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 computer systems are becoming more powerful and are using larger memories. However, except for very high end systems, little attention is being paid to high availability. This is particularly true for transient memory errors, which typically cause the entire system to fail. We believe that this situation can be improved by addressing memory errors at all levels of the system, bring commodity systems closer to mainframe-class availability.

In this paper, we use fault injection experiments to investigate memory error susceptibility at the highest level using a JVM and four Java benchmark applications. We then consider JVM data structure checksums to increase detection of silent data corruption affecting the JVM and applications. Our results indicate that the JVM's heap area has higher memory error susceptibility than its static data area and that we can detect up to 39% of all memory errors in the JVM and application. We believe that such techniques will allow commodity systems to be made much more robust and less error-prone to transient errors.

1 Introduction

The demand for high performance and availability in commodity computers is increasing with the ubiquitous use of computers and Internet services. While commodity systems are tackling the performance issues, availability has received less attention. It is a common belief that software errors and administration down-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, the probability of certain hardware errors is increasing.

Hardware errors can be classified as hard errors and transient (soft) errors. Hard errors are those that require replacement (or otherwise relinquished use) of the component. They typically are caused by physical damage to a component, e.g. by damage to connectors. Transient errors are those that result in an invalid state that can be corrected, for example, by overwriting a corrupt memory location. Ziegler et al. [21, 22] have shown that factors such as increased semiconductor technology density and reduced supply voltage will lead to increased transient errors in CMOS memory because of the effects of cosmic rays. Tandem [19] indicates that such errors also apply to processor cores and on-chip caches at modern die sizes and voltage levels.

Although the increased use of Error Correction Codes (ECC) can significantly reduce the probability of these transient errors, greater speeds, denser technology, and lower voltages increase the likelihood of these errors becoming significant in future systems. Even if ECC protection is used, multiple bit errors may still escape the scope of the hardware protection and corrupt values in random memory locations. Applications can then potentially use incorrect value on their next access, this is called "silent data corruption". Typical examples are transient errors in the processor registers, in the ALU, multiple-bit memory errors, and so forth. As a result, when these errors escape hardware protection, it is only possible for software to detect them.

In some of the most promising applications of Java technologies, such as in 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 [9], when a transient memory error occurs, 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 [8], there is increased scope for useful MCA handling. At the time of the MCA exception, the CPU can provide

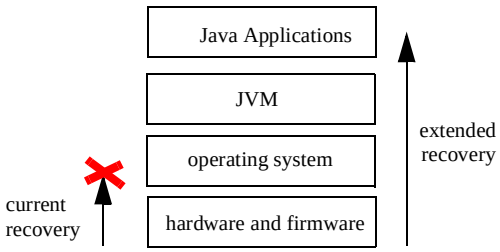

Figure 1 Propagating Memory Errors. *Memory errors are detected by lower layers and either corrected or propagated to higher levels of the system, up to applications*

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

Existing research [12] has outlined the opportunity for memory error recovery with increased hardware support. This research proposes that the operating system can be extended to increase recoverability when it receives a memory error exception. However, recoverability of the whole system is complex and involves participation at all levels from the hardware to the application software. We propose that if the OS determines that a memory error occurred in an application, it can devliver the error exception to the application for further processing. In this paper we focus on Java Virtual Machines (JVM) and Java applications for exception handling at this level (see Figure 1).

At the application level, JVMs and Java applications are of particular interest because of their large garbage-collected heaps, the virtual machine abstraction presented, and the integrated exception mechanism. Large garbage-collected heaps present a sweet-spot for this research, because the garbage collector itself may uncover many errors as part of the heap sweep during collection. These heaps are also usually larger than explicitly allocated heaps, thereby increasing the probability of a memory error during a sweep.

By presenting an abstraction between the operating system and the applications, the virtual machine makes application-level recovery simpler. Since, the JVM has increased information about the application's status and semantics, such as memory usage, there is an improved chance of recovery. Java's integrated exception handling could allow 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 allow the application to handle the memory error gracefully.

