Object Duplication for Improving Reliability

G. Chen, G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
e-mail: {guilchen,gchen,kandemir,vijay,mji}@cse.psu.edu

Abstract— Soft errors are becoming a common problem in current systems due to the scaling of technology that results in the use of smaller devices, lower voltages, and power-saving techniques. In this work, we focus on soft errors that can occur in the objects created in heap memory, and investigate techniques for enhancing the immunity to soft errors through various object duplication schemes. The idea is to access the duplicate object when the checksum associated with the primary object indicates an error. We implemented several duplication based schemes and conducted extensive experiments. Our results clearly show that this spectrum of schemes enable us to balance the tradeoffs between error rate and heap space consumption.

I. Introduction

A major reliability concern due to the increasing size of embedded memories is *soft errors*. Soft errors occur when a memory bit flips its value due to external radiation effects, thus corrupting the stored data. The need for reliable memory has become even more acute due to the use of power-savings techniques such as voltage scaling in current embedded systems.

A common approach to handling these memory errors is to use error detection and correction hardware [14, 13]. However, 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, a hardware approach, which increases the overall cost of the system, 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. In comparison, a software scheme can take application specific requirements into account and tune the policy, considering the limited resources in the embedded device.

The focus of this work is on handling soft errors in object-oriented frameworks. We select an embedded Java Virtual Machine (JVM) as our target object-oriented environment, and inject errors into the heap memory that stores the objects in order to investigate techniques for enhancing immunity to soft errors through various object duplication schemes. While a lot of work has been done on the problem of reliable computation at the circuit, architectural, operating system, and application levels [1, 2, 10, 11, 15, 17, 18], our work focuses explicitly on the integrity of objects, and is complementary to model checking and verification based work [7, 12].

The rest of this paper is organized as follows. Section II discusses our error injection model. Section III presents our object duplication schemes, including full duplication, compression-based duplication, and selective duplication schemes. Section IV presents an experimental evaluation of these schemes. Section V concludes the paper with a summary of our major observations.

II. The Error Injection Model

We use Sun's KVM [5] to implement the object duplication-based error protection techniques proposed in this work. KVM is a compact, portable Java Virtual Machine specifically designed for small, resource-constrained devices. KVM uses a handle-free mark-sweep-compact collector. An error management module is added into KVM to store the error information for each object. For every bytecode executed, KVM invokes our error injection function to inject errors into the object instances in the heap. The error injection function scans the heap; every bit in the object instances has a fixed probability of incurring an error. When an object is accessed, we check the error management module to determine whether the accessed part has any error(s) in it. The default value for the error injection probability for our base experiments is 10^{-10}. While we perform experiments with different error injection rates, the rates used in our experiments are generally higher than those with the current technology. The main reason for this is that errors are more likely to happen only when an application executes for long durations of time or when it is executed repeatedly, and we need an accelerated testing environment. It should be noted that accelerated testing is meaningful because there are many embedded Java applications that need to be operational without errors for long durations ranging from several hours (e.g., cell phones) to months (e.g., sensors).

III. Duplication Schemes

A. Motivation for Object Duplication

In this work, unless stated otherwise, we assume that each object is protected using a "checksum-based scheme" (called CHK). In this scheme, each object has a single checksum attached to it. The checksum calculations are performed in a similar fashion to that in [3]. Specifically, each object header is extended with one additional word to store the precomputed checksum. This checksum is checked upon a read request to a field, and updated upon a write request.

The Java applications used in this study are given in Table I. Calc, firstaid, jpeg, and mvideo are taken from the

0-7803-9451-8/06/$20.00

TABLE I
THE JAVA BENCHMARK CODES AND THEIR CHARACTERISTICS.

