap can change during execution. The boundary between two adjacent subheaps aligns with the page boundaries. The objects in H_g may be shared by both the execution engine instances, i.e., a pair of entries in the two object tables indexed by the same object ID may refer to the same object in H_g. The pages in H_g are read only during the execution time. During global garbage collection (which will be discussed in Section 2.4.2) time, however, a page of H_g can be put in writable mode. During the execution, writing to an object in a H_g triggers a memory protection exception. The exception handler copies the object being accessed to the free area of H_i ($i = 1, 2$), assuming that it is E_i that causes the exception. The corresponding entry of E_i's object table is also updated so that the future accesses to this object made by E_i are redirected to the new copy of this object (in H_i). The state of the object before the write access, however, is preserved in H_g. Since H_g is read-only, this state cannot be modified by transient errors in the processor core. Consequently, when any error is detected, we are guaranteed to be able to roll back to the correct state. Figure 3 illustrates the procedure of writing to an object O in H_g. Note that, in embedded operating systems, the overhead for exception handling is usually much smaller than that in high-end operating systems. In addition, this overhead can be further reduced by integrating the JVM to the kernel of the embedded operating system. The objects in H_i are moved to H_g during the global garbage collection.

Transient errors may also cause the JVM to mistakenly put an H_g page into writable mode. Our scheme can not completely prevent the failure of execution due to such errors. However, we can significantly reduce such risk by using "page mode checking" exception. Specifically, a "page mode checking" exception is triggered whenever an execution engine instance tries to place a read-only page into the writable mode. The exception handler checks if the JVM is performing a global garbage collection. Only during global garbage collections can a page of H_g be put in writable mode. Any attempt to put a read-only page into writable mode in occasions other than global garbage collections indicates an error

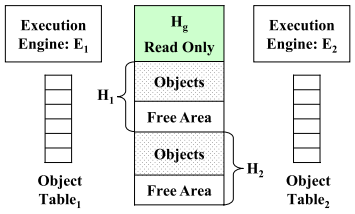


Figure 2: The architecture of the transient fault tolerant JVM.

and causes the execution engine instance that triggers the "page mode checking" exception to roll back to the previous checkpoint.

2.3 Checkpointing and Rolling Back

Our transient fault tolerant JVM implicitly takes checkpoints at two occasions: (1) right after the global garbage collection, and (2) at the invocation and return points of a non-dual-executable method. The following subsections will discuss the details of taking the checkpoints. To roll back to the previous checkpoint, we first set each entry in the object tables of both the execution engine instances to NULL. Then, we scan the objects in H_g in the order of their allocation times, and use the address and the ID field of each scanned object to set up the corresponding entry in the object table. Note that the order used in this scanning is critical, i.e., if an object O_1 is created before O_2, O_1 must be scanned earlier than O_2. This is due to the possibility that an object might be duplicated multiple times, and only the latest copy contains the correct state of the object before the previous checkpoint. By scanning the objects in the order of their creation times, we guarantee that the object table entry contains the pointer to the newest version of each object. Finally, both the execution engine instances load the processor state that was stored at the previous checkpoint and resume execution.

2.4 Garbage Collection

JVM relies on garbage collector to manage heap memory. Letting some object to be shared by two execution engine instances requires modifications to how garbage collection is carried out. Specifically, our transient fault tolerant JVM uses two different garbage collectors: a local collector operating on local subheaps, and a global collector operating on the entire heap. When execution engine instance E_i $(i = 1, 2)$ uses up the memory in H_i, it invokes the local garbage collector to reclaim the space occupied by the garbage objects (i.e., the objects that are unreachable from the root set) in H_i. If the local garbage collector cannot find sufficient space for the new object, the global garbage collector will be invoked to collect garbage objects in the entire heap (H_1, H_2, and H_g).

2.4.1 Local Collector

Each execution engine instance invokes the local collector to collect its local subheap when it uses up the space in its local subheap. The two execution engine instances do not necessarily invoke local collectors simultaneously, and, during the garbage collection process, they do not synchronize with each other. The local collection is performed in two phases: a mark phase and a compact phase. In the remaining part of this section, we describe our local collector in detail. We assume that we are collecting subheap H_i $(i = 1$ or $2)$.

