

Heap Error Detection Using Hardware Monitors

Vincent Weaver
CS717 Final Project

17 December 2004

1 Background

1.1 The Problem

The Computer Engineering community has been creating increasingly complex systems which are far outstripping our ability to understand them. Despite this, users of these machines want autonomic systems that can adapt dynamically to provide greater performance, to note or repair transient faults, to intercept adversarial attacks, and to reduce system management overhead. A large gap remains between what system designers can provide and what users demand.

1.2 Examples

An example of how this could work in practice would be memory performance monitoring. Probes would report symptoms of poor performance, and the goal of a self-aware system would be to discover the cause of those effects. Rather than placing the burden on a user to query hardware performance counting registers (which sample events, but cannot count all of a given type of event) or instrument binaries, a continuously monitored system might employ an unobtrusive background monitor that detects anomalous behavior in some metric. For example, if the cache miss rate suddenly spiked for long periods, additional monitors could be instantiated on the L1-L2 and memory buses to capture individual L1 and L2 misses, and the behavior

could be correlated. This more precise miss characterization helps select an appropriate optimization, such as cache-conscious data placement.

Such a monitors could prove useful in adapting (and thus improving) specific microarchitectural components such as branch prediction, or in improving fault tolerance, balancing thermal load on the chip, or selectively shutting down parts of the chip to conserve power. Current monitoring facilities are ill-equipped to handle such scenarios. Traditional hardware monitors are limited to a fixed set of events, and cannot perform complicated, online analysis. Software must be used to analyze the events, and this comes at the expense of costly and frequent interrupts, or loss of accuracy due to sampling.

1.3 Current Work

Work is underway at Cornell to create a reconfigurable monitoring framework designed to overcome these limitations. The project decouples the data collection from the following analysis. Pervasively deployed, programmable monitor capsules record system events, while user-defined analysis modules, which can be loaded into the respective capsules, provide analysis functionality. Modules are intended to be implemented either in reconfigurable logic or as programs to small, simple microengines within the capsules. Flexibility in deploying the capsules results from the simple assumption: they all observe system events in the form of transactions. Therefore they may be attached to any transaction interface, including memory buses, I/O buses, or network interfaces.

1.4 This Project

For this project I propose to use the described monitoring framework to implement an aspect of fault tolerance within a processing node. Faults in program execution arise from numerous causes. Some are permanent, such as broken hardware, but most are transient, like bit flips caused by cosmic rays or poorly coded applications. If detected in time, these transient errors can be detected and either corrected, or else used to shut down the misbehaving application before serious damage occurs.

Most current research in fault tolerance involves large overheads in either software or hardware. On the software side, catching faults often requires one to massively instrument code. Instrumentation usually requires access to source code, special compilers, or dynamic recompilation systems, all of

which are difficult, slow, and sometimes infeasible. Fault tolerance hardware frequently requires heroic amounts of extra resources. Often an extra CPU, or at least an entire CPU thread, is devoted to the fault detection logic. This requires significant additional power, area, and design and verification time. By using a monitoring subsystem, hopefully much of the speed of a hardware-based fault tolerance mechanism can be gained while avoiding the high overheads of a typical hardware-based scheme.

This project will attempt to detect heap access faults. The heap is the area of memory that programs allocate at runtime. A common programming error is to allocate a region of memory and then accidentally try to read or write data beyond the proper bounds. If one is lucky this will not cause a program error or crash, but often this type of problem will cause data corruption or worse. Another issue is bounds errors like this can sometimes be exploited by malicious users to crash the program or even take over a system.

2 Implementation

2.1 The Idea

As is so often the case, the most practical solution will most likely split the task between hardware and software. A table of valid addresses and sizes will be made at memory allocation time. Since allocations of this type are infrequent, doing this in software should not be a performance bottleneck. The memory allocations can be caught by using library pre-loading to dynamically intercept the function calls. This allows arbitrary programs to be monitored without modification. The table of valid addresses would be uploaded into the monitoring framework's local memory. An instantiated (i.e., programmed) monitoring capsule would then monitor the memory buses to ensure that any heap access was in a valid range.

If more memory ranges are allocated than can be stored in the memory space local to the monitors, then the hardware would know this and call the OS to check for validity. The bounds check would then be done in software. The hope is that such software checks will infrequent enough that, on average, the checking proceeds at near normal speed, and thus imposes minimal time overheads. The checker caches the more frequently used addresses in hardware, exploiting temporal and spatial locality in the application's memory

accesses.

A checker like this can be much more fine grained than the traditional Virtual Memory method that most operating systems use. Those methods can only detect access errors at a page granularity, which is often much bigger than typical memory allocations. Our theoretical monitor can also catch errors — like over-writing of allocation meta-data, and over-writing of padding between allocations — that typical hardware-based memory checking methods cannot detect.

