Analyzing Heap Error Behavior in Embedded JVM Environments

G. Chen, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
{guilchen,kandemir,vijay,anand,mji}@cse.psu.edu

ABSTRACT

Recent studies have shown that transient hardware errors caused by external factors such as alpha particles and cosmic ray strikes can be responsible for a large percentage of system down-time. Denser processing technologies, increasing clock speeds, and low supply voltages used in embedded systems can worsen this problem. In many embedded environments, one may not want to provision extensive error protection in hardware because of (i) form-factor or power consumption limitations, and/or (ii) to keep costs low. Also, the mismatch between the hardware protection granularity and the field access granularity can lead to false alarms and error cancellations. Consequently, software-based approaches to identify and possibly rectify these errors seem to be promising. Towards this goal, this paper specifically looks to enhance the software's ability to detect heap memory errors in a Java-based embedded system.

Using several embedded Java applications, this paper first studies the tradeoffs between reliability, performance, and memory space overhead for two schemes that perform error checks at object and field granularities. We also study the impact of object characteristics (e.g., lifetime, re-use intervals, access frequency, etc.) on error propagation. Considering the pros and cons of these two schemes, we then investigate two hybrid strategies that attempt to strike a balance between memory space and performance overheads and reliability. Our experimental results clearly show that the granularity of error protection and its frequency can significantly impact static/dynamic overheads and error detection ability.

Categories and Subject Descriptors: C.4 [Computer Systems Organization]: Performance of Systems — *fault tolerance*

General Terms: Experimentation, Performance, Reliability

Keywords: Softerrors, JVM

1. INTRODUCTION

The continuing hardware innovations and the deployment of computer systems in several new domains are making it imperative to consider the possibility of hardware errors in system design. On the hardware front, the ability to pack billions of transistors into a small area and the employment of lower supply voltages to reduce power consumption are making combinational circuits and storage cells more susceptible to soft errors due to cosmic ray strikes [20, 21, 5, 17]. At the same time, computing devices are being used in a variety of embedded environments to monitor and control physical world phenomena — in space vehicles, avionics, automobiles and nuclear reactors to name a few domains — making their operational integrity very critical. While soft errors have been known to be problematic in harsh environments such as space and in nuclear reactors, they are also becoming a common problem in current systems operated under normal conditions due to the scaling of technology that results in the use of smaller devices and lower voltages. Furthermore, it has been shown that the use of low-power operating modes in some of these devices accentuates the soft error problem [5].

Error detection, and possible recovery, in both the hardware and software layers is becoming essential. At the hardware end, circuits and architectural techniques for spatial (e.g., duplication of data, ECC, parity, etc.) and temporal (e.g., re-execution of operations, NMR [18, 15], etc.) redundancy can reduce the impact that these transient errors could have on execution. However, in many embedded environments, one may want to consider error protection in software as well, mainly due to following reasons:

- Hardware-based error protection can increase form-factor and overall system cost.
- Hardware-based error protection may be overly expensive (and in some cases overkill) for providing the required reliability [2, 9]. In particular, in an embedded system that runs multiple applications, some applications may require extensive protection, whereas some other applications may not require any protection at all.
- The mismatch between the granularity of hardware-based error protection and the granularity of data accesses can lead to missing some errors or giving false errors in some cases.
- Even if one provisions hardware safe-guards, some errors could still propagate to the software since the hardware is typically customized to handle specific errors.

Consequently, the use of a software-based approach to identify and possibly rectify these errors is promising. Towards this goal, this paper specifically looks to enhance the software's ability to detect heap memory errors in a Java-based embedded system.

Java is rapidly becoming the vehicle of choice for software development in many embedded environments [14, 10, 16, 19]. We are witnessing micro-editions of Java (e.g. [3]) being used for developing embedded software. Of the different components (code, stack, heap) of a Java program, the heap is typically the dominant space consumer, making it more susceptible to errors, and is the target of study in this paper. While error protection schemes can also be employed for the rest of the memory such as the code portion, exploring this is beyond the scope of this paper.