However, 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 towards tackling this potential problem. The major contributions in this paper include: quantifying the memory error consumption and susceptibility rate in the Kaffe JVM and sample Java applications; and, evaluation of extensions to the Kaffe JVM to detect silent data corruption.

The rest of the paper is organized as follows. In Section 2, the paper outlines work related yo 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 corruption are described in Section 4. Section 5 presents the experimental results. Lessons learned are presented in Section 6. The paper ends with recommendations for future work in Section 7 and conclusions for this work in Section 8.

2 Related Work

The effects of and trends for soft-errors were first reported by Ziegler et al. [21, 22], based on field and experimental evidence that alpha particles and cosmic rays were the source of several random system failures. Since then soft errors have become a greater concern because semiconductor susceptibility to these particles increases with technology density and voltage drops.

Availability in computer systems is determined by hardware and software reliability. A high level of 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 S/390 Parallel Sysplex [15] and Tandem's NonStop Himalaya [5].

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 addressed problems of memory errors themselves or taken up software reliability research in general. Examples include Windows 2000 and Linux. They typically rely on fail-over 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 problem of reliability and software recovery [3, 7, 11]. These include techniques such as check-pointing [7] and backward error recovery [3]. Much 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 techniques to study the fault tolerance of UNIX systems. Fine is a set of experimental tools capable of injecting hardware- and software-induced errors into the UNIX kernel and tracing the execution flow and kernel's key variables. Our fault injection work operates at the application level and uses the debugger tool *ptrace* to trace the application's behavior.

Some research has 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 Failures In Time (FIT – errors in one billion hours) when using Single Error Correct-Double Error Detect (SEC-DED) ECC [6]. This is equivalent to around 900 errors in 10,000 machines in 3 years. Tandem [19] estimates that a typical processor's silicon can have a soft-error rate of 4,000 FIT, of which approximately 50% will affect processor logic and 50% will affect 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, HP Labs has studied the future trends of these error rates, their repercussions on processor error handling support, operating system handling/recovery, and application recoverability [12]. This paper reports part of this.

3 Memory Error Susceptibility

Memory errors present themselves in a computer system as either serious exceptions, when detected, or silent data corruption in memory, if undetected. However, in many current Java environments, memory errors will be discovered as silent data corruption since no memory detection or correction hardware is used. In this paper, we concentrate on the analysis and recovery of those corruptions that occur in the application's data area. Errors in the native instruction sequence and 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, suppose an error occurs on an ID string array so that one ID is changed unexpectedly. If this ID is never matched in searches, the error won't lead to any application errors.

Studying the affect of transient memory errors on JVMs and Java applications has many valuable benefits. Most of all, it lets us understand the application behavior under silent data corruption so that we can design efficient software methods to detect silent data corruption. Since it's infeasible to detect all of the errors, our 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 experimental environment used.

3.1 Memory Error Definitions

We refer to the act of an application accessing a memory location containing a soft error as *error*

consumption. We define the memory error consumption rate ($R_{consumption\_rate}$) as the ratio of the 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 the portion of the total error rate that is actually seen by the application, because only errors in those memory locations that are accessed are noticed. The consumption rate is always smaller than one. Thus, our definition of consumption rate is the upper bound on 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 because some high-end operating systems use a memory scrubber to pass over physical memory removing any correctable errors it finds. In the presence of ECC memory, the memory scrubber can clear all correctable errors that exist in memory.

If the error consumption eventually causes the application to crash or to return an erroneous result, we say that the caused an *application error.* Verification of the latter is performed by comparing the result against a known correct result. Lastly, we refer to the *error susceptibility* of a memory region as the likelihood of an application error being caused on error consumption. 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 (as in the previous formula), i.e.,

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

We assume that memory errors are distributed uniformly in the application's total virtual memory area. Since memory errors affect physical memory, this is similar to assuming that the working set fits into physical memory.