Mark phase: In this phase, the collector for E_i traverses the reference graph[2] to mark the objects that are reachable from the root set along the pointers in the reference fields of each object. The root set is the set of objects that are known to be live when the garbage collector is invoked. For local collection, the root set includes the following objects: (1) The objects that were copied from H_g by the memory protection exception handler; (2) The stack frames of the live threads in H_i; (3) The instances of class "Class" in H_i; and (4) Other JVM implementation-specific data structures in H_i.

Compact phase: In this phase, we slide the live objects that were marked in the mark phase to one end of subheap H_i. Consequently, we obtain a contiguous free area in the other end of H_i. When an object is moved, the corresponding pointer in the object table entry (see Section 2.1) should also be updated accordingly. Since there is no need to update the reference fields of the objects, the implementation of the compact phase is straightforward.

Errors happening during local garbage collection will cause the states of the two execution engine instances to diverge[3], and this divergence will be detected at the next checkpoint. Since the local garbage collector does not change the contents of the objects in H_g (which are in the read only mode), it cannot damage the state of the previous checkpoint. Therefore, we can recover from the errors that happen during the local garbage collection by rolling back to the state of the previous checkpoint.

2.4.2 Global Collector

The global collector collects the garbage in the entire heap. Since global garbage collection affects the states of both the execution engine instances, it requires E_1 and E_2 to synchronize with each other. When designing the global garbage collector, there are two major challenges. The first one is how to detect errors during global garbage collection. Note that we have only one copy of H_g, and thus it is impossible to detect errors by comparing the states of two global collector instances. The second challenge is how to recover from the errors that occur during the global garbage collection. Since the global collector changes the contents H_g, it damages the state of the previous checkpoint. A straightforward solution is to copy the entire H_g to a safe place. When an error happens, we can roll back to the previous state by copying back the state of H_g. However, this solution is not attractive for a memory-constrained embedded system due to its heavy memory overhead. The rest of this section presents our solution to these challenges. Our solution requires the global garbage collector follow the steps below:

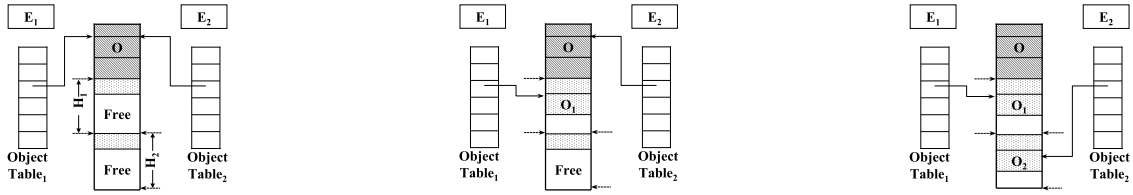
Step 1. Each execution engine instance (E_i) independently traverses the reference graph in both H_g and its local subheap (H_i) to mark the objects reachable from the root set. We use the least significant bit of each object table entry as the mark flag. Note that, since the objects are aligned to word (4 bytes) boundaries, this bit is not used by a reference. When marking objects, we also compute the CRC checksum of the contents of all the live objects using a table-based algorithm [14].

Step 2. E_1 and E_2 synchronize with each other to compare the checksums computed in step 1. If the two checksums do not match, both E_1 and E_2 have to roll back to the previous checkpoint because errors might have occurred since the previous checkpoint. Otherwise, we continue with step 3.

Step 3. Reaching this step indicates that E_1 and E_2 have the same live objects. Now, we are going to compact life objects. Note that, at this point, the contents of H_1 and H_2 are identical. We combine H_g and H_1 into a "collection area" and set all the pages in this area to read-only mode. We also make the object table of E_1 (where the live objects are marked) readable to both E_1 and E_2. From now on, both

[2]A reference graph is a directed graph, each node of which corresponds an object in the heap, and each edge of which corresponds to a reference field of the source object.

[3]It is very unlikely that the same errors would occur in two local collections.



(a) Initially, the entries of both object tables point to object O in H_g.

(b) Upon E_1 writing to the field of O, O is copied into H_1.

(c) Upon E_2 writing to the field of O, O is copied into H_2.

Figure 3: Writing to an object in H_g.

E_1 and E_2 work on the collection area and the object table of E_1. H_2 and the object table of E_2 are not used in the following steps. Setting the collection area and the object table to read-only is to prevent any error during garbage collection from damaging the state of the heap.