When considering software protection against soft errors, we are faced with a wide range of design choices in implementing error detection for heap objects, each presenting a tradeoff between the error coverage (reliability), performance overheads, and space requirements. At one end, we can opt for provisioning detection at the level of individual data fields of an object, which we refer to as field-level error detection. Without loss of generality, we assume a checksum [18] to implement such error detection in this paper, and whenever a field is accessed, we compute the resulting checksum and compare it with what has been pre-computed to ensure integrity of the value accessed. On a larger granularity (as was the case in an earlier related work [1]), one could associate a checksum with

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

the entire object (computed across all its fields), which we refer to as object-level error detection. The advantages of field-level detection are in (i) possibly lower false alarms, since errors that occur in other fields of an object do not affect the integrity of the field that is being accessed (while an error in some other field of that object can cause a checksum discrepancy in object-level detection), and (ii) lower performance overheads since we do not need to examine all the fields of the object to compute the checksum when accessing an individual field. The downside of the field-level detection, compared to maintaining the checksum at the object granularity, is the higher memory space requirements that can become a severe limitation in memory-constrained embedded environments. It could also potentially cause performance penalties in accessing the memory hierarchy.

While a lot of work has been done on the problem of reliable computation at the circuit, architectural, operating system, and application levels [18, 6, 15, 13, 8, 5, 20, 2, 11], our work focuses explicitly on heap objects in Java. An earlier study [1] on enhancing software awareness of hardware errors in the context of Java, only considered object-level protection. To our knowledge, there has been no prior in-depth investigation into the pros and cons of the granularity of provisioning protection for heap objects in Java programs.

Using several embedded Java applications, this paper makes the following major contributions:

• We first illustrate the impact of hardware errors on the software execution of these Java programs. We demonstrate that the field-level and object-level checksumming strategies have very different error detection and propagation characteristics.

• We conduct a detailed evaluation of the tradeoffs between reliability, performance and memory space overheads for these two granularities of error detection. We also study the impact of object characteristics (e.g., lifetime, re-use intervals, access frequency, etc.) on error propagation, and the pros and cons of these two schemes.

• Our experimental results show that there is a significant difference between these two schemes in terms of their reliability, performance and memory space overheads, suggesting that one may be able to opt for hybrid schemes between these two extremes. We present such a hybrid scheme, where checksums are associated with groups of fields of an object, and evaluate its potential. We also propose and evaluate an alternate strategy where checksum computations are performed selectively, i.e., not at every object access.

The rest of this paper is organized as follows. The next section presents our experimental setup and benchmarks. Section 3 discusses the error detection schemes evaluated in this work, and gives an error classification. Section 4 presents detailed experimental results. Section 5 concludes the paper.

2. EXPERIMENTAL SETUP

We implement our error protection mechanisms and evaluate them using Sun's KVM [3]. KVM is a miniaturized JVM for handheld devices and suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers. We included instrumentation code to the KVM for collecting error statistics and gathered this information through actual execution of the applications on a SPARC workstation. In order to collect performance statistics of these applications, the instrumented code for gathering error statistics was disabled, and the KVM was executed using Shade [4], an instruction-set simulator and custom trace generator. The architecture simulated in this work has 32KB, two-way set-associative data and instruction caches (both with a block size of 32 bytes). The cache access latency is 1 cycle and the memory access latency (upon a cache miss) is 80 cycles. Unless stated otherwise, we assume a 1MB default heap size.

Table 1 lists the Java benchmark codes used in this study. The first three columns give, respectively, the name, the source, and a brief description of each benchmark. One can see from this table that our benchmark suite includes both utility programs and game programs. The fourth column of this table shows the number of (dynamic) object references (accesses) during the execution. The fifth column gives the average object size in bytes. The next two columns give, respectively, the number of misses and miss rate for the instruction cache. The next two columns present the same information for the data cache. Finally, the last column gives the execution cycles. All these values are for the *original* benchmarks, i.e., without any error protection.

One can see from Table 1 that most of the objects are small in size. More specifically, except for two benchmarks (auction and firstaid), the average object size is less than 20 bytes. Note that this small number puts a limit on the performance degradation that could be incurred by our object-level error detection scheme. At the same time, this small number means that the storage overheads of maintaining the checksums can potentially be significant.

An important component of this work is the error injection model used. An error management module is added into KVM to store the error information for each object. For every bytecode executed, KVM invokes the 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, including the checksum field, has a fixed probability of incurring an error. We can think of this as flipping a coin for every bit, which has a fixed probability of having an error. Once a bit incurs an error, this error's address is recorded by the error injection module. The default values for the error injection probability for our base experiments was 10^{-9}. It must be emphasized that the error injection rates used in our experiments are much more aggressive than in current technologies to reflect future technologies and the higher number of errors that will occur in longer running applications. Note that many embedded applications such as those employed in ATMs, industrial microcontrollers and automobiles are long running and reliability-critical. When an object is accessed by some bytecode or garbage collection operation, we check the error management module to determine whether the accessed part has any error(s) in it. For every error in the accessed part, we classify it (as will be discussed shortly in detail), add it to our error consumption record, and finally clear the error information in the error management module. It must be observed that in collecting error statistics, after clearing an error, we continue with the execution. The methodology for correcting the error itself is orthogonal to our focus.