3.2 JVM Memory Error Susceptibility

In a JVM, 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 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 memory errors is also interesting.

Since all silent data corruption is not detected by hardware solutions, we need to design a software solution to detect these errors. We propose a simple detection scheme using checksumming of heap objects. Fault injection will be used to evalulate the efficiency of this approach.

3.3 Experimental Setup

We chose Kaffe for experimentation 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, has reasonable performance, and is widely used.

For our experiments, we used Redhat Linux 6.2, running Kaffe 1.0.5 with the "interpreter mode". Since we assume an IA-64 error handling architecture and Kaffe hasn't been ported to IA-64 yet, we used a IA-32 architecture Pentium-III processor based system instead. Where appropriate, we will point out the different memory error implications of using each type of processor.

4 Experiment Methodology

In this section, we first explain the method and setup of the fault injection experiments. 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 consump-

tion, and monitor the application behavior after any 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 IA-32 architecture, there are eight debugging registers that can be used to set data breakpoints. They are identified as DR0 – DR7. DR6 is the breakpoint status register, DR7 is the debug control register, and DR0 – DR3 are used to set the addresses of breakpoints.

For each breakpoint address, the IA-32 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 set only 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.

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 waking the parent process can examine and set the status of the child process with the *ptrace* call.

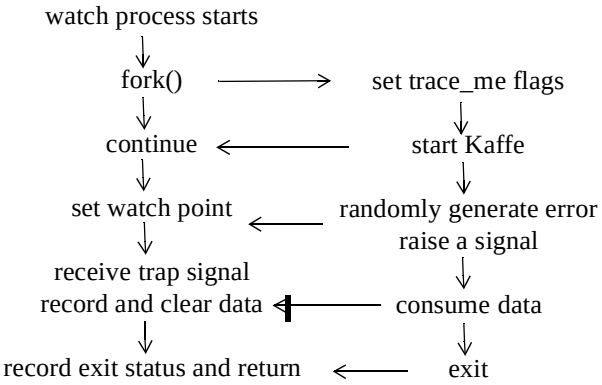

Figure 2 Tracing error consumption using ptrace.

The way we use *ptrace* is illustrated in Figure 2. We modified the Kaffe executive to start the watch (monitor) process first. The watch process uses *fork* to create and run the VM. At certain points of the VM's 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 (because they have the same address space layout, we can obtain the child's data address easily) and sets the appropriate data breakpoint.

After the child process resumes, it may or may 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 instrumented 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 location's 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. For our purpose, the information we record includes the object type, size, and base address.

Next, 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. Kaffe uses the mark and sweep algorithm, which makes this inspection fairly easy because when the GC is running all of the 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 specific group of bytecode operations, such as getfield and putfield. For each of these operations, we add code to do a checksum computation. The heap object management can be modified to store the checksum results.

Space For Checksums

Instead of directly extending Kaffe's object data structure to have extra fields for storing checksum data, we extended 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. It contains information such as block size, garbage collection status, and object type. Two bytes are added for each small object, using one byte for a bit pattern checksum and another for checksum validity. The checksum must be invalidated after native calls because native accesses are not checksummed in our implementation.

For big objects and arrays, it is not efficient to have only one checksum across the whole structure. When one byte in a one-megabyte array is accessed, we don't want to compute a checksum for the whole array. 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 is computed much more efficiently for large objects or arrays.

Checksum Computation

When a Java application is running, objects are accessed when:

- it's created using the *new* operator,
- one of its fields is read or written by the bytecodes *get/putfield, get/putstatic*,
- an entry in an array is read by one of the bytecodes: *iaload, laload, faload, daload, caload, saload, baload* and *aaload*,

- an entry in an array is written by one of the 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 on by some native functions,
- the object is walked by the garbage collector.

In Kaffe, because static fields are class related they are stored within the class objects rather than the data objects. Due to time limitations, we were unable to instrument Kaffe to add checksum protection to the static areas of class objects. Therefore, our results are based only on instrumenting data object accesses.