Benchmark	Description	Execution Cycles	Errors Injected/ Consumed/Detected
auction	ticket auction	467.4	75 / 27 / 25
calc	calculator	338.5	71 / 14 / 14
firstaid	firstaid info	618.5	423 / 127 / 126
jpeg	jpeg viewer	1,052.9	1314 / 329 / 313
image	photo album	1,157.1	430 / 302 / 271
manyballs	bouncing balls	475.2	53 / 13 / 11
mvideo	video player	2,732.6	63 / 44 / 40
pushpuzzle	puzzle game	479.7	72 / 14 / 13

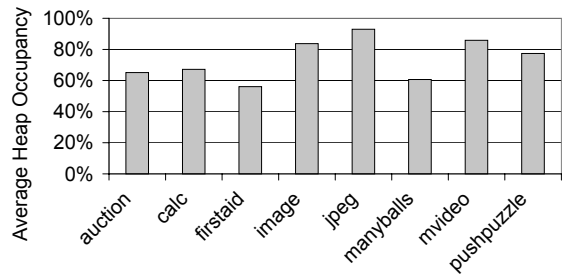

Fig. 1. Average heap occupancy of our applications.

http://www.microjava.com site, and auction, image, manyballs, and pushpuzzle come with the MIDP 1.0.3 reference implementation [8]. The third column gives the execution cycles (in millions) for the "base execution." In this base execution (denoted BASE in the rest of the paper), the objects are not protected. The last column shows statistics on the behavior of the CHK scheme. It gives the total number of errors injected into the memory , the number of errors that have been consumed by the application, and the number of errors detected by CHK. Note that many of the injected errors have not been consumed, meaning that the memory location with the error has not been accessed. We see that although CHK is successful in detecting most of the consumed errors (it detects 93.1% of the errors on the average) , it will not be able to correct any of them. One of our objectives in this work is to correct as many of these errors with as little performance overhead as possible.

We next look at the heap usage statistics of our applications. Figure 1 shows the average heap space used by each application over the time as a fraction of the peak heap space used by that application. For example, the first bar indicates that, on the average, the benchmark auction uses 65.2% of the peak heap space it needs for its objects. One can see from these results that, across all applications, only 74.1% of the peak heap usage is utilized on the average. The remaining heap space can be used for some other purpose. Moreover, the actual maximum heap space allocated for the objects of the application can even be larger than the peak space demanded by an application. That is, in reality, we may have a larger unused heap area that can be exploited for some other purpose.

B. Full Duplication

In this paper, we use this unused heap space for duplicating objects. In this scheme (called DUPL), each time a new object is created, we also create a duplicate object in the heap. Both the primary object and duplicate are protected using checksums. When accessing an object, we first access the primary object. If its checksum indicates no error, we continue with execution as usual. On the other hand, if its checksum indicates an error, we access the duplicate and check its checksum. Note that, under realistic operating conditions, the chances that the checksum of the duplicate also shows an error will be very low. Therefore, we should be able to correct the error in the primary copy most of the time. An important advantage of this scheme is that as long as there is no error, we incur very small performance penalty over CHK. Our performance overheads are mainly due to creation of duplicate objects and updating them on writes. And, when an error occurs, correcting it using one more object access should be acceptable to most operating environments.

The primary and the duplicate objects are allocated in the primary area and duplicate area, respectively. The primary area starts from the lowest address of the heap, and grows toward the higher addresses. The duplicate area, on the other hand, starts from the highest address, and grows toward the lower addresses. When the boundaries of these two areas meet with each other, a mark/compact garbage collector [9] is invoked. To compact the heap, the primary objects are slided toward the lowest address, and the duplicate objects are slided toward the highest address. The header of each primary object contains a pointer to its duplicate, which is called the *forward pointer*. Similarly, each duplicate object contains a pointer to its primary object, which is referred to as the *backward pointer*. During the mark-compact garbage collection, if a primary object is moved, the backward pointer in its duplicate should be updated. Similarly, if a duplicate object is moved, we need to update the forward pointer in its primary copy. Forward-backward pointer pairs allow the primary and duplicate objects to find each other. Further, this mechanism enables us detect or repair the errors in the forward or backward pointers. Note that, though not evaluated in this work, it is possible to have more efficient strategies to connect primary and duplicate objects (e.g., having one object with duplicated fields and two checksums, or placing the duplicate at a fixed distance in memory from the primary object). The reason that we use the pointer-based scheme explained above is that it suits better for more sophisticated duplication strategies, such as the selective scheme as will be discussed later.