Step 4. We use cursor S_i $(i = 1, 2)$ to indicate the object that is currently being processed by E_i. S_i is initialized to the beginning of the collection area. We use pointer P to indicate the target page into which the live objects will be compacted. P is initialized to 0, i.e., to point to the first page of the heap.

Step 5. We use a buffer page to temporarily store a set of live objects (Figure 4(a)). The size of the buffer page is the same as the size of the pages of the heap. E_1 uses S_1 to scan the heap and copies the scanned live objects into the buffer page until the buffer page is full. E_2 uses S_2 to scan the heap and computes the checksum (C_1) of the scanned live objects until the total size of the scanned objects reaches the size of the buffer page. An object with ID x is considered to be live if and only if the x^{th} entry of the object table of E_1 is marked and points to the address of this object.

Step 6. E_1 sets the buffer page to read-only mode, computes the checksum (C_2) of the objects in this page, and then compares it against C_1 (Figure 4(b)). If $C_1 \neq C_2$, we rewind S_1 and S_2 to their previous positions and then go to step 5.

Step 7. E_1 sets the mode of page P to writable, copies the objects in the buffer page into this page, and then sets the mode of page P back to read only. After setting page P to read only, E_1 computes the checksum (C_3) of the objects in this page (Figure 4(c)). If $C_3 \neq C_2$, there might be errors during copying. Therefore, we repeat step 7 until we have $C_3 = C_2$.

Step 8. We increase P by one (i.e., move to the next page). If all the objects in page P have been scanned by S_1 and S_2 (i.e., the end address of page $P < S1 = S2$), we continue with step 9; otherwise, we jump back to step 5.

Step 9. Reaching this step indicates that we have collected enough free space so that the gap between the compacted area and the uncompacted area is larger than a page. Consequently, in the following steps, we are able to compact the live objects to the target page P directly, without using the buffer page. E_1 sets the mode of page P to writable and then uses S_1 to scan and copy the live objects to page P until P is full. E_2 uses S_2 to scan the heap and computes the checksum (C_1) of the scanned live objects until the total size of the scanned objects reaches the size of page P (Figure 4(d)).

Step 10. E_1 sets the mode of page P to read-only, and then computes the checksum (C_2) of the objects in this page (Figure 4(e)). If $C_1 \neq C_2$, we rewind S_1 and S_2 to the previous positions, and then go back to step 9.

Step 11. We increase P by one (i.e., move to the next page). If S_1 and S_2 have not reached the end of the collected area, go to step 9 (Figures 4(f) and (g)).

Step 12. Mark page$_0$ through page$_{P-1}$ as H_g. The rest of the pages in the heap are equally distributed between H_1 and H_2. Make the pages in H_1 and H_2 writable to E_1 and E_2, respectively (Figure 4(h)).

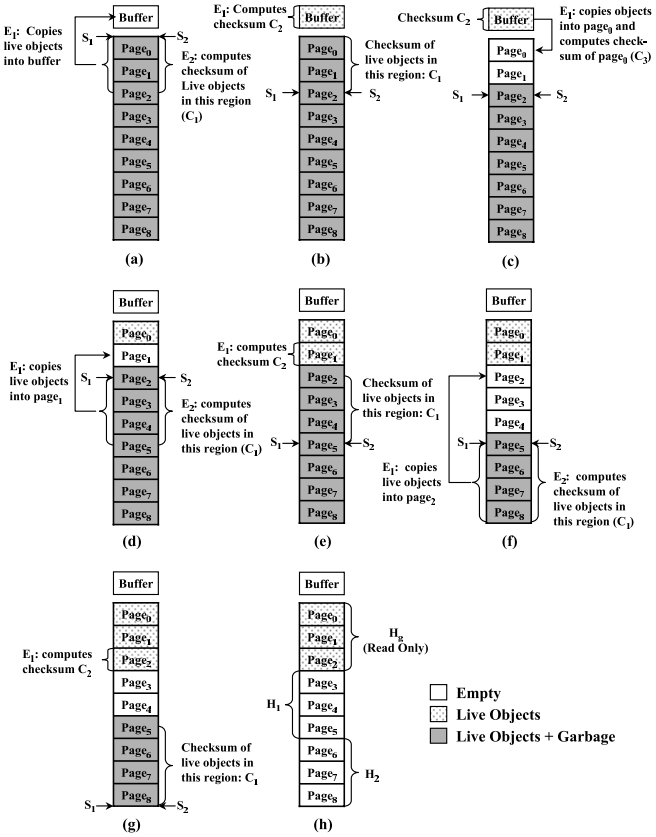


Figure 4: Global garbage collection.