3. ERROR DETECTION MECHANISMS AND ERROR CLASSIFICATION

In this work, we explore a continuum of error protection schemes based on checksums. The schemes we study vary from CSOBJ (object-level error detection), which associates one checksum word with each object to CSFLD (field-level error detection), which associates a checksum with each field of each object. Apart from these two extremes we also study two other schemes that attempt to strike a balance between these two from the memory space overhead, performance, and reliability perspectives.

In KVM, the heap contains various types of objects, including free blocks, Java object instances, arrays, internal VM objects such as method tables, execution stacks, threads, etc. Each object is preceded with an object header that contains control information about the object. In our implementation we consider only the Java object instances and internal VM objects. However, it is conceivable to extend our strategy to other heap objects as well.

3.1 CSOBJ: Object-Level Checksum

In this scheme, each object has a single checksum attached to it. The checksum calculations are performed in a similar fashion to that in [1]. Specifically, each object header is extended with one additional word to store the precomputed checksum. Upon a read request to any field (getfield), a new checksum is calculated by XORing all the words of the different fields of the object, and compared to the checksum stored in the object header.[1] If these two checksums match, this indicates that the object is valid and the read operation proceeds as usual. If they do not match, however,

[1]Note that this checksum-based scheme is more powerful than the conventional parity-based mechanism.

Table 1: The important characteristics of the Java benchmark codes used in this study.

Benchmark	Source	Brief Description	Object References	Average Object Size	Instruction Cache Misses	Instruction Cache Miss Rate	Data Cache Misses	Data Cache Miss Rate	Execution Cycles
auction	MIDP 1.0.3	ticket auction program	64653	26.6	2233957	0.0116	3274116	0.0900	467463899
calc	http://www.microjava.com	calculator program	46646	12.9	1669555	0.0118	2263703	0.0839	338521526
firstaid	http://www.microjava.com	firstaid information	129844	22.2	2884707	0.0109	4627566	0.0890	618526342
image	MIDP 1.0.3	photo album	283117	14.3	6910756	0.0126	5261979	0.0331	1157109464
jpeg	http://www.microjava.com	jpeg image viewer	800388	15.7	6684924	0.0133	4323291	0.0293	1052995589
manyballs	MIDP 1.0.3	balls bouncing on the screen	68340	15.1	2196302	0.0113	3420055	0.0891	475180873
mvideo	http://www.microjava.com	mobile streaming video player	165228	12.6	15447693	0.0108	10598184	0.0327	2732635831
pushpuzzle	MIDP 1.0.3	puzzle game program	131127	14.7	2272132	0.0116	3050549	0.0742	479712863

this means that the object is not valid, and an appropriate recovery action, if any, is taken. On the other hand, if an object is accessed by a native function, a bit in its header is set to represent an invalid checksum (such an object is termed as "checksum-invalid" as opposed the other types of objects that are called "checksum-valid" objects); that is, our current implementation does not provide error detection across native functions. Upon a write operation on an object (putfield), if it is checksum-valid, all the words in the object are XORed together, and the result is stored into the checksum field in the object header. In contrast to [1] that has a checksum field of 1 byte and one extra byte as a tag to indicate whether an object is checksum-valid or not, our scheme uses one word for checksum, since objects in the heap are word-aligned in the KVM implementation. Consequently, one extra word is required for each object and XORing operations are done at a word granularity rather than a byte granularity as in [1]. The additional bits for the checksum also serve to provide better error detection abilities.

3.2 CSFLD: Field-Level Checksum

In this scheme, each field in an object has a checksum associated with it. Since different objects can have different number of fields, it is not possible to place the checksum in the object header (which is a fixed size entity). Therefore, in our implementation, we stored these checksums as (user-transparent) fields in the objects. Compared to storing a checksum in the object header, this method can be expected to incur more space overheads, but as we will discuss shortly, it also provides more accurate error detection. We also use a bit in the object header to store the checksum-validity tag to tell whether an object has been accessed by any native function. The computation of the checksum for a checksum-valid object is performed as follows. Upon a read operation, all the words in the field being read are XORed together, and the result is compared with the stored checksum of that field. In comparison, on a write operation, all the words in the field being written are XORed together, and the result is stored in the corresponding checksum field.