Discussion: A full duplication scheme doubles the memory requirement of heap objects, which might be undesirable for an embedded environment. There exist at least two ways of reducing the memory space and/or performance overheads associated with DUPL. First, since the duplicate objects are read only when there is an error in the primary copy, we can compress them so that their heap space occupancy can be reduced. The downside is that when we need to access a compressed duplicate, it first needs to be decompressed before the access can take place. Therefore, there is a tradeoff between performance and heap space saving. The second alternative for reducing the overheads is to use duplication selectively based on object lifetimes. If we are careful in identifying the objects that really need duplicates, we can reduce the overall overhead. The next two sections investigate these two approaches.

C. Compression-Based Full Duplication

Since it is known that, in most Java applications, the objects contain a lot of zero bytes [4], we use a "zero-removal" algorithm to compress the duplicate objects to reduce their heap space requirements. We modified KVM to implement object

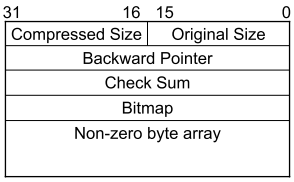

Fig. 2. The format of a compressed duplicate.

compression. Figure 2 shows the format of a compressed duplicate object. The compressed object contains a bitmap and a non-zero byte array. Each bit in the bitmap corresponds to one byte of the object in the uncompressed format. A 0-bit indicates that the corresponding byte is zero and this byte is not stored; a 1-bit indicates that the corresponding byte is stored in the non-zero byte array. The details of the zero-removal compression/decompression algorithm are beyond the scope of this paper.

When an error is detected in the primary object during execution, we check if its compressed duplicate is corrupted. If the duplicate is not corrupted, we correct the error in the primary object by decompressing the duplicate into the address of the primary copy. Otherwise, we terminate the program and report the non-correctable error. When the contents of the primary object are updated, we check the bitmap in the compressed duplicate to determine the number of bytes that are currently used to store the accessed field. If this number is not smaller than the number of non-zero bytes in the updated value, we update the corresponding bitmap bits and non-zero bytes of the accessed field in the compressed duplicate. Otherwise, we need to discard the current duplicate, and create a new one since the new compressed duplicate cannot fit in the space reserved for the old one. It should be noted that, during garbage collection, the reference fields of the primary objects may be updated due to compaction. Therefore, we discard and collect all the duplicates whose primary objects contain reference fields. After the compaction phase, we re-create those discarded duplicates from the contents of their primary objects. This compression-based version of DUPL is referred to as COMPDUPL in the rest of this paper.

D. Selective Duplication

In this strategy, the main goal is to maintain as few duplicates as possible without significantly hurting the error correction rate achieved using full duplication. Recall that both DUPL and COMPDUPL maintain duplicates as long as the primary object is alive. However, since the duplicate is only needed when there is an error in the primary object, we can get rid of the duplicates under certain circumstances.

One example of how a duplicate can be eliminated is based on an analysis of object "drag times". The drag time of an object is the ratio between the time the object spends beyond its last use and the time since its creation to its death [16]. We found that the drag times for auction, calc, firstaid, image, jpeg, manyballs, mvideo, and pushpuzzle are 53%, 45%, 59%, 53%, 82%, 54%, 88%, and 48%, respectively. That is, the objects in our embedded applications spend a large percentage of their lifetimes in the heap beyond their last-use. Our first selective scheme is specifically designed to exploit this observation to reduce the lifetime of duplicates. Another way of cutting the number of duplicates is to consider the lifetime of the objects.