Step 13. Both E_1 and E_2 set the entries of their object tables to NULL, and then they scan H_g, using the addresses and the ID fields of the scanned objects to update the entries of their object tables.

Step 14. Both E_1 and E_2 resume execution of the application code.

From the above description, we observe that, during the global collection, no two pages can be in the writable mode simultaneously. This is to minimize the risk that a transient error damages the state of the heap. Writing to a read-only page during garbage collection indicates an error. Further, the pages in the heap are put in the writable mode one-by-one in the order of their addresses. This information can be exploited to enhance the reliability of the global collection. Specifically, when putting page$_i$ into the writable mode, the page mode checking exception handler checks if the page in the writable mode last time was page$_{i-1}$. If it is not page$_{i-1}$, we know that an error must have occurred.

2.5 Handling Non-Dual-Executable Methods

A non-dual-executable method is a method whose side effects or return value is sensitive to the time of invocation. Such a method may return a different value or affect the

state of the application differently when invoked at different times. Executing a non-dual-executable method on two execution engine instances may cause the states to diverge. To avoid such a divergence, we execute such methods using only one execution engine instance and then copy the return values and the objects that are created by the non-dual-executable methods into the H_g so that they can be accessed by the other execution engine instance as well. To invoke a non-dual-executable method, we follow the following steps:

Step 1. Both E_1 and E_2 invoke a local collection and then compute the checksums of the live objects in the their local subheaps.

Step 2. E_1 and E_2 synchronize with each other to compare the checksums computed in step 1. The difference in the checksums indicates an error, and then both E_1 and E_2 must roll back. If no error is detected, we promote the live objects in H_1 to H_g. Note that H_1 is adjacent to H_g, therefore, a promotion can be performed by simply increasing the size of H_g. After the promotion, the pages that do not belong to H_g are equally distributed between subheaps H_1 and H_2.

Step 3. At this point, both H_1 and H_2 are empty. E_2 pauses while E_1 continues with the execution of the non-dual-executable method.

Step 4. E_1 returns from the method.

Step 5. E_1 invokes a local collection, promotes the live objects in H_1 into H_g, and distributes the pages that do not belong to H_g equally between H_1 and H_2.

Step 6. E_2 updates the references to the promoted objects in its object table.

Step 7. Both E_1 and E_2 resume execution.

In the above description, it should be noted that we invoke the local garbage collection twice. The overall performance overhead due to non-dual-executable methods, however, is not as significant as it seems to be. The first reason for this is that most of the non-dual-executable methods are methods performing I/O operations, such as reading user input from the keyboard. Compared to the long pauses incurred by these types of I/O operations, the additional overhead due to two local collections is not expected to be very significant. The second reason is that the non-dual-executable I/O methods usually create only a small number of objects (if any); and consequently, the extra cost due to the second local collection tends to be very small. Further, these two local collections reclaim free space in the local heaps, and thus postpone the future invocation of garbage collection.

2.6 Scheduling Java Threads

A Java program can create multiple Java threads. In our transient fault tolerant JVM, all the Java threads are user level threads that are scheduled by the JVM; and a Java thread that is created by E_i is scheduled only by E_i. Further, to prevent the state divergence, E_1 and E_2 schedule the Java threads in the identical order.

3. EXPERIMENTS

3.1 Experimental Setup

We implemented our transient fault tolerant JVM based on KVM [1]. KVM is a JVM implementation for the embedded devices with limited memory budget. The total memory space required for loading KVM and its Java class library is about 350KB. The memory size for the heap, however is determined by the specific application being executed. The fourth column of Table 1 shows the minimum heap size that is required to run each of our benchmarks on the original KVM. To evaluate our approach, we execute our JVM in an embedded environment, which is simulated using the Shade [8] instruction level simulation toolkit.