2.2 Actual Implementation

2.2.1 Determining Software Checker Slowdown

The first thing investigated was the performance of a conventional software bounds checker. I used the `valgrind`[5] heap access checking utility on `equake` with the standard “`inp.in`” input. Surprisingly the slowdown was by a factor of 63; runtime went up from 4 minutes to 253 minutes. This shows there is a lot of room for improvement with heap bounds checking.

2.2.2 Obtaining an Allocation and Address Trace

The next step was to find out the feasibility of using the monitoring framework. The biggest potential problem is the limited storage space available in a monitoring unit; before any work is done it must be determined if caching a subset of the memory ranges gives reasonable performance.

To simulate the monitoring setup I decided to use `FIT`[2], a software instrumentation that is very similar to `atom`[1]. `FIT` is open-sourced, and runs on Linux on multiple platforms. In this case I used it on x86 Linux. I used a software-instrumentation tool instead of a simulator like `SimpleScalar`[4] for speed reasons. There is no need for cycle-accurate simulation when just doing a feasibility study.

The monitor will need to be notified whenever an allocation happens, and build a table out of the allocated address ranges. This is much harder than it sounds to simulate. On a final hardware implementation you could just use shared libraries and the `LD_PRELOAD` mechanism to override the standard C-library allocation routines. But `FIT` only works on statically-linked executables. In order to wrap the allocation functions I had to write my own allocation routines that did some custom work before calling the real

routines. I then used the C pre-processor to redefine the allocation routines as mine, requiring a re-compile of the benchmarks against my own allocation libraries.

One thing FIT can do is grab every memory access made by the instrumented executable and pass these out to the user. It has a method of transparently translating the new, instrumented address back to the “original” address an un-instrumented executable would give. This is good, but unfortunately only the instrumented analysis code gets these translated addresses. If you have code such as “pointer=malloc(25);”, if you look at the value of the pointer in the actual code (such as doing a ‘printf(“%p”,pointer);’) you get the modified pointer, not the “original” one you want. There is a method of making FIT use the same pointers throughout, but it requires a kernel patch and is extremely complicated to do.

This is only a problem because there is no good way to use FIT to output the result of an allocation function. The easiest way would simply be to have the custom-allocation library print these out to a file, but in that case the pointers printed would not match the generated address stream for reasons given in the previous paragraph.

The way I worked around this was to have the custom allocation routines call a separate function that simply dereferenced the first byte of the new allocation. I used FIT to add hooks before and after this new function, watching for any heap address. Since the only heap access is the dereference, we know that this address is the first byte of our allocation.

Adding before/after hooks might sound reasonably simple, but the FIT project had not implemented the “find function by name” atom-functionality yet. So I had to add the functionality myself. [I sent a patch to the project, they are good about including patches I send into their distribution].

Another problem is getting the allocated address/size pair out of the instrumented file and into our monitor simulator. There is no easy way to grab the value of a variable from the instrumentation. The address can be obtained via the method described above, but the size cannot. So the address value is sent out with the memory-access trace (suitably prefixed to indicate it is an allocation and not an access) but the size has to be determined separately. In the end I had to use the non-optimal solution of having my allocation routines writing all the sizes out to a file. Thus you run the instrumented file once to get all the sizes, then you have to run it again to generate the traces and the monitoring program reads the proper size in from the file whenever it gets a new allocation.

The final problem is that the `malloc()` series of calls modifies the meta-data before the allocations, because that's how the routines keep track of the allocated space. Unfortunately the memory checker flags these accesses as illegal! To keep this from happening the memory checker is disabled before a call to the libc allocation routines, and re-enabled afterward. This is done using FIT before/after function hooks as described earlier.

2.2.3 Simulating the Monitoring Framework

Using FIT we now have a stream of memory accesses, allocations, allocation sizes, and frees. This is the kind of data the monitoring hardware would be given. So in software I wrote in C a program that would mimic how the monitoring software would react and keep statistics of the results.

The monitoring routine builds up a big binary sort tree of *all* allocations. This is updated at alloc and free time. Also at this time a window of interest is picked around the current heap. Addresses outside this window will be ignored (otherwise stack, text, and data accesses would be seen as illegal heap accesses). This window includes a few hundred bytes on either side of the minimum and maximum allocated areas to track underflow/overflow errors at the extremes of the heap.

The tree of *all* allocations models the tree that would be kept in software by the OS. A separate tree is modeled that corresponds to the one that would be in hardware. This hardware tree should be as small as possible, since the monitors have a very limited size. The hardware tree is also modeled as a binary sort tree, because such an algorithm can be implemented reasonably small and efficiently in hardware.