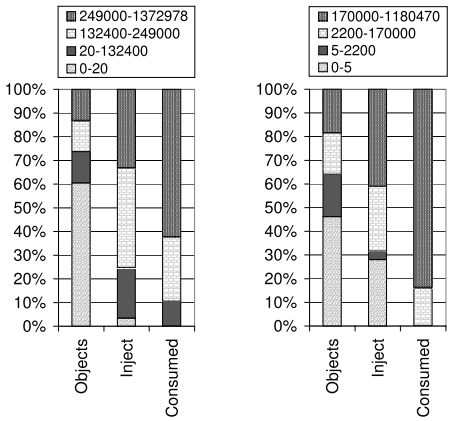

Fig. 3. The breakdown of objects, injected errors, and consumed errors into four life groups. Left: firstaid. Right: pushpuzzle. Each portion of the first bars indicates the percentage of objects that fall into that life group (in terms of a cycle range). Each portion of the second (third) bars represents the percentage of errors injected into (consumed by) objects within that life group.

Figure 3 gives for two applications, firstaid and pushpuzzle, the breakdown of the objects created, of the errors injected, and of the errors consumed into four categories formed based on their lifetimes. We see that most of the objects created are small in size, and in comparison, the injected errors are more uniformly distributed across the different life groups. However, the last bars clearly indicate that the most of the consumed errors are in long-living objects. Consequently, one can expect a protection strategy that pays special attention to long-living objects to be effective in practice. Our second and third selective duplication strategies are designed to take advantage of this observation.

Early Termination of Duplicates: In the first selective scheme, when we create an object, we create its duplicate as usual. However, we also predict the *last-use* of the object. At each invocation of the garbage collector, we now collect not only the unreachable primary objects and their duplicates, but also the duplicates whose primary objects have become last-used. To implement this scheme, each duplicate is augmented with a time-stamp that records its allocation time. At each garbage collection, the duplicates that are older than a threshold are collected. The success of this strategy critically depends on the accuracy of the last-use prediction (i.e., the threshold value used). To determine good prediction values, we plotted the CDF (cumulative distribution function) for object last-uses. Each (x, y) point on the curve of a specific application in Figure 4 indicates that the last-use of $y\%$ of the total object words occurs within the first x cycles after its creation. From this graph, one can determine good estimates to use for predicting last-uses. For example, we see from the calc's curve that, if we create the duplicate at the same time when the primary object is created and then discard the duplicate about 3,000 cycles after the creation time, 90% of the total object words will be beyond their last-use point when their duplicates are discarded. Beyond its last-use point, an object will not be accessed by the application any more. Therefore, the errors occurring beyond the last-use point will not affect the correctness of the application. The downside of this scheme is that the objects that have not reached their last-uses will be vulnerable between their predicted last-uses and their actual last-uses. In addition, if the predicted last-use is longer than the actual last-use of an object, we increase heap occupancy unnecessarily. Therefore, an accurate last-use prediction is very important.

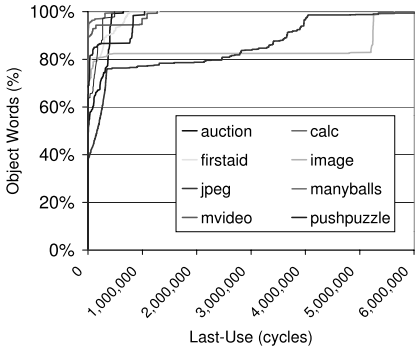

Fig. 4. CDF for object last-uses.

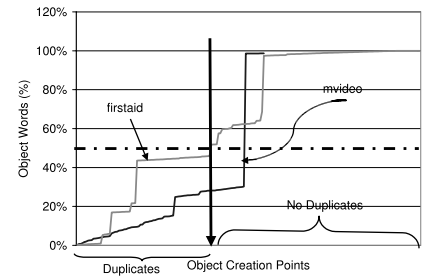

Fig. 5. CDF for object sizes created by each object creation point for firstaid and mvideo. The figure also shows how the object creation points are marked for firstaid.

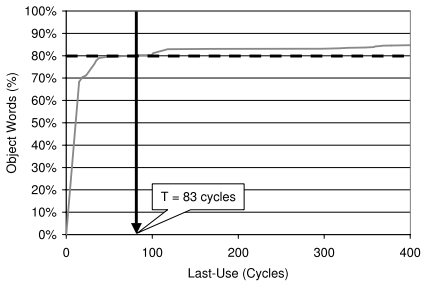

Fig. 6. CDF for last-uses of the objects in mvideo.