Using our instrumentation when an object field or an array entry is read by some bytecode, we compute the checksum of the read value with the rest 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 a bytecode, we update its checksum value. For simplicity, in our implementation the checksum is computed by XORing all bytes in the object rather than by a polynomial checksum as used in TCP/IP.

5 Experiment Results

In this section, we present our experimental results for error consumption and silent data corruption detection. 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 from the SPEC JVM98 benchmark suites [18]. We selected four applications from this suite:

- _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 used the medium data configuration – ten percent. With this data size, the experiments finish in a reasonable time, and are large enough to cause the garbage collector to run.

For both static and dynamic areas, we inject 1,000 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 Pentium III 500MHZ platform. The total code size for error injection and tracing is about 470 lines with about 780 lines for memory error detection.

5.1 Memory Error Consumption

This experiment is divided into two parts. In one part, we inject memory errors into the VM's static memory area; in the other part, we inject errors into the object heap. These two areas are used differently by Kaffe. 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 which is walked by the garbage collector when it is started. The heap can have a higher error consumption rate than the static data area because of garbage collection.

Static Memory

The results from injecting errors into the static data area are summarized in Figure 3. In the graph, the mid-gray part comprises those errors that are not consumed by the application even though they are injected; the dark-gray part comprises errors that are consumed by the application but don't cause any application errors, i.e., the application accessed the erroneous data but it still executed correctly; the light-gray part illustrates the number of application errors, in this case, the 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 applica-

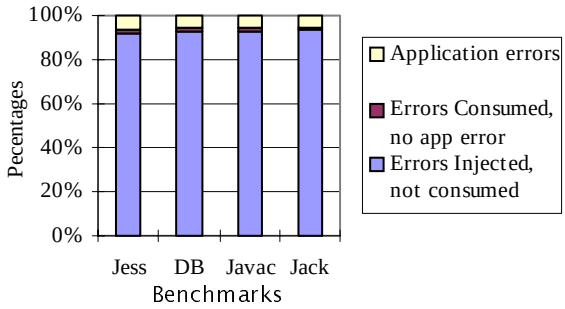

Figure 3 Error consumption in the JVM's static data.

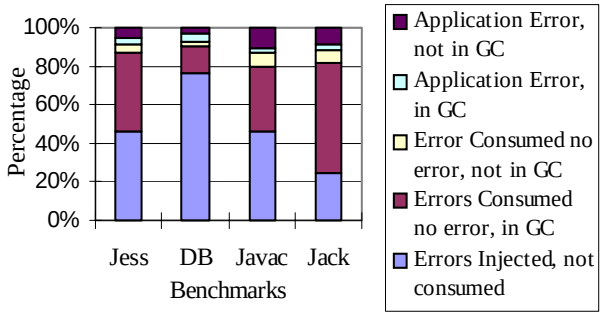

Figure 4 Error Consumption in the JVM's heap region.

tions 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.

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 experiment, we injected errors into the range of virtual addresses the heap occupies. In these experiments, the application heap sizes varied from 5,243KB to 8,397KB (see Table 2).

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 results from our heap injection experiements are summarized in Figure 4 with the appropriate susceptibility rates 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.

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

Table 3 Susceptibility in the Heap

Our first observation 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 reveals that most consumption comes from the garbage collector. Kaffe uses mark and sweep strategies for garbage collection. When collection is started, it touches almost every object in the heap. It is no wonder that it consumes so many errors. If we don't count the errors consumed in the GC, the error consumption rate is about 9% to 22%, which is still higher than in the static data area.

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

Although most of the consumption takes place in the garbage collector, relatively few errors actually cause real problems. The first reason is that the garbage collector only cares about an object's reference field. It won't use other types of fields for computations. For an object reference, it first checks whether it is valid, which masks out most of the possible errors. On average, only 7% of the error consumption in the GC caused application errors. In comparison, 56% of static data error consumption caused application errors.

To further understand the source of application errors, we also collect the object types for the object into which each error is injected. In Figure 5, we show the result for Javac. We distinguish objects, primitive arrays, reference arrays, and areas that are not used. An example of the lat-

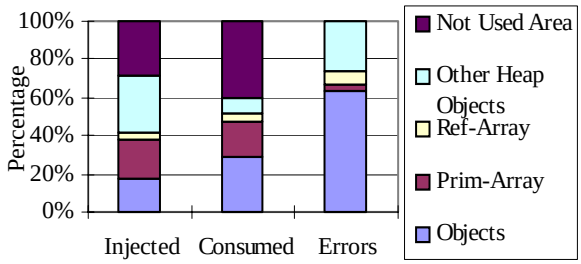

Figure 5 Error Consumption by Object Type.

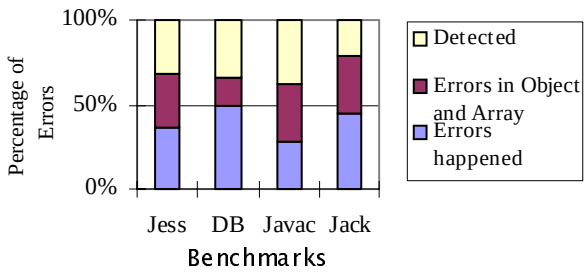

Figure 6 Checksum Detection of Application Errors

ter, are areas that do not belong to any JVM object, such as an object that has been freed by the garbage collector, or a block inside a page that has not been allocated to any object. These results indicate that errors injected into unused parts never caused application errors. However, they may be consumed by overwriting.

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

We can also see that many errors are injected into primitive arrays. This is understandable because user applications tend to store large data sets in arrays. However, because these are large structures containing particular single errors, these errors are less likely to be consumed because 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, reference arrays are much more likely to cause application errors, because a false pointer can easily cause a segmentation fault in the JVM.

Due to the space limitations, details on other error data types is not included here. Briefly, constant fixed objects occupy a large percentage of the "other heap object" part in Figure 5. These objects include data such as bytecodes and the constant pool. In total, these objects occupy between 8% and 30% of the objects types. Since they are read-only objects recovery of these objects types should be straightforward.

5.2 Checksum Silent Data Corruption Detection

To demonstrate the 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 validity rather than update the checksum result, although in the future we will do so.

Another limitation is that we don't compute checksums for large objects, although we do deal with large arrays. We assume that we won't 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. Figure 6 shows the percentage of application errors that can be detected when the error is consumed. The light-gray areas represent errors detected. The dark-gray areas represent those errors that we know took place in objects and arrays and that we could have corrected if we applied checksumming. It has not been applied because the object is too big or was operated on by some native functions that are not easily checksummed. Finally, the mid-gray area comprises the cases where the memory error was not detected and corrected, and caused an application error.

The effectiveness of the detection depends on the nature of the application. If objects and arrays account for most of the actual errors occurring, 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.

The percentage of errors detected by only the current implementation was limited by time constraints. In the future, the implementation can by improved by updating checksums during native function calls and array copies. The technique can also be extended by including more heap objects into the checksum detection, such as constant pools and bytecode sections. Since these heap objects are never changed after they are loaded, the extra checksumming overhead would be small because only checks on read access would be required.

We also compared the relative slowdown of the prototype implementation with the original Kaffe implementation. It is interesting to see the performance overhead induced by the checksum process. We measured the total execution time of the original Kaffe implementation and our prototype implementation. The relative slowdown compared to the original version is shown in Table 4 for each benchmark used.

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

Table 4 VM Slowdown with Detection

6 Lessons Learned

We found that ptrace is a good tool for fault injection experiments. 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 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 to study kernel mode errors.

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

- For the 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 that the heap memory will be the dominant 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 caused 70% of the application errors.
- By adding simple checksums, normally undetected errors can be detected, increasing error coverage by 30-40%.
- Adding checksums clearly comes at a performance cost. Our unoptimized checksum routine adds this functionality for an increase in run time of 32-57%. Optimizing the checksum computation for the platform (maximizing explicit parallelism) or using hardware support for block checksums should help make this more acceptable for comtemporary JIT run-times.
- The coverage of silent data corruption detection should be easy to increase by placing checksums over more object types (e.g., static objects). 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. This assumption should be relaxed for future experimentation.

7 Future Work

Some further work is needed to complete our study of memory failure recoverability at the application level. First, we need to extend and optimize our prototype silent data error corruption implementation to handle other heap objects, including large objects, the constant pool, byte code, etc. Using these extensions, we can expect to achieve a higher error detection rate.

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

Third, it would be interesting to investigate further the 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 provides an elegant exception programming model through the use of `try/catch` blocks [1]. One future path for investigation would be to consider supporting this exception mechanism to signal memory errors to applications interested in providing error recovery or application state tidy-up on exit. Such support may be of great interest to fault tolerant Java applications, Java databases and Java persistent systems.

When a memory error occurs it can either affect the JVM's or the application's integrity. Determining whether the error affected the JVM or the Java application is fairly complex because the JVM's state is stored both inside and outside of the heap. We propose that it would be possible that when errors occur in the JVM's data areas outside the heap, the JVM could throw an asynchronous `UnrecoverableMemoryError` exception. This is similar to the existing `VirtualMachineError` exception. This could allow for cleaner fail-over handling between redundant machines.

Errors in the VM's heap structures are much more serious and difficult to detect. While the sensitive memory is small, errors can seriously affect both the VM and application. To achieve a suitable level of coverage all heap structures would need to be fully checksummed and updated on modifications. However, a similar `UnrecoverableMemoryError` exception could be raised with sufficient detection support.

The majority of memory errors are likely to occur in the state of an object. We propose that these circumstances it may be possible to raise a `MemoryErrorException`. However, a large question with this approach is limiting the scope for handling which the exception has the execution. The depreciated `Thread.stop()` method highlights some of the concerns. Raising a `MemoryErrorException` should not allow the system to leave the state of objects in an undefined state. Nor should it generate an exception the target can't be prepared to handle.

We believe one possible solutions to overcome these problems would be to dispatch `MemoryErrorExceptions` to all dependent threads to allow for informed and safer clean-up from the exception. Since this is an internal VM exception, all threads should be prepared to handle it if they so desire.

However, handling such an exception mechanism is probably too complex to use throughout an application. So it is proposed that to limit the scope to where it is most useful, application programmers could wrap only critical code with such exception handling. Critical sections such as outgoing RPC/RMI access or database accesses would make good candidates since they may hold reproducable transactions and could benefit in improved reliability from this approach. Exceptions occuring at other times can resort to using such exceptions for application clean-up to improve graceful exit/restart when state is lost.

Clearly support for this exception handling is complex and poses challenges in performance, coverage, and support. We would like to see research undertaken to investigate this aspect further.

8 Summary

In this paper, we have described our work in studying the memory error susceptibility of the Kaffe virtual machine using fault injection. We found that for the Kaffe VM and the benchmark applications we ran, that heap objects comprise most of the memory error consumption. We also presented our prototype implementation for detecting silent data corruptions by object checksum. We found that this

simple technique can detect up to nearly 40% of all application errors caused by silent data corruption.

All experiments were executed in Kaffe's interpretive mode. In order to use Kaffe with its superior performance JIT compiler, the JIT would need to be modified to generate 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 has been has reported here, because the same heap management system is used.

While introducing extra overhead of between 32-57% might seem counter to today's JIT research, this overhead represents an upper-bound on performance loss. On the IA-64 architecture, performance can be improved by perhaps four times, because of the ability to use multiple arithmetic units explicitly to parallelize the computation compared to IA-32 architecture processors.

Acknowledgments

We are indebted to Peter Markstein for commenting on the context and presentation of the paper. His 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 Intl. 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 R & D, vol 40, no 1, pp 1-136, Jan. 1996.

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