The hardware tree is identical to the full one until space is exhausted. After the hardware tree is full, no further allocations are added to it at allocation time. The next time there is an address-miss (possible illegal access) the OS is notified. The OS then looks up the address in the full tree. If the address is not there either we have an illegal address and the user is notified. But if the address is in the full full tree, we just had a tree-cache miss.

Current behavior for a tree-cache miss is to calculate the oldest address range in the hardware tree, throw it out, and then add the address range corresponding to the current cache miss. This might not be the optimal replacement algorithm, but it is a reasonable one to do as it can be done all in software and does not require complicated hardware to track least-

recently-used access like a data cache would do.

At the end of the simulation various statistics are displayed, the most interesting one for this feasibility study is the tree-cache hit-rate.

3 Results

3.1 Heap Error Detection

The heap access monitor, or at least the FIT simulation, can successfully detect heap access errors. I wrote a test application that intentionally would overflow an allocated array, and the monitor immediately noticed and reported the error. Note that the underlying OS did *not* notice the problem and did not seg-fault.

A more interesting experiment would be to randomly write all over the allocated area and see how long it would take for the monitor to notice this. Right now the monitor only would notice problems if by chance you try to access the allocation meta-data before the allocation, or else try to access padding beyond the end of the allocation.

3.2 Benchmarks

In Figure 1 we see the hit rate of equake for various table sizes. Note that this is by number of entries, in general the entry will be 64 bits so multiply each value by 8 to get the size in bytes.

The benchmark used was equake from spec2k, using the “train.in” input from the MinneSPEC[3] reduced input sets. The reduced input set was used in the hopes of running faster than the full input set but having similar allocation patters. The graph shows the statistics after 700 million memory accesses (of which approximately 50 million were heap accesses). There is no significance to this number other than that is how far along the simulations were after 12 hours (when this writeup was being written).

Equake does some of the heaviest dynamic allocation of any of the SPEC benchmarks. With the normal “inp.in” standard input set it allocates over 1 million 24-byte regions without freeing any. Most other applications have much less intense allocation behavior; gcc by comparison allocates around 30,000 areas, though of much more varied size and with a lot more “free()” activity

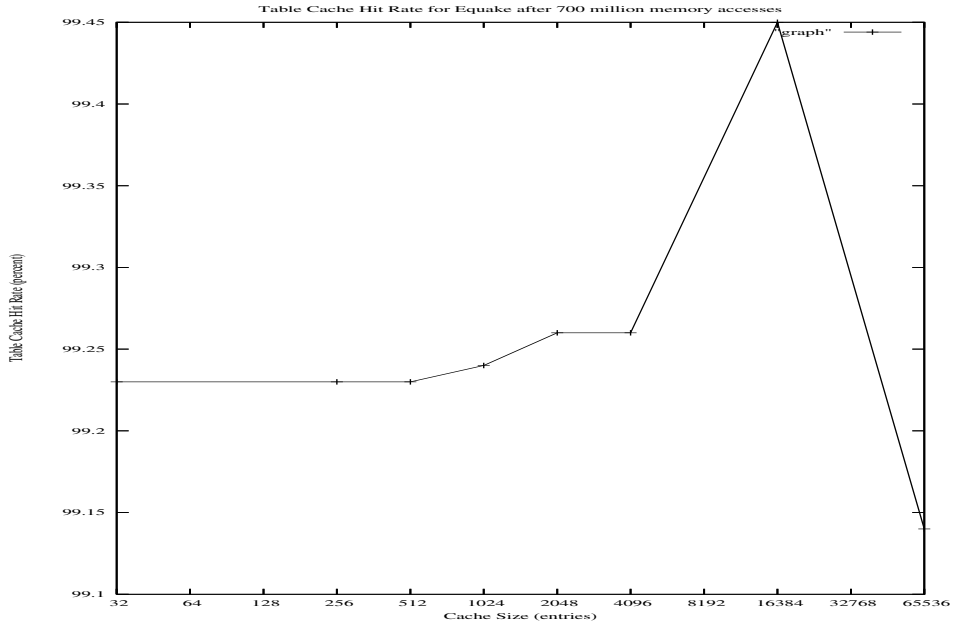


Figure 1: Hit Rate for equake

I originally wanted to run some other benchmarks, especially *gcc* but ran out of time. The problem with *gcc* is that it uses *realloc* which is much harder to implement than *malloc*, *calloc*, or *free* due to the fact that it either grows in place or else does a *free* and *malloc* at the same time. It was difficult just to grab the proper address of an *alloc* or *free*, trying to get both at the same time is a more difficult problem.

3.3 Performance

What we see in Figure 1 is that with a reasonable amount of lookup table we can get over a 99% hit rate. This is an incredibly high number, it shows that *equake* has high locality when operating on allocated data. Remember that a full software-based solution like *valgrind* takes 60 times longer to run, so only having to drop into software mode 1% of the time can give an enormous performance boost.

Definitely more analysis needs to be done. The cache behavior starts out poorly at first, in the first million instructions none of the caches do better than 90% and the large tables do significantly better than the small ones.

But over time once the cache is properly populated things seem to behave much better, at least with equake.

An anomaly is why the hit-rate actually decreases by moving up to a 64k entry cache. I have no good theory as of yet as to why that happens.

4 Future Work

4.1 More Accurate Simulation

Rather than just simulate hit rate, the next step would be to incorporate the algorithm into a cycle-accurate simulator. The Fusion group at Cornell has a modified version of SimpleScalar that was developed for using the monitors in a cache-monitoring system. This simulator could be modified to interact with the routines developed for this project. This would give a much more accurate representation of actual system performance.

The SimpleScalar work is implemented using the Alpha architecture, so some changes would have to be made to my routines before they would work. Some of the results might be different from this project, as the CISC x86 chips this work was done on have very different memory access patterns than a RISC chip like the Alpha.

It would of course be interesting to see the behavior of an actual hardware implementation, but that is unlikely to happen in the near future.

4.2 Better Algorithms

My current implementation runs very slowly, primarily because the global binary search tree is not balanced (at least on a benchmark like equake which generates a huge number of linear addresses, which is worst-case scenario for a search tree). The tree basically becomes a straight linear search which is horribly inefficient. A large speedup could be obtained by re-balancing the tree every time there is a new alloc. Allocations happen infrequently enough that this should give a performance boost.

Another thing that should be investigated is better replacement algorithms. Currently I use a “throw out the oldest” algorithm. This might not be the best algorithm to use in this application, and other replacement methods should be looked into.

Performance could also be sped up if some compression of the table could happen. This could be as simple as limiting the size of the size value, or else using our knowledge of memory layout to reduce the size of the address value. This could allow a bigger table using the same amount of local RAM on the monitoring chip.

4.3 Better Testing

Currently only one benchmark was run, due to time constraints. A full run of at least all the spec2k benchmarks, if not many others, should be run before deciding if this is a loss or a win.

4.4 Fault Injection

Another thing that should be investigated is how well this mechanism can detect hardware or bit-flip errors rather than just poor-programming errors. To do this some manner of random fault injection would have to be utilized. There are various constraints here, one being how best to separate out errors caused by heap corruption versus faults injected elsewhere in the system.

4.5 Lossy Detection

Memory accesses happen often, and the monitoring hardware most likely would not run at full CPU clock speed. This is mitigated by the fact that on modern CPUs the memory bus runs at only a fraction of the CPU clock cycle. Even so, the memory accesses might overwhelm the monitoring unit.

It would be interesting to see what percentage of memory accesses could be “lost” and still maintain a good level of error coverage. Also it should be investigated to see if any loss of addresses could be mitigated by having a pipelined monitor that can process multiple errors at once. This should be possible to do due to the tree nature of the checker.

4.6 Expanded Scope

This method of detecting heap faults can be expanded into other similar error detections, such as stack bounds checking or checking for illegal accesses in the text or data segments. It might be useful to actually monitor the OS,

watching to make sure kernel errors do not cause illegal writes to IO space, potentially crashing the machine before the error can be debugged.

5 Conclusion

This project has shown that there is potentially enough locality in heap accesses to allow good fault detection coverage. This application of the monitoring infrastructure should be investigated in much more detail.

6 Acknowledgments

I would like to thank Greg Bronevetsky and Daniel Marques who helped work out some of the conceptual problems with this approach, and gave me many implementation ideas.

References

- [1] *Atom Reference Manual*. Digital Equipment Corporation. Maynard, Massachusetts: December 1993.
http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15740-f03/public/doc/atom_ref.pdf
- [2] De Bus, Bruno, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, Koen De Bosschere. *The design and implementation of FIT: a flexible instrumentation toolkit* Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering, pp 29-34. June 2004.
<http://www.elis.ugent.be/fit/index.html>
- [3] KleinOowski and David J. Lilja, *MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research*. Computer Architecture Letters, Volume 1, June, 2002.
<http://www-mount.ee.umn.edu/~lilja/spec2000/>
- [4] *SimpleScalar* <http://www.simplescalar.com/>
- [5] *Valgrind*
<http://valgrind.kde.org/>