Allocation Site Based Selective Duplication: Our second selective scheme is based on profiling. We collected information on the number of accesses to the objects created by each object creation point for our applications. We observed that, for many object creation points, the objects created are not accessed very frequently. Consequently, one can choose not to provide duplicates for such objects. Figure 5 shows how we select the creation points for which we want to provide duplicates. The x-axis represents the object creation points sorted (from left to right) according to non-increasing number of accesses to the objects created by each point. The y-axis shows CDF for the sizes of the total objects created by each object creation point. For example, if we want to provide duplicates for only 50% of the frequently accessed objects, we need to draw a horizontal line from the y-axis and find the corresponding point on the x-axis. All the object creation points on the right of this point are then annotated. These annotations are recognized and handled by the JVM, and no duplicate is created for the objects allocated by these points.

Lazy Duplication: Our last selective scheme, referred to as DELAYED, defers the creation of the duplicate to a point, where we expect the object to be long-living and frequently-accessed once it reaches that point. In other words, this scheme is lazy in creating object duplicates. It is implemented as follows. Each object has a time-stamp, indicating its creation time. At each access to the object, we compare the current time against the time-stamp. If the difference between them is larger than a threshold (T), we create a duplicate for the object. This selective strategy does not create duplicates for the objects whose last accesses are shorter than T. A critical issue here is how to determine a suitable value for T. Note that, if we are late in creating the duplicate, we can increase the number of non-correctable errors since the object is vulnerable until a duplicate is created for it. On the other hand, if we create the duplicate too early, there will be very little improvement in heap space consumption over DUPL. To determine good threshold values, one can use the CDF curves presented in Figure 4. For example, Figure 6 zooms in the initial portion of the curve for mvideo. If we want to avoid the duplicates for 80% of the total allocated object words, we draw a horizontal line, determine the corresponding values on the x-axis, and use that value as T.

IV. Experimental Results

In our experimental evaluation, we collect three types of statistics:

• *Heap space results:* These results indicate the memory space overhead due to object duplication. We are interested in two metrics: (1) the total amount of heap space allocated during execution, and (2) the variance in heap occupancy during the course of execution. We define "heap occupancy" in this context as the total size of the "live application objects" in the heap.

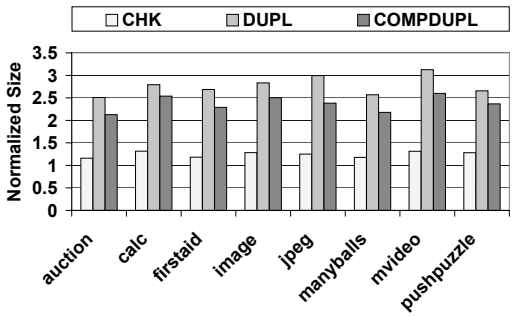

Fig. 7. Allocated heap space during the entire execution.

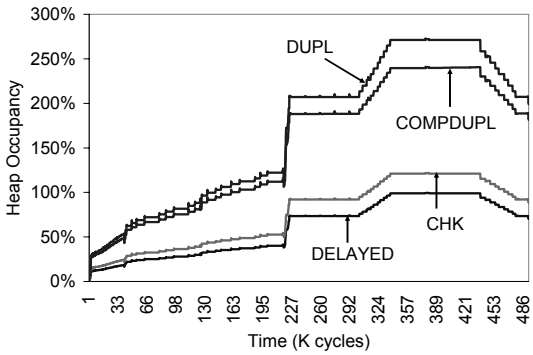

Fig. 8. Heap space occupancy by the application objects during execution of manyballs. The y-axis is normalized with respect to the peak heap space required by BASE.

• *Error resilience results:* These results are collected by injecting errors into the heap memory and instrumenting the application code to collect error statistics. The main metric that we are interested in is the "non-correctable error rate", which is the percentage of soft errors that are detected by checksums but could not be corrected by a given protection scheme.

• *Performance results:* These results correspond to execution cycles, and are obtained using an enhanced version of the Shade tool-set [6].