3.3 Qualitative Comparison

It should be noted that CSOBJ and CSFLD behave differently as far as errors are concerned. The CSOBJ scheme performs an object-wide checksum comparison at each field access. If the checksum is wrong, this means that one or more fields in the object (not necessarily the one that has just been accessed) are corrupted. That is, some of the errors signaled by the CSOBJ scheme can be "false alarms." In this sense, it is less accurate than the CSFLD scheme, which always identifies the corrupted field accurately when an error occurs. However, from a memory space overhead perspective, CSOBJ is clearly more efficient than CSFLD, which has a separate checksum for each field of every object. Finally, when one considers the performance angle, it is difficult to favor one of the schemes over the other. This is because both the schemes come with their performance overheads. Specifically, the CSOBJ scheme incurs reading of the entire object each time a field is accessed. This in turn can lead to data cache pollution if the extra fields read (i.e., the fields other than the one being accessed) are not accessed later when they are still in the cache. The CSFLD scheme, on the other hand, stores a checksum along with each field. Many of these checksum fields can be brought into the data cache when normal field accesses occur, thereby polluting the cache. This space overhead can also impact heap behavior. In addition to these, both the schemes incur extra instruction cache accesses and datapath operations due to checksum computations. In fact, we expect the number of extra datapath operations of CSOBJ to be much larger than that of CSFLD, and this can favor CSFLD over CSOBJ from a performance angle. Therefore, selecting CSOBJ or CSFLD as the error protection mechanism entails performance, space, and reliability implications.

3.4 Other Schemes

While CSOBJ and CSFLD represent two extreme solutions to the error protection problem, one may also consider alternate schemes. In fact, there are two ways of getting the tradeoffs between these two extremes:

- CSOBJ is efficient as far as memory space overhead is concerned but inefficient from a performance overhead angle. Therefore, one way of optimizing its behavior is by reducing the frequency of checksumming. One such scheme that we investigate in this paper is based on the idea that, if the same object is accessed twice (or more) in a very short period of time, we can improve performance by skipping the checksum computation in the second (and maybe successive) access(es). The rationale behind this idea is that the probability of having an error between the two successive accesses (within a short period of time) is very low. That is, by skipping the checksum computation (and comparison) for the latter access, we can save execution cycles at the expense of a slightly weaker error resilience. At the same time, we have to bound the temporal gap between the accesses since not doing so can result in missing several errors that could otherwise be caught by the CSOBJ scheme. One particular implementation studied in this work is referred to as CSTHR — i.e., CSOBJ with a threshold. It is the same as the CSOBJ scheme except that it also accommodates the optimization described above. In our implementation, each object has one more field to keep record of the time (in number of bytecode executed) when the last checksum operation is performed on this object. Upon a read operation, an object will be checked only if the difference between current time and the stored time is larger than a predefined "threshold" value.
- CSFLD is efficient from a performance overhead perspective but inefficient from a heap space angle. So, instead of applying it to each individual field, one can apply it to a group of fields at a time. Based on this observation, our next strategy, referred to as CSGRP, tries to strike a balance between CSOBJ and CSFLD by associating a checksum with a group of fields. Specifically, the fields in a given object are divided into groups, and whenever there is an access to a field in a particular group, the checksum calculation is performed over all the fields in that group. Clearly, this scheme stands between the CSFLD and CSOBJ schemes in terms of memory space overhead and performance. It should be noted that there might be different ways of implementing this scheme. In one implementation, we can fix the number of fields per group for all objects. In another implementation, we can fix the number of groups per object. Yet another option would be employing profile data to cluster the fields according to their access locality.

3.5 Error Classification

Table 2 lists all possible classifications for an injected error depending on how it is consumed. In this context, an error is said to be *consumed* if a read operation is performed in the field corrupted by this error. Also, an error is said to be *cleared* if a write operation is performed in the field corrupted by this error. NC means the error in question is neither consumed nor cleared. In a sense, this class

Table 2: The error classification adopted in this paper. These classes are disjoint and together they cover all cases.

Class	Description
NC	not consumed/cleared
CA	consumed at the accessed field or at the checksum
COF	consumed at a field different from the accessed one; the field is accessed later (not meaningful for CSFLD)
COT	consumed at a field different from the accessed one; the field is not accessed later (not meaningful for CSFLD)
NM	consumed but not detected due to multi-bit errors
NN	consumed but not detected because of the checksum-invalid tag; this occurs due to native function accesses
NP	consumed at a different field than the written field when computing checksum (not meaningful for CSFLD)
CL	cleared by a write operation

