In preparation for the submission to Java™ Virtual Machine Research and Technology Symposium (JVM '01)
April 23-24, 2001, Monterey, California, USA. Please do not reference or distribute!

JVM Susceptibility to Memory Errors

Deqing Chen[†], Alan Messer, Philippe Bernadat, Guangrui Fu,
Zoran Dimitrijevic,[††] David Jeun Fung Lie,[‡‡] Durga Mannaru,[*] Alma Riska,[‡] and Dejan Milojicic

Univ. of Rochester,[†] HP Labs, UCSB,[††] Stanford Univ.,[‡‡] Georgia Tech.,[*] William and Mary College[‡]

lukechen@cs.rochester.edu,[†] [messer, bernadat, guangrui, dejan]@hpl.hp.com,
zoran@cs.ucsb.edu,[††] davidlie@stanford.edu,[‡‡] durga@cc.gatech.edu,[*] riska@cs.wm.edu[‡]

Abstract

Modern systems are getting more powerful and using larger memories. However, except at the very high end systems, much less attention is being paid to high availability. This is particularly the case for transient errors that typically take the whole system down. We believe that this situation can be improved by addressing memory failures at all levels of system. If we can handle the memory errors at the application level, we can bring mainframe-class availability closer to commodity systems.

In this work we investigate memory susceptibility of a JVM (Kaffe) and four Java benchmark applications by fault injection experiments. We then implement a prototype for detecting silent data corruptions in the JVM and evaluate its effectiveness. We find that the JVM's heap area has higher memory error susceptibility than its static data area. Our prototype implementation for detecting silent data corruption can detect up to 39% of all JVM and application errors caused by memory errors. By using such techniques we propose that commodity systems can be made much more robust and error-prone to transient errors.

1 Introduction

Demand for high performance and availability of commodity computers is increasing with the ubiquitous use of computers and the Internet services which serve them. While commodity systems are tackling the performance issues, availability has received less attention. It is a common belief that software errors and administration time are, and will continue to be, the most probable cause of loss of availability. While such failures are clearly commonplace, especially in desktop environments, it is believed that certain other hardware errors are also becoming more probable.

Hardware errors can be classified as hard errors and transient (soft) errors. Hard errors are those that require replacement (or otherwise relinquished use) of memory resource. These typically happen as a consequence of physical damage of the memory chips, e.g. by damage to connectors. Transient errors are those that result in an error in the memory content, but once overwritten, the same memory areas can be re-used. Ziegler et al. [19, 22] have shown that factors such as increased technology density and reduced supply voltage will lead to increased transient errors in CMOS memory due to the effects of cosmic rays. Tandem [19] indicates that such errors also apply to processor cores and on-chip caches at modern die sizes/voltage levels.

Although the increasing application of Error Correction Codes (ECC) can significantly reduce the probability of these transient errors, increasing speeds, denser technology, and lower voltages increase the possibility of these errors becoming significant in future systems. Even if ECC protection is used, there is still a possibility of multiple bit errors that escape the scope of the protection. When this occurs values in random memory locations are corrupted, leaving the application to use a potentially incorrect value when used on execution, this is called "silent data corruption". Typical examples are transient errors on the processor registers, ALU, multiple-bit memory errors, and similar. In some of the most promising applications of Java technologies, such as embedded systems, no parity or ECC protection is used, allowing more of these errors to be exposed to the system.

In current commodity systems, there is little consideration for transient memory errors. For example, in most systems based on the IA-32 architecture, when a transient memory error happens the CPU simply enters a Machine Check Abort (MCA) exception from which the OS can only panic or reboot.

However, in the new IA-64 architecture, there is increased scope for useful MCA handling. At the time of the MCA exception, the CPU can provide

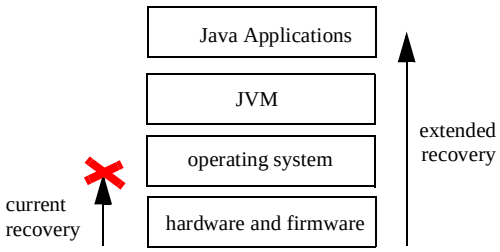

Figure 1 Propagating Memory Errors. *Memory errors are detected by lower layers and either corrected or propagated to higher levels of the systems, up to applications*

much more information about the current CPU status and can notify the operating system to handle the exception. This ability allows new opportunities for future systems to recover more gracefully from memory errors.

Existing research [12] has outlined the opportunity of memory error recovery with this increased hardware support. However, the whole system recoverability is a complex problem that involves participation at each level from the hardware to the application software. Using this research, the operating system can be extended to increase recoverability when it receives a memory error exception. If it finds the error happens in the Java virtual machine or the application, it is possible it can deliver the error exception to the virtual machine for further processing (see Figure 1).

At the application level, the JVM and Java application is of particular interest due to the large garbage collected heap, machine abstract presented, and the integral exception mechanisms.

Large garbage-collected heaps present a sweet-spot for this research, since the garbage collector itself may uncover more errors as part of the heap sweep during collection. They are also usually larger than explicitly allocated heaps, thereby increasing the probability of error during this sweep.

By presenting an abstraction between the operating system and the applications, the virtual machine makes application level recovery simpler. Since, the virtual machine has increased details of the application's status and semantics, such as memory usage, improve the chance of recovery.

Java's integral exception handling allows applications to be written that are memory error aware [12] by trapping new exceptions. If the virtual machine can isolate the error solely to the application, it can generate these exceptions and thus allow the application to handle the memory error gracefully.

Memory failure recoverability is a complex problem. This paper tries to identify the memory error susceptibility in the Java virtual machine and Java applications as a first step to tackle this potential problem. The major contributions in this paper include: quantifying the memory error consumption and susceptibility rate in the Kaffe virtual machine and Java applications; and, evaluation of extensions to the Kaffe virtual machine to detect silent data corruptions.

The rest of the paper is organized in the following manner. In Section 2, the paper outlines related work to the problem. Section 3 describes the problems that we are addressing. The methodology of the fault injection experiment and the method for detecting silent data corruptions are described in Section 4. Section 5 presents the experimental results. Lessons learned are presented in Section 6. The paper ends with conclusions and recommendations for future work.

2 Related Work

The effect and trends for soft-errors were first reported by Ziegler et al. [19, 22] based on field and experimental evidence that alpha particles and cosmic rays were the source of several random system failures. Since this time, soft-errors have become more of a concern, because semiconductor susceptibility to these particles increases with density increases and voltage drops.

Availability in computer systems is determined by hardware and software reliability. Hardware reliability has traditionally existed only in proprietary servers, with specialized redundantly configured hardware and critical software components, possibly with support for processor pairs [2], e.g. IBM S/390 Parallel Sysplex [15] and Tandem NonStop Himalaya [5].

Software reliability has been more difficult to achieve in commodity software even with extensive testing and quality assurance [13, 14]. Commodity software fault recovery has not evolved too far at this time. Most operating systems support some form of memory protection between units of execution to detect and prevent wild read/writes. But most commodity operating systems have not tackled problems of memory errors themselves or taken up software reliability research in general. Examples include Windows 2000 and Linux. They typically rely on failover solutions, such as Wolfpack by Microsoft [16] and High-Availability Linux projects [20].

A lot of work has been undertaken in the fault-tolerant community regarding the problems of reliability and its recovery in software [3, 7, 11]. These include techniques such as checkpointing [7] and backward error recovery [3]. A lot of this work has been conducted in the context of distributed systems rather than in single systems. There are also techniques for efficient recoverable software components, e.g. RIO file cache [4], and Recoverable Virtual Memory (RVM) [17].

The Fine [10] project uses fault injection technique to study the fault tolerance of the UNIX system. Fine is a set of experimental tools capable of injecting hardware induced software errors and software errors into the UNIX kernel and tracing the execution flow and kernel's key variables. Instead, our fault injection work operates at the application level and using the debugger tool – *ptrace* – for tracing the application's behavior.

A few pieces of research have attempted to quantify the absolute number of errors that would be seen in particular configurations [21, 19, 6]. For example, it is estimated that a 1Gb memory system based on 64Mbit DRAMs still has a combined visible error rate of 3435 FIT when using Single Error Correct-Double Error Detect (SEC-DED) ECC [6]. This is equivalent to around 900 errors in 10000 machines in 3 years. Tandem [19] estimate a typical processor's silicon can have a soft-error rate of 4000 FIT, of which approximately 50% will affect processor logic and 50% the large on-chip cache. Due to increasing speeds, denser technology, and lower voltages, such errors are likely to become more probable than other single hardware component failures.

Most recently work at HP Labs [12] has undertaken further research into the future trends of these error rates, their repercussions on the processor support, operating system handling/recovery , and working towards application recoverability. The work in this paper was undertaken as part of this work at HP.

3 JVM and Java Applications Memory Error Susceptibility

In a system that has not enough ECC hardware or multiple-bit error happens, transient memory errors can be exposed to the software system. In addition, silent data corruption can not be caught by ECC (it typically happens on the hardware that is not protected by ECC) and hence needs to be handled in an application-specific way. In this paper, we concentrate our effort on those corruptions that happen in the application's data area. The errors in the kernel area are beyond the scope of this study and are addressed elsewhere [12].

Suppose a transient error happens on a word inside an application's data area, the error may or may not be consumed (accessed) by the application. If the error is consumed, the error may or may not eventually lead to an application error. For example, let's suppose an error is injected to an ID string array so that one ID is changed unexpectedly. If this ID is never matched in a search sometime later, the error won't lead to visible application errors.

In this paper, we call the application access of a memory soft error the error consumption. If the error consumption eventually leads the application to crash or output a wrong result, we say that this error causes an application error. The application susceptibility to errors is related to this data area.

Studying the error consumption and susceptibility of applications has many valuable benefits. Most of all, it lets us understand the application behavior under the silent data corruption so that we can design an efficient software method to detect silent data corruption. Since it's unfeasible to detect all of the errors, this study focuses on data areas most

susceptible to memory errors. The rest of this section defines the terms we used in the paper and describes the environment we used for experiments.

3.1 Memory Error Susceptibility

In this paper, we define the memory consumption rate ($R_{consumption\_rate}$) as the ratio of number of errors consumed ($N_{error\_consumed}$) versus the number of memory errors (N_{memory_errors}), i.e.,

$$R_{consumption\_rate} = N_{error\_consumed} / N_{memory\_errors}$$

This equates to portion of the total error rate which are actually seen by the application, since only errors in those memory locations accessed are noticed. The consumption rate is always smaller than one. Thus our definition of consumption rate is the upper bound in errors seen by the execution in a real situation. For simplification, in this paper we assume a memory error persists until it is consumed or the application exits. This is necessary since some high-end operating systems use a memory scrubber to pass over physical memory removing any correctable errors if finds. In the presence of ECC memory, the memory scrubber can clear all correctable errors that exists in the memory.

The memory susceptibility ($S_{susceptibility}$) for a memory area is defined as the ratio of actual application errors ($N_{errors\_in\_application}$) divided by the number of memory errors (same as in previous formula), i.e.,

$$S_{susceptibility} = N_{errors\_in\_application} / N_{memory\_errors}$$

Application errors are those that either cause an application crash or execution to return an erroneous result. Verification of the latter is performed by comparing the result against a known correct result. In this paper, we assume that memory errors are distributed uniformly in the application's total virtual memory area. Since memory errors affect physical memory this is equivalent to assuming the working set fits into physical memory.

3.2 JVM and Java Applications Memory Error Susceptibility

In a Java virtual machine, the data area can be divided roughly into two partitions, those allocated statically for the virtual machine (VM) and those allocated on the heap for Java objects. We want to identify the error susceptibility of these two different memory areas to guide our future recovery studies. For errors in the heap, we also want to know how the susceptibility varies with different heap object types.

One feature of the JVM is that unused Java objects are not freed explicitly by the application, rather they are collected and freed by the garbage collector. How the garbage collector (GC) consumes the memory errors is also interesting to us.

Since silent data corruption may not be detected by the hardware solutions, we need to design a software solution to detect these errors. We propose a simple detection scheme by checksumming the heap objects with evaluation of its efficiency by fault injection.

3.3 Kaffe Virtual Machine

Kaffe became our choice because it is an open source package that allows us to get its source code and extend it freely. Having its source code allows us to examine its memory usage, to instrument it for fault injection experiments, and to extend it to detect silent data corruption. It is also a mature system and has reasonable performance, and is widely used.

The operating system we use is Redhat Linux 6.2. We use Kaffe 1.0.5 with the "interpreter mode" throughout our studies. Since our work only assumes an IA-64 like error handling architecture and we don't have Kaffe on IA-64 yet, we use a Pentium-III platform of the IA-32 architecture instead of the IA-64 architecture. Where appropriate we shall point out the implications of using one or the other processor.

4 Experiment Methodology

In this section, we first explain the method and setup of the fault injection experiment. Next we describe our prototype implementation for detecting silent data corruptions.

4.1 Fault Injection Experiment Method

Our basic experiment method is to inject errors into the application data area, track the error consumption, and monitor the application behavior after the consumption. We use the *ptrace* system call to trace the JVM execution and manipulate the debug registers to set a data breakpoint to track the error data consumption.

Data Breakpoints

In the IA32 architecture, there are eight debugging registers we can use to set data breakpoints. They are identified as DR0 – DR7. To our interest here, DR6 is the breakpoint status register, DR7 is the debug control register, DR0 – DR3 are used to set the addresses of breakpoints.

For each breakpoint address, IA32 architecture allows the user to set it for breaking on execution, breaking on writes, or breaking on read-write. In this experiment, we set the CPU to break on read-write of the injected-error address. At each time, we only set one address. This method has the limitation that we can't figure out whether the access is a read or a write. We can overcome this limitation by duplicating the breakpoint and setting one for read-write and the another for write. But we are unable to get the correct debugging status register value from the Linux system. Therefore we don't know which breakpoint fires. It may be possible to overcome this limitation in the future.

Using ptrace

Debug registers are privileged CPU resources and a user application can't read and write them directly. Fortunately Linux provides the *ptrace* system call for accessing these registers from user processes.

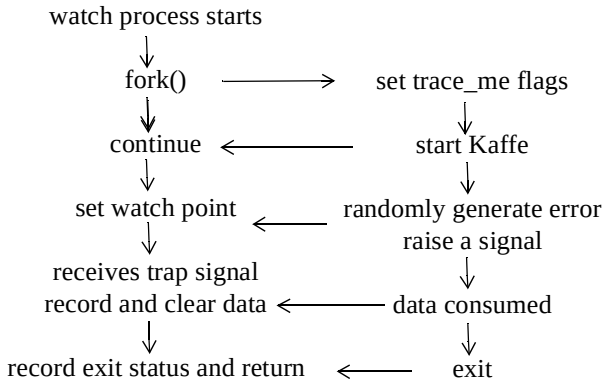

Figure 2 Method for tracing error consumption using ptrace.

Normally, a *ptrace* system call is used in the following way. The debug process uses *fork* to create a child process. On return from the fork, the child process calls *ptrace* with the parameter TRACEME to inform the parent process that it wants to be traced. The child process then calls *execl* or other similar functions to execute the debugged application. On the other side, the parent process calls a *wait* on the return from the *fork*. When the child process first calls *execl*, or generates some uncaught signals, the parent process wakes up from the previous *wait*. After being woken up, the parent process can examine and set the child process status by using the *ptrace* call.

The way we use *ptrace* is illustrated in Figure 2. We modified Kaffe executive to start the watch (monitor) process first. The watch process uses *fork* to create and run the VM. At certain points of VM execution, a memory error is generated and a SIGTRAP is raised to inform the parent – the watch process – to set a data breakpoint on the error address. On receiving this signal, the watch process peeks at the child process data (since they have the same address space layout, we can get child's data easily) and sets the appropriate data breakpoint.

After the child process resumes, it may consume or not consume the injected error. If the error is consumed, the child process traps and the parent wakes from this trap signal. The consumption is recorded and the breakpoint is cleared. Whenever the child process exits normally or incorrectly, the watch process is signaled and the status is

recorded. If the child process exits normally, we further check whether its output is correct.

Generating and Recording Memory Errors

We instrument the Kaffe virtual machine to inject memory errors into the data memory area and to record the memory status. Since we are using the interpreter mode, the virtual machine executes a loop interpreting each byte code. Code is instrumented so that after a certain number of byte codes have been executed, the loop calls our error injection procedure to generate a memory error.

Each Memory error is injected into one of two data memory areas:

- the static memory area of the VM, and
- the object heap.

In each test set, errors are injected into one of the above areas. Each time, a byte is randomly chosen from the specified area and the byte bits are flipped.

If the error is injected into the object heap, we record the type information of the object where the byte is located, the information we record includes the object type, size, base address.

Now, the VM stores the error address into a global variable and raises a SYSTRAP signal to inform the watch process that a memory error has been generated. After receiving this signal, the watch process peeks at the global variable to get the error address and set a data breakpoint at the address. Then the VM is allowed to continue.

When the error is consumed, we also inspect the VM status to see whether it is consumed by the garbage collector. The Kaffe uses the mark and sweep algorithm which makes this inspection fairly easy; when the GC is running all of other user threads are stopped.

4.2 Detecting Silent Data Corruption

Based on our experimental results on error consumption, we have implemented a prototype solution for detecting silent data corruption for the Kaffe virtual machine. We believe the method can be applied to other virtual machine implementations as well.

The basic idea is that in a pure Java application every Java object or array is accessed through a determined group of bytecode operations, such as getfield and putfield. For each of these bytecodes, we add code to do a checksum computation according to their functionality. The heap object management can be modified to store the checksum results.

Space For Checksum

Instead of directly extending Kaffe's Object data structure to have extra fields for storing checksum data, we extend the heap memory management data structure to have more bytes for each memory block. This conforms to the way that Kaffe manages the object status.

In the Kaffe heap memory management module, objects are classified into small objects and big objects. Small objects are generally objects with sizes smaller than the system page size. Large objects are objects needing more than one page.

Small objects are grouped into pages. Each page is divided into many same size blocks. Each block is assigned to one object. At the head of the page, there is a meta-data structure for blocks inside the page, such as block size, garbage collection status and object type. Two bytes are added for each small object. One byte for bit pattern checksum, another for checksum validity.

For big objects and arrays, it is not desirable to have only one checksum data for efficiency reasons. For example, when one byte in a one-mega-byte array is accessed, we don't want to compute a checksum for the whole one-mega byte data. Thus, we divide the object into fixed size small blocks and the checksum is computed on these small blocks. Although we add extra memory overhead, the checksum computing is much more efficient for large objects or arrays.

Checksum Computing

In our implementation, the checksum is computed by xoring all bytes in the object rather than adding short values together as in TCP/IP.

When a Java application is running, an object can only be accessed through the following ways:

- It's created by using new operator,
- one of its field is read by the bytecode getfield (static field access excluded, see explanation below),
- an entry in an array is read by bytecodes iaload, laload, faload, daload, caload, saload, baload and aaload,
- the object is walked by garbage collector,
- one field is written by the bytecode putfield (static field write excluded again),
- an entry in an array is written by bytecodes iastore, lastore, fastore, dastore, castore, sastore, bastore and aastore,
- one part of an array is copied by System.array_copy,
- the object or array is operated by some native functions.

in Kaffe, static fields are stored with the class objects(this makes

sense, static fields are global variables of the class). If we want to

include them in checksum protecting, we have to checksum the class object.

But other fields of the class objects are manipulated by the VM in many

places, it's hard to instrument all of them. We instrumented Kaffe to deal with these situations. When an object field or an array entry is read by some bytecode, we compute the checksum of the read value with the other part of the object or array and compare it with the checksum we have previously stored in the object's block meta data structure. When an object is updated by bytecode, we update its checksum value.

5 Experiment Results

In this section, we present our experiment results for error consumption and silent data corruption. In our experiments, we assume a uniform memory error probability over the whole memory area. For the convenience of the experiments, we inject the same number of errors in the two experiment sets.

The benchmark applications we used in the experiments are extracted form SPEC JVM98 benchmark suits [18]. The four applications we used are:

- _202_jess, a java expert system,
- _209_db, a java database,
- _213_javac, a java compiler, and
- _228_jack, a java parser generator.

In all of the experiments we conducted, we use the medium data configuration – ten percent. With this is data size, we can have the experiments finished in reasonable time and the garbage collector is forced to run.

For both static and dynamic areas we inject 1000 memory errors for the four benchmarks. For the dynamic area experiments, the benchmarks are run with the error detection mechanism so that we can record which error consumptions have been detected. The total running time for the experiments took about 70 hours on a Pentinum III 500MHZ platform. The total code lines for error injection and tracing is bout 470 lines and the code lines for memory error detection is about 780 lines.

5.1 Memory Error Consumption

This experiment is divided into two groups. In one group we inject memory errors into the VM's static memory area; in the other group, we inject errors into the object heap. These two areas are used differently by the Kaffe virtual machine. The static data area includes the global variables and constants. Intuitively, errors in this area are much more likely to cause real problems in the Java application once they are consumed. On the other hand, a Java application's data objects are stored on the heap and the heap is walked by the garbage collector when it is started. The heap can have a higher

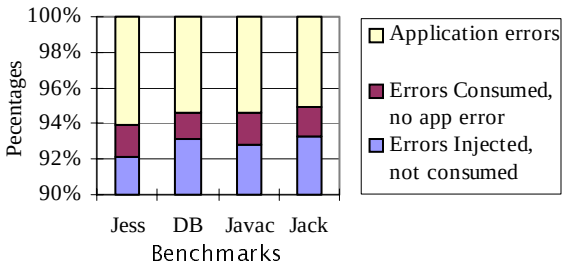

Figure 3 Error consumption in JVM's static data region.

error consumption rate than static data area due to the factor of garbage collection.

Static Memory

The result for injecting errors into static data area is summarized in Figure 3. In the graph, gray part comprises those errors that are not consumed by the application even though they are injected; dark-gray part comprises errors that are consumed by the application but don't cause any application errors, i.e. application accessed the erroneous data but it still executed correctly; white part illustrates number of application errors, in this case application either crashes or gives a wrong result.

The susceptibility rates are listed in Table 1. The size of this data area is about 350KB. We can see from the graph that all of the benchmark applications exhibit similar behavior. Their error consumption rate is about 6% to 7% with an average of 6.7%. The average memory susceptibility rate is about 5.5%. Among all of the errors consumed, 81% of them cause errors in the applications. As we mentioned before, our tools can't differentiated between write consumption or read consumption here. This means that almost all of the read consumption causes a real problem in the application.

Static Data	Jess	DB	Javac	Jack	Avrg
Susceptibility	6.2%	5.4%	5.4%	5.1%	5.5%

Table 1 Susceptibility in Static Data

Object Heap

In the next experiment, we inject errors into the object heap. In Kaffe, the heap size grows dynamically as the application's need grows. In our experiments, we inject errors into the range of virtual address the heap occupies. Their heap size varies from 5243KB to 8397KB in the experiment.

Heap Size	Jess	DB	Javac	Jack
Minimum Heap Size	5243KB	7348KB	5243KB	5243KB
Maximum Heap Size	5243KB	8397KB	7000KB	7000KB

Table 2 Heap Size Used in Error Injection

The result is summarized in Figure 4 and the susceptibility rates are listed in Table 3. The three cases (application error, consumed but no error, and injected but not consumed) have the same meaning as in Figure 3.

The first observation we can draw is that the heap has a much higher error consumption rate. For example Jack has a 75% error consumption rate in the heap versus 6.7% in the static data area. But a closer look revealed that most consumption come from the garbage collector. Kaffe uses mark and sweep strategies for garbage collection. When GC is started, it virtually touches almost every object in the heap. It's no wonder it consumes so many errors.

If we don't count those errors consumed in the GC, the error consumption rate is about 9% to 22%, which is still higher than the static data area.

It should also be noted that the susceptibility also depends on memory region size. However, if we assume a uniform probablility error rate in the memory area, since the heap size is much bigger than the static area, we can conclude that the heap is much more memory susceptible of the two.

Although most of the consumption takes place in the garbage collector, relatively fewer of them actually cause real problems. The first reason is that the garbage collector only cares about object's reference field. It won't use other types of fields for computations. For object reference, it first checks whether it is valid. This masks out most of the possible errors. On the average, only 7% of the error consumption in GC caused application errors. In

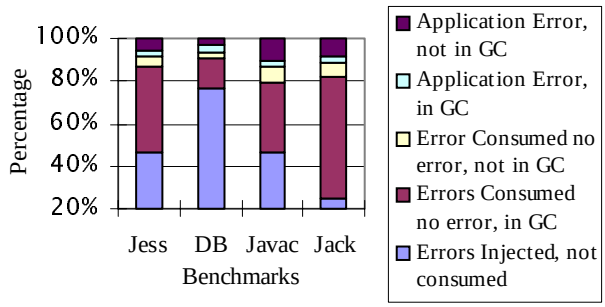

Figure 4 Error Consumption in the JVM's Heap region.

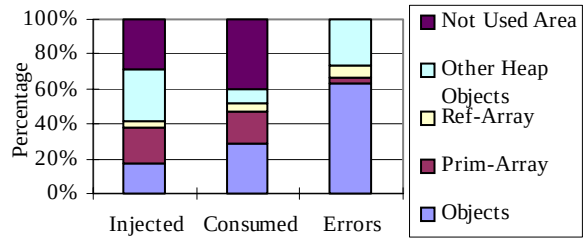

Figure 5 Error Consumption by Object Types.

comparison, 56% of static data error consumption cause application errors.

Object Heap	Jess	DB	Javac	Jack	Avrg
Susceptibility	8.3%	7.1%	13.2%	11.9%	10.1%

Table 3 Susceptibility in the Heap

To further understand the source of application errors, we also collect the object types for the object into which each error is injected. Here we show the result for Javac as an example in Figure 5. We distinguish objects, primitive arrays, reference arrays, and areas that is not used. Example of the latter is areas that do not belong to any JVM. object. For example, an object has been freed by the garbage collector, or a block inside a page that has not been allocated to any object. In Figure 5 the errors injected into not-used part, never caused application error. However, they may be consumed, by overwriting.

From the graph we can see that although only less than 20% errors injected are in normal (created with new) objects, they are much more likely to be consumed and cause application errors – more than 60% application errors are caused by these objects.

We can also see that many errors are injected into primitive arrays. This is understandable since user applications tend to store large data sets into arrays. But since these are large structures particular single errors they are less likely to be consumed, since array accesses may rarely use the erroneous data. Therefore, depending on application data usage errors in primitive arrays may cause less application errors than these error consumption rates indicate. On the other hand, the reference arrays are much more likely to cause application errors, since a false pointer can almost always cause a segment fault in the JVM (except those situations mentioned before in GC).

Due to the space limitations, details on other error data types is not included here. Briefly, in the "other heap object" part in Figure 5, constant fixed objects occupy a large percentage of these objects. These objects include bytecodes, the constant pool, etc.

5.2 Checksum Silent Data Corruption Detection

To demonstrate effectiveness of our scheme for detecting silent data corruption, we implemented a prototype in Kaffe. Compared to the proposal, the prototype implementation has several limitations. First when native functions or System.array_copy is called, we simply clear the object's or array's checksum status rather than update the checksum result, although in the future we will do so.

Another limitation is that we don't compute checksum for large objects, although we do deal with large arrays. We assume that we don't expect to see many large objects in Java applications because in a Java object, embedded objects are stored as a reference.

We ran the fault injection experiments on our prototype implementation with the four benchmarks. We recorded the cases when consumed errors are detected. We also compared relative slowdown of the prototype implementation with the original Kaffe implementation. These two results are summarized in Figure 6. Here we show the percentage of application errors that can be detected when the error is consumed. The white areas represent errors detected. "Errors in Object and Array" represent those errors that we know took place in objects and

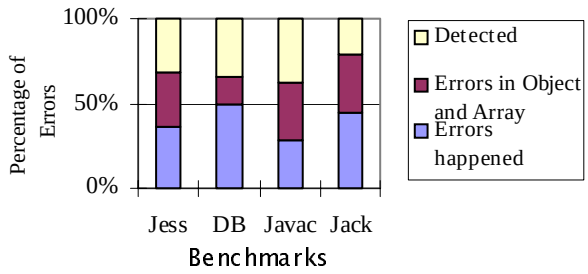

Figure 6 Checksum Detection of Application Errors

arrays and we could have corrected if we applied checksumming; checksumming has not been applied because the object is too big or operated by some native functions that is not easily checksummed. Finally errors happened are the cases where error was not detected and corrected so it caused an error.

The effectiveness of the detection depends on the nature of the application. If the objects and arrays account for most of the actual errors happening, the technique is more effective. For example for Javac, errors in objects and arrays account for nearly 80% of all error occurrences, our technique can detect up to 38% of all errors. This percentage is only the current implementation limitation, it can be significantly improved.

In the future, we can improve our implementation by updating checksums during native function calls and array copies. We can also extend the technique by including more heap objects into the checksum detection, such as constant pools and bytecode sections. Since they're never changed after they are loaded into the heap, the extra overhead for checksumming them would be small, since only checks on read access would be required.

It is also interesting to see the performance overhead induced by the checksum process. We measure the execution time of original Kaffe implementation and our prototype implementation. The relative slowdown compared to the original version is shown in Table 4 for each benchmark used. On the IA-64 architecture, performance can be improved by at least four times, due to the ability to use multiple arithmetic units explicitly to parallelize the computation.

	Jess	DB	Javac	Jack
Slowdown	57%	43%	47%	32%

Table 4 VM Slowdown with Detection

6 Lessons Learned

We found ptrace is a good tool for fault injection experiment. It lets us generate data breakpoints in the Kaffe VM and track the consumption of the injected errors. At the time of error consumption, the breakpoint allows us to stop the VM and examine the its internal state. Originally we had thought of collecting execution traces to study the error consumption rate, but it would be extremely difficult for us to derive the VM's status at the time of error consumption from the traces. Of course, ptrace has limitations. It is not clear to us whether we can use it successfully on the kernel mode study.

From the experiment data and analysis, the following interesting observations can be derived:

- For Kaffe virtual machine and the Java applications running in it, the memory errors in the object heap have a higher error consumption rate and susceptibility rate than those in the static data area. The heap size is also much larger than the static data size. If we assume a uniform error distribution, we can draw the conclusion the heap memory will be the dominate part in memory susceptibility.
- A large portion of error consumption in the heap is caused by the garbage collector (up to 75% in the case of Jack). But this consumption leads to less application errors than other consumption (7% vs. 56%).
- For memory errors occurring in the object heap, errors injected in normal objects (created with new) and arrays cause 70% of the application errors.
- By adding simple checksums normally undetected errors can be detected, increasing error coverage by 30-40%. While this comes at a cost, even

using an unoptimized checksum routine this functionality only increase run-time by 32-57%.

- Silent data corruption detection should be easily increasable by placing checksums over more object types (e.g. static objects). While the overhead could be further reduced by limiting additional unnecessary checks.
- Several objects in the Java heap can be relatively large and were not covered by our checksums.

7 Summary and Future Work

We still need some further work to complete our study on memory failure recoverability at the application level. First of all, we need to extend and optimize our prototype silent data error corruption implementation to handle other heap objects, these include large object, constant pool, byte code, etc. Using this extension we can expect to achieve a higher error detection rate.

Secondly, to further reduce the affect of the garbage collector on detectng errors, it is would be possible to modify it to use memory defensively to expect memory errors and recover from them itself. This is very similar to the construction of the memory scrubber task in high-level operating systems.

Thirdly, it would be interesting to investigate further to relationship between consumption rates and susceptibility. While both factors depend largely on the application workload and its input, we would like to understand further any correlations or classifications of susceptibility to consumption rates.

7.1 Handling memory errors with Java

Java is ideally suited to handle memory errors. Its exception mechanism provides an elegant memory error recoverable programming model [1].

A typical exception handler looks like this

```
try {
  ... some action ...
 } catch (Exception_type_1 e) {
  ... some exception handling ...
 } catch (Exception_type_2 e) {
  ... some exception handling ...
 } catch ...
    ...
 } finally {
  ... action performed in any case ...
 }
```

We propose introducing a new MemoryErrorException class, a subclass of RuntimeException. We define it as a runtime exception since the error may happen at anytime and the developer shouldn't have to catch it when unnecessary.

The developer is free to insert try-catch statements at critical locations and to define their scope.

The MemoryErrorException includes some fields or methods to

- indicate the severity (restartable or not)
- indicate the type (transient, SDC)
- point to the affected java object and object field.

When the error affects the JVM's internal state, as a last attempt the VM throws an exception indicating that it can't recover. The handler should cleanup its state in minimum action and exit.

Otherwise the handler may query the affected memory and perform appropriate actions. Note that the error may not be overwritten nor cleared by the handler, in which case other exceptions may be raised subsequently.

This model applies to both transient memory errors and silent data corruption.

8 Summary

In this report, we have described our effort in studying the memory error susceptibility of the Kaffe virtual machine by the method of fault injection. We found that for the Kaffe VM and the benchmark applications we ran the heap objects comprise most of the memory error consumption. We also present our prototype implementation for detecting silent data corruptions by object checksum. We find that this simple technique can detect up to nearly 40% of all application errors caused by silent data errors.

All experiments are executed in the Kaffe's interpretive mode. In order to use Kaffe with its superior performance JIT compiler, the JIT would need to be modified to generated the checksum routine inline with object accesses. Given that errors can occur in any memory, it would also be possible to consider checksumming the generated code, if its size proves this to be necessary. Apart from this, Kaffe using its JIT should have the same overall behavior as it been has reported here, since the same heap management system is used.

While introducing extra overhead of between 32-57% might seem counter to todays JIT research, this overhead represents an upper-bound on performance loss. Early results indicate the CRC checksums on the IA-64 architecture can run up to four times the speed as on IA-32 architecture processors.

Acknowledgments

We are indebted to Peter Markstein and (add other contributors XXX) for commenting on the context and presentation of the paper. Their help significantly improved the document.

References

[1] Arnold, K., Gosling, J., Holmes, D., The Java Programming Language, Third Edition, Sun Microsystems, 1999.

[2] Bartlett, J., "A Nonstop Kernel", Proceedings of the Eighth Symposium on Operating Systems Principles, Asilomar, Ca, pp 22-29, Dec. 1981.

[3] Brown, N. S. and Pradhan, D.K. "Processor- and Memory- Based Checkpoint And Rollback Recovery", IEEE Computer, pp 22-31, Feb. 1993.

[4] Chen, P. M., et al., "The Rio File Cache: Surviving Operating System Crashes", *Proc. of the 7th ASPLOS,* pp 74-83, October 1996.

[5] Compaq, Product description for Tandem Nonstop Kernel 3.0. Download Feb. 2000, http://www.tandem.com/prod_des/tdnsk3pd/tdnsk3pd.htm.

[6] Dell, T. J., "A White Paper on the benefits of Chipkill - Correct ECC for PC Server Main Memory", IBM Microelectronics Division, Nov. 1997.

[7] Gray, J., and Reuter, A., "Transaction Processing: Concepts and Techniques," Morgan Kaufmann, 1993.

[8] Intel IA-64 Architecture Software Developer's Manual, Volume 2, Intel 1999.

[9] Intel IA-32 Architecture Software Developer's Manual, Volume 3, Intel 1999.

[10] Kao, W., et al., "Fine: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults", IEEE T-SE vol. 19, no.11, November 1993.

[11] Kermarrec, A-M., et al., "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability", *Proc. of the 25th FTCS*, pp 289-298, June 1995.

[12] Milojicic, D., et al., "Increasing Relevance of memory Hardware Errors – A Case for Recoverable Programming Models", 9th ACM SIGOPS European Workshop.

[13] Murphy, B., et al. "Windows 2000 Dependability", Proc. IEEE International Conference on Dependable Systems and Networks, June 2000.

[14] Murphy, B., et al. "Measuring System and Software Reliability using an Automated Data Collection Process", Quality and Reliability Engineering International Vol 11, pp 341-353, 1995.

[15] Nick, J.M., et al., "S/390 Cluster Technology: Parallel Sysplex", *IBM Systems Journal*, vol 36, no 2., pp 172-201, 1997.

[16] Pfister, G., "In Search of Clusters", Prentice Hall, 1998.

[17] Satyanarayanan, et al. "Lightweight Recoverable Virtual Memory". Proc. SOSP, pp 146-160, Dec. 1993.

[18] Standard Performance Evaluation Corp. (SPEC) "SPECjvm98 Specification", August 1998. http://www.spec.org/osg/jvm98/

[19] Tandem, Compaq Corporation, "Data Integrity for Compaq NonStop Himalaya Servers", White Paper, 1999.

[20] Tweedie, S. "Designing a Linux Cluster", Technical White Paper, Red Hat, January 2000. Also see: http://www.linux-ha.org/

[21] Ziegler, J. F "IBM experiments in soft fails in computer electronics 1978-1994", IBM Journal of Research and Development, Vol 40, No 1, pp 1-136, January 1996.

[22] Ziegler, J. F. "Terrestrial cosmic rays", IBM Journal of Research and Development, Vol 40, No 1, pp 19-40, January 1996.