Table 1 presents the information about the benchmarks that are used in our experiments. They represent a typical set of benchmarks that are executed in embedded Java environments. The third column in this table gives the total size of the objects that are allocated during the execution of each benchmark. The fourth column gives the maximum heap occupancy (i.e., the maximum value of the total size of the objects that are live simultaneously in the heap) for each benchmark. Finally, the last column is the maximum value of the number of objects that are live in the heap simultaneously. This value determines the minimum number of entries in the object table.

Benchmarks	Description	Accumulated Allocation Size	Max. Heap Occupancy	Max. Number of Live Obj.
hello	Print "hello"	8KB	4KB	40
chess	Chess game	209,430KB	27KB	2194
crypto	Encryption package	1,528KB	116KB	1181
db	Database management	101KB	43KB	196
edge	Edge detection	241KB	216KB	268
fft2d	2-D Fourier filter	94KB	81KB	40
xml	XML parser	143KB	65KB	1026

Table 1: The Java benchmarks for our experiments.

Benchmark	Original Max. Heap Occupancy	Min. Heap Size with Fault Tolerance	
		Size	Normalized
hello	4KB	8KB	200%
chess	27KB	32KB	119%
crypto	116KB	132KB	114%
db	43KB	48KB	112%
edge	216KB	350KB	162%
fft2d	81KB	86KB	106%
xml	65KB	88KB	135%

Table 2: Minimum heap space ($H_g + H_1 + H_2$) requirement for our transient fault tolerant JVM. The values in the fourth column are normalized with respect to those in the second column. The memory page size is 2KB.

3.2 Experimental Results

Our dual-execution based error detection and checkpoint based error recovery mechanism incurs both memory space and performance overheads. In this section, we quantify these overheads.

Table 2 gives the memory overheads incurred by our transient fault tolerance mechanism. The memory page size used in the experiments is 2KB. The second column in this table gives the original maximum heap occupancy for each benchmark, which is also shown in Table 1. The third and fourth columns give the absolute and normalized (with respect to the original maximum heap occupancy) values of the minimum heap size that is required by our fault tolerance JVM to execute each benchmark without an "out-of-memory" exception. In Table 2, we observe that, for the benchmark hello, an application whose original memory requirement is small, the memory overhead incurred by the transient fault tolerant mechanism is large (100%). This is because the size of each subheap must be rounded up to the minimum multiplier of the page size. Note that, the page size used (2KB) is large, compared to the maximum heap occupancy of this benchmark. For the applications whose original memory requirements are large, the impact of the page round-up problem reduces. We also observe that the memory overhead of the benchmark edge is larger (62%) than most of the other benchmarks. This can be explained by observing that this benchmark allocates large arrays (about 64KB each) and that we need to keep up to three copies of each array in three subheaps simultaneously when the array is write-accessed. For most of our benchmarks, however, the memory overheads are within 20%, which indicates that our object sharing technique reduces the memory overheads significantly.

Figure 5 shows the breakdown of the execution times when no error occurs during the entire execution time. We experiment with both single-CPU and two-CPU based embedded

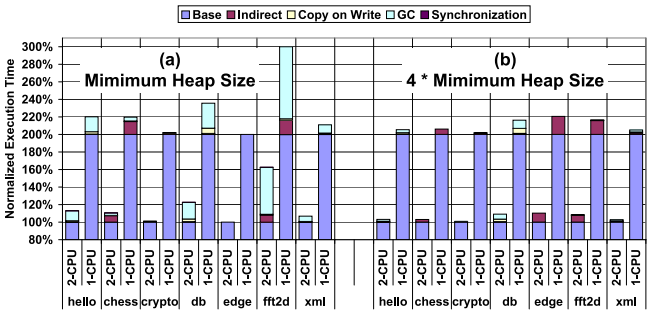


Figure 5: Normalized execution times of our transient fault tolerant JVM. The execution time is normalized with respect to the overall execution of the original KVM with the same heap size. Base: the time for executing Java bytecode. Indirect: the overhead due the indirect object accesses via the object tables. Copy on Write: the overhead for copying objects from H_g to H_1 and H_2 on write accesses. GC: the time for garbage collection, including both local and global collections. Synchronization: the overhead for synchronizing E_1 and E_2 at checkpoints and during global garbage collections. (a) Normalized execution time with the minimum heap size shown in the third column of Table 2. (b) Normalized execution time with the heap size that is four times of the minimum heap size.