represents harmless errors that do not affect the behavior of the application being executed. If a data field is accessed and an injected error in this field is detected by the checksum scheme employed, the error is classified as CA. Note that this class also includes the errors that occur in the checksum fields. If an error in a field A of an object is detected during the checksum operation for a read access to another field B, this error is classified as COF or COT. Specifically, if there is no bytecode instruction or garbage collection operation that accesses field A until the end of program, this error belongs to the COT class. Otherwise, it belongs to the COF class. Note that COF and COT classes are not meaningful for the field-level detection scheme — CSFLD — since in this scheme an error can be detected only through an access to that field. It should be observed that there are several cases where our checksum-based schemes cannot detect the error injected. For example, some of the cases of two error occurrences in the same position of different words cannot be detected by our checksum-based approach. These types of errors are referred to as NM (multi-bit errors) in this paper. If an object is accessed by a native function, it becomes checksum-invalid, and any error consumed in this object cannot be detected. This error class is termed as NN. In CSOBJ, when there is a write operation on a field, the other fields are read to calculate checksum. Since there is no checksum-checking operation for a field-write operation, errors in the other fields are consumed but not detected. These errors belong to the NP class. And finally, if a field containing an error is overwritten in a write operation, the errors pending on it are counted as belonging to the CL class. It should be observed that these classes are disjoint, and together, they cover the entire spectrum. That is, an injected error is guaranteed to belong to one of these classes.

4. EXPERIMENTAL EVALUATION

4.1 Results with CSOBJ and CSFLD

The graph on the left of Figure 1 shows the average increase in the cumulative sizes of the allocated objects due to the two different error protection schemes: CSOBJ and CSFLD. The CSOBJ technique incurs around 20-25% additional space overhead due to the checksum. Note that the checksum occupies four bytes and the object average size ranges from 12.6 bytes to 26.6 bytes (see Table 1). In contrast, the CSFLD incurs more space overhead, almost twice as much as an object with no checksums. This is due to a checksum being associated with each field.

It is also important to study how this space overhead incurred for the objects impacts the overall heap requirements. It must be observed that the heap contains not only the protected objects but also methods, stack frames, and array objects that are left unchanged in our implementation. The results of our experimentation with the *smallest* heap size required to complete the applications successfully (i.e., without giving an "insufficient memory error") when using CSOBJ and CSFLD are shown on the right part of Figure 1. It can observed from this graph that the overall impact on the heap size requirements is limited to be less than 7% for all the benchmark codes in our experimental suite when using both the schemes. That is, the heap space impact of including checksums for object instances is not excessively large.

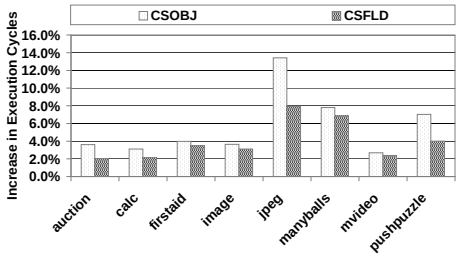

Figure 2: Increase in execution cycles as compared to using no checksums. The average execution cycle increases due to CSOBJ and CSFLD are 5.66% and 3.99%, respectively.

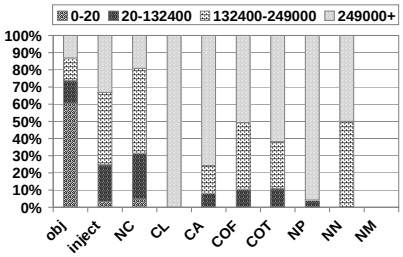

Figure 4: Error consumption behavior of firstaid based on the object lifetimes measured in the number of bytecode executions.

Our next focus in on the increase in the original execution cycles when these two error detection mechanisms are used. The graph in Figure 2 gives the percentage increase over the case when we have no error protection. One can make several observations from this graph. First, in five of our benchmarks, the increase in execution cycles are lower than 4% for both the schemes. The second observation is that the CSFLD incurs less performance overhead as compared to the CSOBJ scheme. Specifically, the average execution cycle increases due to CSOBJ and CSFLD are 5.66% and 3.99%, respectively. This is mainly because the CSOBJ scheme accesses the entire object whenever a field is accessed, and this increases the number of instructions executed.

After having presented heap space overhead and performance results, we next focus on how the errors are consumed with the different error detection strategies. Figure 3 (left side) shows the distribution of error consumptions when using CSOBJ. It can be observed that a significant portion of the errors injected do not influence the functionality of the JVM (that is, they end up being in the NC class). The NC portion increases even further for CSFLD technique as can be seen on the right side of Figure 3. In fact, in the CSFLD case, some fields may remain untouched even if the object itself is accessed in future after an error is injected. In other words, the lifetime of an individual field might be shorter than the lifetime of the object which it belongs to. This increased portion of NC in the CSFLD scheme translates to unnecessary error indications (and consequent error recovery if implemented) in the case of CSOBJ due to the COF error classification. Specifically, COF involves cases where the injected error occurs in another field of an object that is never accessed in the program subsequently, but influences the checksum correctness when accessing another field of the same object. As a result, the CSFLD technique has a significant advantage over CSOBJ in avoiding "false alarms." These false alarms lead CSOBJ to flag errors twice as much as CSFLD (note that CA+COF+COT in the CSOBJ scheme is twice as much as CA in the CSFLD scheme). Finally, we also observe from Figure 3 that undetected multiple bit errors, and the errors that go undetected due to the invalidation of checksum by native functions do not constitute a significant portion.

In order to better understand the interaction between the error consumption behavior and the object characteristics, we also gathered various statistics. Let us focus on the CSOBJ scheme. First, we show the correlation between the lifetime of the object and the error consumption pattern for two representative applications (the

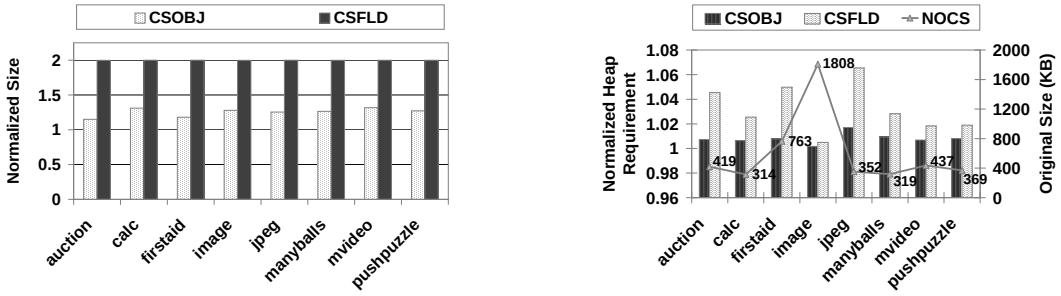

Figure 1: Left: Increase in the total size of the objects allocated due to checksums. Right: Increase in the total heap space requirements to run an application successfully. The curve shows the heap requirements of the original applications without checksums (see the y-axis on the right). The bars correspond to the normalized minimum heap requirements when using CSOBJ and CSFLD as compared to not using checksums (see the y-axis on the left).

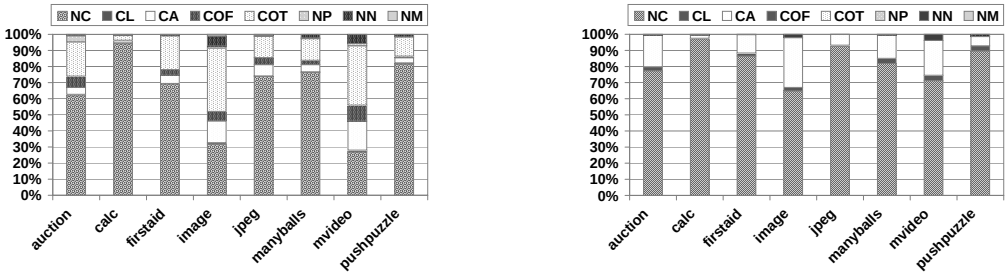

Figure 3: Error consumption behavior of different benchmarks. Left: CSOBJ. Right: CSFLD. Note that the NC class dominates the behavior with both the schemes. The average contribution of NC in CSOBJ and CSFLD cases is 67.7% and 84.7%, respectively.

trends with the remaining applications are similar to one of those presented here; so, they are not shown explicitly). The first bar in Figure 4 shows the distribution of lifetimes of the objects used in benchmark firstaid. In this context, we define the "lifetime" of an object as the number of bytecodes executed between the creation of the object and the time when it becomes garbage. The second bar gives the distribution of objects that had errors injected into them according to their lifetime. It is clear from comparing the first two bars that the objects with a longer lifetime are more prone to soft errors. The remaining bars show the object lifetime distribution for each of the error classes. Looking at the third and second bars, we can conclude that the errors injected into the objects with shorter lifespan are more likely to be harmless as compared to those injected into the objects with longer lifetime. The fourth bar indicates that most errors that are rectified by overwriting new values (CL) occur for objects that have long lifetimes. In other words, if an object lives long enough, then there is a good chance that its error will be overwritten by a write operation. Finally, we observe that in firstaid, 90% of the errors (classes CA, COF and COT) occur on the objects with a lifespan of greater than 132400, even though they constitute only 32% of all the objects in the firstaid benchmark. Consequently, protecting objects selectively based on lifetimes can be explored in future as a good mechanism for reducing the overheads (of error protection) without incurring significant degradation in reliability.

We found that there are similar correlation between the error consumption behavior and other object characteristics, e.g. the number of accesses and object sizes. Objects accessed frequently are more prone to errors, mainly due to the fact that they tend to live longer. Large objects have more errors injected, and hence consume more errors.

4.2 Results with CSTHR and CSGRP

CSTHR is a variant of the CSOBJ scheme where we perform selective checksums depending on the access pattern of the object. CSGRP, on the other hand, divides an object into groups, and associates a separate checksum with each group. Figure 5 shows the tradeoffs involved using different thresholds for the CSTHR approach (only five bechmarks are shown due to clarity concerns). The graph on the left illustrates the increase in execution cycles of CSOBJ when CSTHR is used with different threshold values. We can observe from this graph that using smaller thresholds (e.g., less than 5) is not good from a performance viewpoint as compared to the CSOBJ scheme. In these cases, the overhead for the counter updates and comparison is larger than the checksum computations saved by thresholding. However, for larger thresholds (e.g., greater than 75), we observe performance gains that range from 1% to 4%. However, when the threshold increases beyond 250, the number of errors that are missed due to selective checksum operation becomes non-negligible (see the graph on the right side of Figure 5). Consequently, we conclude that CSTHR may have limited opportunities for performance gain while maintaining reliability constraints.

Next, we focus on the CSGRP technique that associates the checksum with a set of fields, and permits a spectrum of alternatives between CSFLD and CSOBJ. Figure 6(a) illustrates, for the firstaid benchmark, how the area overhead increases as we change the number of fields per group. That is, in these experiments, every object instance is divided into groups of equal number of fields. Note that while the CSGRP scheme comes really close to the CSOBJ scheme when the number of fields per group is equal to 8, there is still some difference between the two. This is because there are some objects that have more than 8 fields. While the concern for space overhead favors working with a large number of fields per group (4 or above), the performance concern demands a smaller number of fields per group (see Figure 6(b)). This is because both the number of instructons executed and the total number of execution cycles increase with the increased number of fields per group as depicted in Figure 6(b). Note, however, that the increase in cycles is much smaller in magnitude as compared to the corresponding increase in the number of instructions executed. This is mainly because as we increase the number of fields per group, we observe a better data cache behavior. Figure 6(c) shows the error consumption behavior of a version of CSGRP where each object has two groups. It should be observed that in a sense this represents an intermediate solution between CSOBJ and CSFLD. The significant observation is that the COF cases reduce significantly as compared to using the CSOBJ strategy. This is due to the finer granularity of checksum in the CSGRP approach.

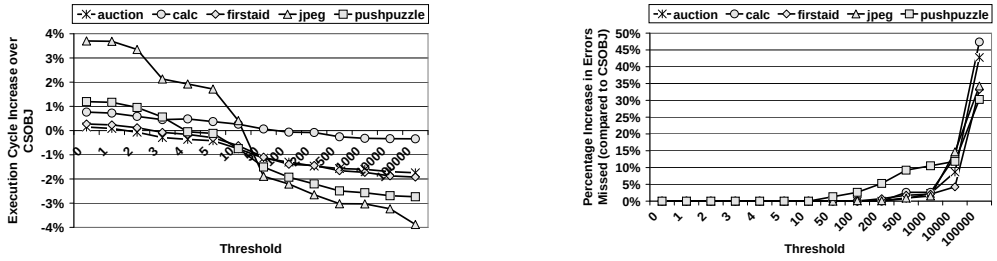

Figure 5: Left: Performance of CSTHR with the firstaid benchmark. Right: Error behavior of CSTHR with the firstaid benchmark.

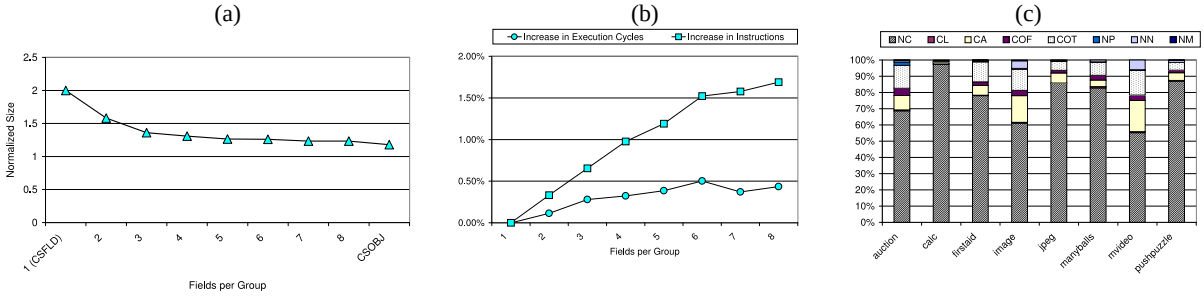

Figure 6: The results for the firstaid benchmark. (a) Normalized size of all allocated objects with different number of fields per group for CSGRP. Values are normalized with respect to those obtained using no checksums. (b) Increase in execution cycles and the number of instructions executed with with different number of fields per group for CSGRP as compared to CSFLD. (c): Error consumption behavior of different benchmarks for CSGRP when there are two groups for every object.

4.3 Evaluation of Hardware-Based Protection

Finally, we evaluate a typical hardware-based protection that uses parity bits. In principle, there can be at least two types of problems with a hardware scheme: false error signals and error cancellations (i.e., missed errors). The false error problem occurs when the multiple fields are placed into the same hardware unit for which a parity bit is provided. For example, if the hardware provides parity for every 8 bytes and 8 bytes hold two fields, the error in one field will trigger a false alarm even if the execution accesses the other field. This is particularly a concern because not all fields are equally important and some of the fields may be beyond their last use. The second problem, namely missed errors, occur with high error frequencies. As an example, under the same hardware-based parity protection scenario above, if both the fields have an error at a given moment, they will cancel out each other, and parity will not reflect the error. Our field-based schemes are much more effective than this hardware-based strategy, due to the more powerful (checksum-based) protection and the match between the access granularity and protection granularity.

5. CONCLUDING REMARKS

Soft error rate is gaining more attention as an industry-wide problem that is the inevitable consequence of scaling CMOS. In particular, in power-aware embedded systems that operate under low supply voltages, soft errors are considered as one of the main problems [7]. This paper looks to enhance the software's ability to detect memory errors in a Java-based embedded system. Specifically, it presents several checksum-based techniques that help us study tradeoffs between reliability, performance, and space overhead.

6. ACKNOWLEDGMENTS

This work was supported by NSF Career Award #0093082.

7. REFERENCES

[1] 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. the 1st USENIX Symposium on Java Virtual Machine Research and Technology,* April 2001.

[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, April 1999.

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

[4] 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,* May 1994.

[5] V. Degalahal, N. Vijaykrishnan, and M. J. Irwin. Analyzing soft errors in leakage optimized SRAM designs. In *Proc. International Conference on VLSI Design,* January 2003.

[6] T. J. Dell. "A white paper on the benefits of Chipkill – correct ECC for PC server main memory," IBM Microelectronics Division, Nov. 1997.

[7] "Intel scans for soft errors in processor designs." http://www.eetimes.com/story/OEG19990616S0016.

[8] W. Kao et al. Fine: a fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering,* Vol 19, No 11, November 1993.

[9] S. Kim and A. K. Somani. An adaptive write error detection technique in on-chip caches of multi-level caching systems. *Journal of Microprocessors and Microsystems,* 22(9):561–570, March 1999.

[10] J. Lyman. Java's surprising comeback. http://www.newsfactor.com/perl/story/18365.html.

[11] D. Milojicic et al. Increasing relevance of memory hardware errors - a case for recoverable programming models. In *Proc. 9th ACM SIGOPS European Workshop,* 1999.

[12] S. Mitra. Test and Reliability Techniques for Robust System Designs. Tutorial in *15th Symposium on High-Performance Chips,* August 2003

[13] B. Murphy et al. Windows 2000 dependability. In *Proc. Dependable Systems and Networks,* June 2000.

[14] L. D. Paulson. Handheld-to-handheld fighting over Java. *IEEE Computer,* p. 21, July 2001.

[15] R. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package,* Kluwer Academic Publishers, 1995.

[16] N. Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Proc. 2nd Java Virtual Machine Research and Technology Symposium,* San Francisco, CA, August 2002.

[17] G.R. Srinivasan. Modeling the cosmic-ray-induced soft-error rate in integrated circuits: an overview. *IBM Journal of Research and Development,* 40(1):77–89, January 1996.

[18] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications,* John Wiley & Sons, 2001.

[19] K. S. Venugopal, Geetha Manjunath, and Venkatesh Krishnan. sEc: a portable interpreter optimizing technique for embedded Java virtual machine. In *Proc. 2nd Java Virtual Machine Research and Technology Symposium,* San Francisco, CA, August 2002.

[20] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *Proc. the International Conference on Dependable Systems and Networks,* pages 411–420, July 2000.

[21] J.F. Zeigler. Terrestrial cosmic rays. *IBM Journal of Research and Development,* 40(1):19–39, January 1996.