A. Heap Space Results

The first two bars for each application in Figure 7 give the increase in the total size of the heap space allocated with CHK

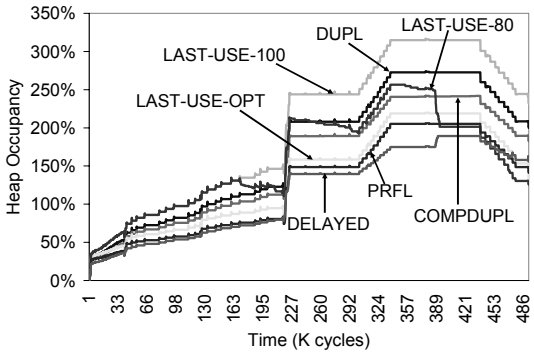

Fig. 9. Heap space occupancy for the application objects during the execution of manyballs for selective schemes.

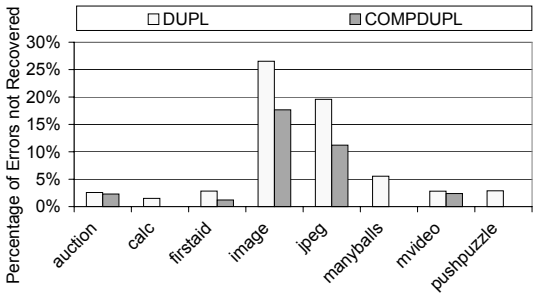

Fig. 10. Non-correctable error rates for the different schemes.

and DUPL during the entire execution. One can see that, while the allocated memory space increase due to CHK is within reasonable limits (around 24% increase over the base execution), DUPL increases the size of the allocated heap space by a factor of 2.77 on the average. We also see that the value for COMPDUPL is lower than the corresponding value for DUPL.

To see how the heap space requirements change during execution, we give in Figure 8 the variance in the heap space occupancy (y-axis) over the time (x-axis) with the different schemes for the manyballs benchmark. Recall that the peak point of a heap occupancy curve gives the minimum heap space required for the user objects. In a multiprogrammed environment, the entire shape of the curve can also be important as memory space saved at any point can be reused by some other application. When we look at the curves for CHK and DUPL, we see that at any given time DUPL occupies much more heap space than CHK. Also, its peak heap occupancy is 124% higher than that of the CHK scheme. We also observe that while object compression (COMPDUPL) brings some heap space savings, the results are not as good as someone might want. The main reason for this is the fact that most of the objects in our embedded applications are smaller than 16 bytes, which makes object compression less effective.

The heap occupancy of the selective duplication schemes is plotted in Figure 9 for manyballs. The curves marked as LAST-USE-OPT, LAST-USE-100, and LAST-USE-80 represents the selective schemes based on early termination of duplicates. LAST-USE-OPT represents an optimal version where we detect the last-uses of the objects by looking at the object traces we gleaned. In other words, in this post-processing based approach, we have a prediction accuracy of 100%, and we get rid of the useless duplicates without any delay. As a result, we do not incur any extra non-correctable errors over DUPL. In comparison, the curves marked as LAST-USE-100 and LAST-USE-80 show the results from our actual implementation. The difference between them is that, in the first one we predict, for each application, the last-use in such a way that it includes the last-uses of 100% of the total object words. In other words, we use the longest last-use time of all objects. In the LAST-USE-80 version, we want to cover the last-uses of 80% of the total object words. We see from Figure 9 that LAST-USE-OPT performs very well, and its heap occupancy is lower than that of the COMPDUPL scheme. Similarly, LAST-USE-80 also exhibits a good heap occupancy. The main reason for this is that it eliminates the duplicate objects aggressively. In contrast, LAST-USE-100 does not perform well from the heap occupancy perspective as it waits too much for removing the duplicates. It should be mentioned that all these three selective versions incur an additional space overhead due to the time-stamp maintained. The line LAST-USE-100 is higher than DUPL due to such overheads. The curve marked PRFL in Figure 9 shows the heap occupancy of the allocation site based selective scheme. In this particular execution, we created duplicates for 60% of the frequently used objects. We observe from these results that the heap occupancy of this scheme is better than that of the first selective strategy in most of the cases. However, this behavior can change if we annotate the object creation points more or less aggressively We also observe from Figure 9 that the lazy duplication scheme, marked as DELAYED, generates better behavior than COMPDUPL.

B. Error Resilience Results

Having looked at the heap space occupancy, we next consider the error resilience of our duplication based schemes. The graph in Figure 10 shows the non-correctable error rate for the different schemes. We see that DUPL corrects more than 96% of the errors in five of our eight benchmarks (auction, calc, firstaid, mvideo, and pushpuzzle). It is not very successful with the image and jpeg benchmarks, mainly due to large number of errors consumed in these applications (see Table I). While the success of DUPL in correcting errors is significant when one considers the entire benchmark suite, there is still a large number of errors not corrected. One reason for this is the high error-injection rate we used (10^{-10}). When error-injection rate increases, the chance that both the original object and its duplicate being injected with errors will increase. To see how DUPL would behave under lower error rates, we also performed experiments with error rates 10^{-11} and 10^{-12}. The results given in Figure 11 indicate that the DUPL scheme is very successful in correcting errors with these rates. More specifically, with an error rate of 10^{-12}, it corrects all the errors detected by checksums. We also observe in Figure 10 that COMPDUPL performs better than DUPL. Specifically, it reduces the average non-correctable error rate from 8.2% (DUPL) to 4.3%. This is because of the reduction in the heap space allocated to the duplicates in this approach.

The non-correctable error rates for the three selective duplication schemes are shown in Figure 12. We see that the error correction behaviors of LAST-USE-OPT and LAST-USE-100 are the same as that of DUPL (given in Figure 10) since they destroy a duplicate only when they are certain that the primary object has reached its last-use. On the other hand, LAST-USE-80 incurs extra non-correctable errors over the DUPL scheme as it eliminates some duplicates while the primary copies are still alive. We also see that, as compared to DUPL, PRFL incurs significantly more non-correctable errors in some appli-

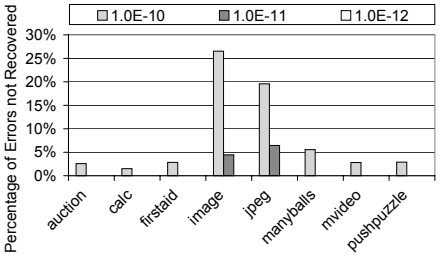

Fig. 11. Non-correctable error rates for DUPL with different error injection rates. Note that, with the last error rate, DUPL corrects all the errors.

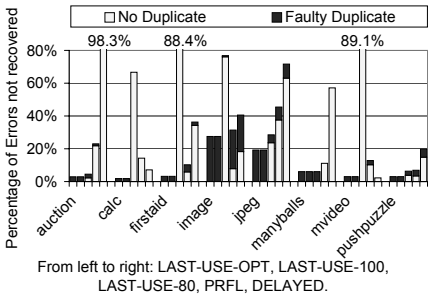

Fig. 12. Non-correctable error rates for the different selective schemes. Each bar is broken to show whether the non-correctable error is due to lack of the duplicate or due to faulty duplicate.

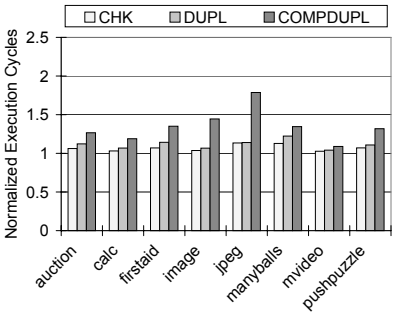

Fig. 13. Normalized execution cycles for the different schemes.

cations, since we create duplicates only for 60% of the frequently used objects. The error resilience can be improved if we increase the percentage of objects that we create duplicates. Finally, the non-correctable error rates for DELAYED is not excessive in some applications.

Overall, with the particular experimental parameters, used above, LAST-USE-OPT, LAST-USE-100, LAST-USE-80, PRFL, and DELAYED reduced the maximum heap occupancy of DUPL by 22.0%, -14.0%, 18.4%, 17.4%, 23.1%. The non-correctable error rates for these five schemes are 8.2%, 8.2%, 50.2%, 19.1%, and 41.6% in that order.

C. Performance Results

The bar-graph in Figure 13 gives the normalized execution cycles for the different schemes. Note that these results include the extra GC time due to duplication. We see that the performance penalty incurred by DUPL is not too high. Specifically, it increases the execution cycles of BASE by 11.3%, and 6.9% of this comes from the checksum overhead itself. This relatively small increase in execution cycles over CHK can be explained as follows. The only time we spend extra cycles with DUPL is when we create an object or write to an already existing object. Since both of these events are very rare as compared to object reads, DUPL does not bring an excessive performance overhead over CHK. Specifically, the number of object writes (creations) is less than 10% (1%) of the number of object reads in our applications. COMPDUPL increases the execution cycles of DUPL by around 20% across all applications. This is because of the compressions and decompressions that need to be performed during execution. While the decompression needs to be performed only when there is an error, the compression is done whenever an object creation or an object write takes place. Overall, the compression-based duplication helps reduce the non-correctable error rate and heap occupancy, but increases execution cycles of DUPL. Therefore, it is more suitable for embedded environments where performance degradation can be tolerated. Finally, we found that the performance of all the selective schemes we implemented is very close to that of DUPL since they eliminate only some writes for which they have already destroyed the duplicate.

V. Concluding Remarks

In this paper, we demonstrated how duplication can improve the data integrity of objects by recovering a significant percentage of the errors detected by a checksum-based scheme. Our baseline full duplication based scheme recovered 91.8% of the errors at the expense of 11.6% degradation in performance. Compressing the duplicates brought up the error coverage to 95.7% and reduced average heap occupancy of the full duplication-based scheme by 18.6%. However, we found that it also increased the execution cycles significantly. We also presented results from three different implementations based on selective object duplication. Using the three selective schemes proposed in this paper, we can tradeoff different requirements of the application and help the designer to determine the best operating point considering both maximum heap space consumption and error rate.

Acknowledgements

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

References

[1] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications," in *Proc. DSN'00*.

[2] C. Chen and A. K. Somani, "Fault containment in cache memories for TMR redundant processor systems," *IEEE Transactions on Computers*, 48(4):386–397, March 1999.

[3] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. Lie, D. Mannaru, A. Riska, and D. Milojicic, "JVM susceptibility to memory errors," in *Proc. JVM'01*.

[4] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko, "Heap compression for memory-constrained Java environments," in *Proc. OOPSLA'03*.

[5] "CLDC and the K virtual machine (KVM)," http://java.sun.com/products/cldc/.

[6] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling," in *Proc. ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, 1994.

[7] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. OOPSLA'03*.

[8] "J2ME Mobile Information Device Profile," http://java.sun.com/j2me/.

[9] R. Jones and R. D. Lins, *Garbage Collection Algorithm for Automatic Dynamic Memory Management*, John Wiley & Sons, 1999.

[10] W. Kao, R. K. Iyer, and D. Tang, "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Transactions on Software Engineering*, SE-19(11):1105–1118, November 1993.

[11] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. Micro'03*.

[12] C. S. Pasareanu, M. B. Dwyer, and W. Visser, "Finding feasible counter-examples when model checking Java programs," in *Proc. TACAS'01*.

[13] R. Phelan, "Addressing soft errors in ARM core-based designs," *White Paper*, ARM Limited, 2003.

[14] C. Pyyhtia, "Quality issues facing embedded memory," in *Proc. Sophia Antipolis Conference on Micro Electronics*, 2002.

[15] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proc. ISCA'00.*

[16] R. Shaham, E. K. Kolodner, and S. Sagiv, "Heap profiling for space-efficient Java," in *Proc. PLDI'01.*

[17] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Transactions on Reliablity*, 49(3):273–284, 2000.

[18] C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design," in *Proc. DSN'00.*