environments. In the two-CPU environment, E_1 and E_2 execute in parallel; and in the single-CPU environment, E_1 and E_2 are scheduled on the same CPU. We conduct experiments with two different heap sizes: the minimum heap size that allows each benchmark to execute without an "out-of-memory" exception, which is shown in the fourth column of Table 1 (Figure 5(a)); and the heap size that is four times of the minimum heap size (Figure 5(b)). The execution time of each benchmark is normalized with respect to the overall execution time of the original KVM with the same heap size. Each bar is broken into five components as explained in the caption of this figure. In Figure 5, we observe that, when using single CPU, our JVM increases the execution of each benchmark by more than 100%, compared to the original KVM. The major reason for this overhead is that we execute the application twice in the single CPU. In the two-CPU environment where we can execute E_1 and E_2 in parallel, the performance of our transient fault tolerant JVM gets closer to the original KVM (that uses single CPU). In Figure 5(a), we also observe that, when using the minimum heap size, the garbage collection in the JVM can incur significant overheads (up to 53% in the two-CPU environment) since the collector has to be invoked very frequently. By increasing the heap size to four times of the minimum heap size, the overhead due to garbage collection is reduced to a marginal extent. Besides the garbage collection, indirect object references via the object tables incur another important performance overhead. By using the pre-dereference optimization described in Section 2.1, we managed to reduce this overhead to less than 10% (3% on the average) of the overall execution time.

4. RELATED WORK

Transient error detection and recovery has been an active research area for several years. Ray et al. [11], and Reinhardt and Mukherjee [12] propose architectural schemes that use the spare functional units in a superscalar processor to detect and recover from the errors in the datapath. Gomaa et al. [9] use on-chip multiprocessors for transient error detection and recovery. These approaches require hardware support that is not commonly available in low-end embedded systems. Satyanarayanan et al. [13] propose a lightweight recoverable virtual memory system for Unix applications with persistent data structures that must be updated in a fault-tolerant manner. Caldwell and Rennels [5] present a transient fault tolerant scheme for embedded spacecraft computing. Their schemes use multiple processors to perform the same computing and let them vote for the correct results. Chen et al. [6] report fault injection experiments that investigate memory error susceptibility of JVMs. Chen et al. [7] analyze the heap error behavior in embedded JVM environments. As compared to these studies, our approach detects and recovers from transient errors using dual execution. An important characteristic of our approach is that it reduces memory overheads brought by dual execution and checkpointing.

5. CONCLUSIONS

In this paper, we present transient fault tolerant JVM for embedded environments. Our JVM use two execution engine instances to execute the bytecode of the same application and compare the states of the two execution engine instances to detect any transient errors in the datapath that can cause the states of the two execution engine instances to differ from each other. Our JVM recovers from transient errors by rolling back to the state of the last checkpoint. The two execution engine instances and the checkpoint share the objects in the heap to reduce the memory space overhead. Our experiments with seven embedded Java applications show that the average memory overhead due to our transient error tolerant mechanism is 35%, and the average performance overhead is not too much in a two-CPU environment.

6. REFERENCES

[1] J2ME, CLDC 1.0.4 reference implementation. http://wwws.sun.com/software/communitysource/j2me/.

[2] J2ME technology for creating mobile devices (white paper). http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[3] Us mobile devices to 2006: A land of opportunity. Based on an extensive research program conducted by Datamonitor on mobile device markets in the US.

[4] J. F. Aiegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1), 1996.

[5] D. W. Caldwell and D. A. Rennels. A minimalist fault-tolerant microcontroller design for embedded spacecraft computing. *Journal of Supercomputing*, 16(1-2), 2000.

[6] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. Milojicic. JVM susceptibility to memory errors. In *the 1st USENIX Sysmposium on Java Virtual Machine Research and Technology*, 1999.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Analyzing object error behavior in embedded JVM environments. In *the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[8] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1), 1994.

[9] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. *SIGARCH Comput. Archit. News*, 31(2), 2003.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, 2nd Edition*. Addison Wesley Professional, 2000.

[11] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001.

[12] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, 2000.

[13] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1), 1994.

[14] R. N. Williams. A painless guide to crc error detection algorithms. ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt.