

HeapMon: a Low Overhead, Automatic, and Programmable Memory Bug Detector *

Rithin Shetty, Mazen Kharbutli, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
{rkshetty,mmkharbu,solihin}@eos.ncsu.edu

Milos Prvulovic
College of Computing
Georgia Institute of Technology
milos@cc.gatech.edu

Abstract

Detection of memory-related bugs is a very important aspect of the software development cycle, yet there are not many reliable and efficient tools available for this purpose. Most of the tools and techniques available have either a high performance overhead or require a high degree of human intervention. This paper presents *HeapMon*, a novel hardware/software approach to detecting memory bugs, such as reads from uninitialized or unallocated memory locations. This new approach does not require human intervention and has only minor storage and execution time overheads.

HeapMon relies on a helper thread that runs on a separate processor in a CMP system. The thread monitors the status of each word on the heap by associating state bits with it. These state bits indicate whether the word is unallocated, allocated but uninitialized, or allocated and initialized. The state bits associated with a word are updated when the word is allocated, initialized, or deallocated. They are checked on reads or writes. Bugs are detected as illegal operations, such as writes to unallocated memory regions and reads from unallocated or uninitialized regions. When a bug is detected, its type, PC, and address are logged to enable developers to precisely pinpoint the bug's nature and location. The hardware support for HeapMon consists of augmenting each cached word with one extra state bit, communication queues between the application thread and the helper thread, and a small private cache for the helper thread. We test the effectiveness of our approach with existing and injected memory bugs. Our experimental results show that HeapMon effectively detects and identifies most forms of heap memory bugs. To study the performance overheads of the new mechanism, we test it on SPEC 2000 benchmarks. Our results show that the overhead of our approach is significantly lower than that imposed by existing tools. The storage overhead is 3.1% of the cache size and 6.2% of the allocated heap memory size. Although architectural support for HeapMon is simple, its execution time overhead is only 8% on average, and less than 26% in the worst case.

1 Introduction

1.1 Motivation

Memory bugs, such as reads from uninitialized memory, reads or writes using dangling pointers, memory leaks, etc., are very common and costly. The National Institute for Standards and Technology (NIST) has recently estimated that inadequate infrastructure for software testing costs the US economy \$22.2 billion annually [16]. It was also estimated that Purify, a

memory bug detection tool, improves debugging productivity by a factor of ten, and saves \$7,000 in development costs per programmer per year [10]. Memory bugs are not easy to find via code inspection because a memory bug may involve several different code fragments which can even be in different files or modules. The compiler is also of little help in finding heap-related memory bugs because it often fails to fully disambiguate pointers [18]. As a result, detection and identification of memory bugs must typically be done at runtime [1, 2, 3, 4, 6, 7, 8, 9, 11, 13, 14, 18]. Unfortunately, the effects of a memory bug may become apparent long after the bug has been triggered. For example, a value read from an uninitialized location can be used in other computation and the error only becomes apparent when the application eventually displays the final result or crashes. Finally, memory bugs can be difficult to identify because their occurrence may depend on the input set or the particular system environment, and may take a long time to manifest. For example, a slow but steady memory leak exhausts the available virtual memory only after a long time, and this time depends on the performance of the system, the configured size of the swap file, etc. Meanwhile, a bug could be introducing a performance penalty, such as excessive page faults due to a memory leak.

Dealing with a software problem typically involves several steps: detection, identification, and repair. *Detection* occurs when the existence of a problem is determined ("something is wrong", e.g. the program crashes). *Identification* involves finding out what is wrong (e.g. the program has a memory leak). Finally, *repair* involves figuring out how to eliminate the problem and changing the code accordingly. It is often easy to *detect* a bug - we can notice that results are wrong or that the application has crashed. However, it may be difficult and time consuming to repeat a bug and identify it. Tools such as Purify [11] can be used during the development cycle to identify memory bugs. However, the high overhead imposed by such tools prevents their use in production environments. Problems that are manifested only with certain input combinations, intermittently, or only in long runs can easily survive debugging and remain in production code, where the existing tools can not be used to find them. Therefore, there is a need for a system can identify memory bugs with very low-overhead and without human intervention. Such a system can remain active even in production runs to identify bugs whenever and wherever they may occur.

1.2 State of the Art

A variety of dynamic bug detectors have previously been proposed, such as Purify [11], Valgrind [14], Intel thread checker [4], DIDUCE [3], Eraser [13], CCured [7], Stack-

*This work is supported in part by the National Science Foundation through grants CCF-0429802, CCF-0347425, and CCF-0429598, and by Georgia Institute of Technology and North Carolina State University.

guard [2], *i*Watcher [18], and others [1, 6, 8, 9]. Many of these dynamic bug checkers add instrumentation into the source or object code. This instrumentation is often added conservatively due to lack of information at instrumentation time. A conservative checker must add instrumentation code wherever a memory reference can not be proved to always be correct - which is very difficult to prove. As a result, checking and monitoring code greatly adds to the execution time, making these dynamic checkers unsuitable for use in the production environment. For example, various dynamic checkers are reported to cause slow-downs of 6 to 30 times [3, 13, 18].

The *Intelligent Watcher (i*Watcher) [18] is an approach based on architectural support to monitor dynamic execution. *i*Watcher associates program-specific monitoring functions with memory locations. *i*Watcher adds an overhead of 66-174% when it monitors 20% of dynamic loads, which is much better than *Purify*. However, *i*Watcher requires much more human intervention in inserting the watch points into the data memory region. The effectiveness of the technique is dependent on how accurately and correctly the user inserts the watchpoints. Another important drawback of *i*Watcher is that it requires customized, fixed-functionality hardware.

1.3 Our HeapMon Approach

This paper presents *HeapMon*, a novel hardware/software approach to detecting memory bugs, such as reads from uninitialized or unallocated memory locations. *HeapMon* does not require human intervention and has only minor storage and execution time overheads. The approach relies on a helper thread that runs on a separate processor in a Chip Multi-Processor (CMP) system. The helper thread monitors the status of each word on the heap by associating state bits with it. These state bits indicate whether the word is unallocated, allocated but uninitialized, or allocated and initialized. The state bits associated with a word are updated when the word is allocated, initialized, or deallocated. They are checked on reads or writes. Bugs are detected as illegal operations, such as writes to unallocated memory regions and reads from unallocated or uninitialized regions. When a bug is detected, its type, PC, and address are logged to enable developers to precisely pinpoint the bug's nature and location.

The main advantages of *HeapMon* are: (1) no human intervention is required, either to insert breakpoints or watchpoints; (2) the bug detector is written in software, hence it is programmable, i.e., new functionality can easily be added; (3) the overhead imposed by this technique is very low, enabling its use in production environments to detect and identify bugs; finally, (4) no compiler support is needed beyond re-linking the application with a new static library, or simply running it with a new dynamically-linked library. Hardware support for *HeapMon* consists of augmenting each cached word with one extra state bit, communication queues between the application thread and the helper thread, and a small private L2 cache for the helper thread. We test the effectiveness of our approach with existing and injected memory bugs. Our experimental results show that *HeapMon* effectively detects and identifies most forms of heap memory bugs. To study the performance overheads of the new mechanism, we test it on SPEC 2000 benchmarks. Our results show that the overhead of our approach is significantly lower than that imposed by existing tools. The storage overhead is 3.1% of the cache size and 6.2% of the allocated heap memory size. Although architectural sup-

port for *HeapMon* is simple, its execution time overhead is only 8% on average, and less than 26% in the worst case.

1.4 Paper Organization

The paper is organized as follows: Section 2 discusses the functionality provided by *HeapMon*, Section 3 presents the architectural support for *HeapMon*, Section 4 details the evaluation setup, Section 5 presents and discusses the evaluation results, Section 6 summarizes our findings and conclusions.

2 HeapMon Functionality

Each application is associated with a *HeapMon* handler, which is implemented as a helper thread. Using a helper thread reduces the amount of dedicated hardware and storage, and reduces the execution time overhead due to *HeapMon*. In applications that do not need *HeapMon* functionality, such as legacy FORTRAN applications without dynamic memory allocation, the processor (or a thread context in an Simultaneous Multi-Threading (SMT) system that would be running *HeapMon* can be used for other tasks, no memory is allocated to keep *HeapMon* state, and there are no *HeapMon* checks.

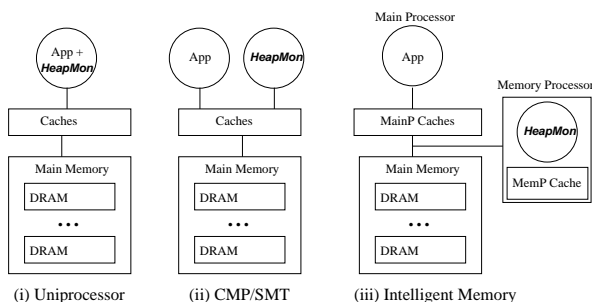


Figure 1. Possible locations for running the *HeapMon* helper thread: on a single processor (i), a separate SMT thread context or CMP core (ii), or a processor near or in the main memory chips (iii).

Figure 1 shows possible locations for running the *HeapMon* helper thread. Each circle in the figure shows a processor, or a thread context in an SMT. *HeapMon* helper thread can run on the same processor on which the application runs (case i), in which case the application and the helper thread switch contexts to share the processor. The helper thread in this case should be implemented as a user-level thread to reduce context switch overheads. Another possible location for the *HeapMon* helper thread is on a separate processor in a CMP system, or a separate thread context in an SMT system (case ii). In this case, context switching is no longer necessary and both threads can be running at the same time. Finally, the helper thread can be run on an intelligent memory, i.e. on a simple processor located near or in the main memory, i.e. (case iii). In this case *HeapMon* does not pollute the main processor's caches and can run more efficiently if the helper thread requires low latency and high bandwidth access to the main memory.

Figure 2 shows the general mechanism of *HeapMon* checking. Each heap memory request proceeds in three steps. First, the request from the main processor is forwarded to the main memory (step 1a) and the *HeapMon* helper thread (step 1b). Requests are events of interest: memory allocation, dealloca-

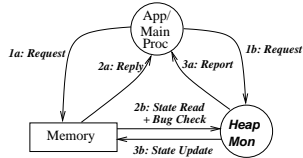


Figure 2. Overall mechanism of the proposed system.

tion, and heap memory accesses. Extra information, such as process id, is piggybacked to the request to give HeapMon enough information to perform the necessary checks. On a read request, the main memory replies with data (step 2a). The tag processor reads the state for the requested word (step 2b) and performs a *bug check* by finding whether the request type is allowed for the current state of the word. The result of the bug check is then reported to the main processor (step 3a) and the state is updated if necessary (step 3b).

The *bug check report* is *positive* if the request is allowed by the current state of the word. The report is *negative* when a bug is detected. Because bug detections are relatively rare, the reporting mechanism should not have a significant impact on the application’s performance. In this paper, we assume that negative reports are not sent to the main processor. Instead, they are logged and later reported to the programmer. However, it is possible to modify the system to create a precise exception in the application thread. In this case, to avoid stalls while waiting for a bug report, the application processor can speculatively consume data replies without waiting for the bug check report. If a negative report does arrive later, a precise exception is still possible using appropriate recovery action, such as a pipeline flush or a rollback.

2.1 Bug Checking Mechanism and Scope

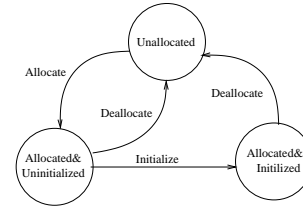
To detect bugs, HeapMon allocates and maintains two bits of state for each word in the heap area. Instead of using special dedicated hardware to store these state bits, HeapMon keeps them in main memory as a software data structure.

The bug detection mechanism in HeapMon is similar to that used in Purify [11]. The possible states and transitions between them are shown in Figure 3. Initially, all words in the heap that are currently not allocated have an *Unallocated* state. When an object is allocated (via `malloc` or an equivalent function), the helper thread changes the state of all words of the object to *Allocated&Uninitialized* state. When a word in the object is written/initialized, the state of the word changes to *Allocated&Initialized*. Finally, when an object is deallocated (via `free` or equivalent functions), the state of each of its words is changed back to *Unallocated*.

To perform these state transitions, the HeapMon helper thread must be notified of two types of events. The first type of events includes memory allocation and deallocation, for which notifications are sent to HeapMon by a modified memory allocation and deallocation library. With this approach, no programmer’s intervention is needed beyond re-linking the application to the modified memory management library. Even this can be avoided if a dynamically-linked library is used. The second type of events are reads and writes to the heap by the application’s processor. They occur very frequently, so we propose a filtering mechanism to reduce the frequency of HeapMon notifications. This filtering mechanism, described in Section 3, exploits the fact that all read/write accesses to words which are

already in the *Allocated&Initialized* state will result in positive reports. Since most words will be used many times after being allocated and initialized, this filtering can reduce the number of bug checks significantly.

A final type of events of interest are changes in the size of the heap region. When the memory manager in an application runs out of heap space, it may request a heap size increase from the OS. At that time, the HeapMon helper thread must follow suit by allocating new state bits for new words in the heap region and initializing these state bits to *Unallocated*. These events are infrequent and should not noticeably affect performance.



Current State	Allowed Requests	Bugs Found On
Unallocated	Allocation	Rd/Wr/Dealloc
Allocated&Uninitialized	Wr/Dealloc	Rd/Alloc
Allocated&Initialized	Rd/Wr/Dealloc	Alloc

Figure 3. State transition diagram for each word, and a table showing the allowed requests and detectable bugs for each state.

We now discuss the bugs that can be detected by keeping the state of each heap memory word. The table in Figure 3 shows which requests are allowed for each current state. By comparing the type of the request with the allowed requests for the current state of the word, memory bugs can be detected.

The bug checking conditions shown in Figure 3 can detect many types of memory-related bugs. For example, access to a dangling pointer usually occurs due to not updating the pointer when the object being pointed to has been deallocated. Figure 4a shows a code section that illustrates this bug. In this code, a node pointed by `nodeToDelete` is removed from a linked list structure and deallocated. Elsewhere in the same application, `currentNode` is dereferenced. The two pointers may never be equal (they point to different locations) in test runs of the program. If, however, certain input combinations in a production environment result in `currentNode` and `pNodeToDelete` that point to the same location, dereferencing `currentNode` after the node’s memory is deallocated is a bug. This bug is detected by HeapMon because the `free` function changes the state of the word to *Unallocated* and any subsequent access to this word is detected as bug. This example also illustrates the importance of architectural support for low-overhead bug detection: certain bugs may be manifested only with some input combinations that may be hard to foresee.

Another class of bugs detected by HeapMon are reads to uninitialized structures on the heap. Figure 4b shows a code section that illustrates this bug. When the node `newNode` is allocated, its `next` field is left uninitialized. Later in the code, the programmer incorrectly assumes that `newNode->next` should either have a `NULL` value or point to a location that corresponds to another node. However, `newNode->next` has actually not been initialized. This type of bug often arises when the programmers forget the assumptions made in parts of the code responsible for setting up the initial values of a variable. This

```

...
if (pNodeToDelete->data == mydata) {
    unlink(pNodeToDelete, list);
    free(pNodeToDelete);
}
...
x = currentNode->data;
                                (a)

typedef struct{
    int data;
    node* next;
} node;
...
node* newNode = malloc(sizeof(node));
newNode->data = x;
...
if (newNode->next == NULL) {
    newNode->next = head;
    head = newNode;
} else
    mergeList(newNode,head);
                                (b)

```

Figure 4. Bug example where there is an access to an unallocated location (a), and a read to an uninitialized location (b).

situation can easily arise when multiple code developers are involved, or when the code is not written in the same time frame. HeapMon detects this bug by recording the fact that the pointer has not been initialized (its state is *Allocated&Uninitialized*). When a read is attempted, the bug is detected.

Yet another class of bugs detected is doubly deallocating an object. Again, HeapMon detects it because the first deallocation instance changes the state to *Unallocated*, resulting in detecting the second deallocation instance as a bug. Finally, a *memory leak* can be detected if, at the end of program execution, there are words in the heap region that are still in one of the *Allocated* states.

3 Architecture Support

This section describes architectural mechanisms needed to implement the proposed scheme, as well as some optimizations we use to improve its performance.

3.1 Bug Check Filtering

Performing a bug check for every heap memory access can be costly due to the high frequency of such accesses, especially since bug checks are performed in software. To reduce the number of checks, we keep *Bug Report* (BRP) bits as part of the cache state in the application’s processor. Every cached word has a BRP bit associated with it. This bit indicates whether or not an access to the word should result in a bug check request to HeapMon. If the BRP bit is ‘1’ for a cached word, any read or write access to that word can proceed without a bug check. Otherwise, the access results in a request for bug check to the HeapMon helper thread. When a cache line is replaced, its BRP bits are discarded. On a cache miss, a bug check request is generated to the HeapMon helper thread, which returns the bug check report, as well as the BRP bits for each word in the line, which are placed in the cache together with the corresponding line.

In principle, the HeapMon helper thread returns a BRP bit of 1 only when it is sure that any subsequent bug checks to the word will generate a positive report, i.e. no bug will be detected. The only state of a word that allows both reads and writes is *Allocated&Initialized*, therefore HeapMon returns a BRP bit of 1 for words in that state. The other two states require HeapMon activity. Read or write accesses to *Unallocated* words are bugs and should be detected, so the BRP bit remains 0 for words in this state. When the state of the word is *Allocated&Uninitialized*, a read to it is a bug. A write to such a word is an initialization, and should update the state to *Allocated&Initialized*. Therefore, the HeapMon helper thread must return the BRP bit of 0 for words in the *Allocated&Uninitialized* state.

In a bug-free application, the fraction of heap accesses whose bug checks can be eliminated is roughly equal to the global hit rate (for heap data) of the lowest level cache. For example, on a system with two cache levels, it equals to the number of L1 and L2 hits divided by the total number of accesses (to the heap region). This is because a bug check request is only initiated when there is an L2 cache miss and on initialization writes, which are usually cache misses. Since most words will be used many times after being allocated and initialized, this filtering can reduce the number of bug checks significantly. However, the effectiveness of the filtering will depend on the application’s L2 cache miss rate, and traditional techniques that reduce the number of L2 cache misses, such as victim caches, higher associativity, etc., will also reduce the number of bug check requests.

There are several other implementation details. First, when the state of a word changes, we assume a hardware support that enables HeapMon to send a command that updates the corresponding BRP bits in the cache. Second, BRP bits only filter read and write accesses, and do not filter heap allocation and deallocation requests. Third, in memory locations that HeapMon is not interested in, such as code, global data, and stack regions, the BRP bit is always 1.

3.2 HeapMon Helper Thread Operation

Figure 5 shows the operation of the HeapMon helper thread. The figure shows the application’s processor, the HeapMon processor, and the hardware communication interface between them.

Communication Buffer. While communication between the two processors can also be implemented completely in software through shared memory, considering the frequency of bug check request, this may add a significant overhead in the application’s thread, as well as add significant latency to the time needed to perform the check. For this reason, we use a hardware 64-entry FIFO *Request Queue* to buffer bug check requests and an 8-entry *Report Queue* to buffer bug check reports¹. The figure also shows the BRP bits in L1 and L2 caches of the application’s processor. In a symmetric system (e.g. a chip-multiprocessor), any processor could be a HeapMon processor or an application processor, so each processor is ex-

¹Since most bug check requests are due to L2 cache misses, the Request Queue is designed to have slightly more entries than the maximum number of outstanding misses that the L2 cache can support. This way the application processor almost never blocks due to the Request Queue being full. The Report Queue does not need as many entries because they are handled by hardware.

tended with a request queue, a report queue, and BRP bits in L1 and L2 caches. For illustration purposes, the figure shows each processor with a private L2 cache, although physically the processors may share a single L2 cache.

In this first design we assume that a processor (or a thread context in an SMT processor) is dedicated to running the HeapMon thread. However, as shown in the evaluation, the utilization of the HeapMon processor is typically very low, making it possible to interleave its execution with other tasks.

Bug Check Requests. When the application’s processor calls an allocation or deallocation function, the function uses special instructions to put the request in the *Request Queue* (circle 1a in Figure 5), a FIFO buffer. Bug check requests are also automatically placed into this queue by the hardware of cache misses and when there is a cache hit to a word with a 0 BRP bit (circle 1b in Figure 5). A request contains the application’s process id, PC of the requesting instruction, request type, virtual starting address, and request size. The application thread and the HeapMon thread run in the same address space, so heap addresses passed to the HeapMon thread are virtual addresses. Using a virtual address, in addition to avoiding security concerns, has two other advantages. The first advantage is that the index computation to locate the state bits is simpler, because the application’s heap is a contiguous range of virtual addresses instead of a scattered collection of physical pages. The second advantage of using virtual addresses is that normal paging mechanisms can be used for both the application’s memory and the memory used by HeapMon. Finally, it should be noted that this scheme is based on the assumption that HeapMon has knowledge of the starting address of the heap region. However, this information can be easily obtained from the library which manages the heap region. A *Heap Filter* contains the start and the end address of the heap region. The filter ensures that only heap memory requests are placed in the queue. HeapMon does not keep any state bits for non-heap data.

After placing requests in the Request Queue, the application thread immediately resumes execution. This overlaps HeapMon checks with execution of the application and hides the latency of bug detection. Our current version of HeapMon only logs the requests for which a negative report has been generated. Therefore, an error log entry only contains information present in the request and does not include, for example, the snapshot of the stack, although this snapshot could provide a more detailed information for debugging. However, we conservatively assume that the processor can not consume data loaded from memory until both the data and the report (BRP bits) are available in the processor’s cache. Therefore, HeapMon causes stalls in the application processor in only two situations: (1) read misses can be delayed due to the HeapMon’s latency of checking state bits and generating BRP bits, and (2) the Request Queue can become full.

Maintaining Heap Request Ordering. To avoid detection of spurious bugs and failing to detect some true bugs, HeapMon assumes that program ordering of heap memory requests is preserved in the Request Queue. To preserve the ordering, we use memory fences in the modified memory management libraries.

HeapMon Helper Thread Invocation and Suspension. If the HeapMon thread is inactive (e.g. its processor is in a sleep mode), it is activated by the *Interface Logic* when its queue receives a request (circle 2 in Figure 5). A special instruction is then used to read the parameters of the request into registers

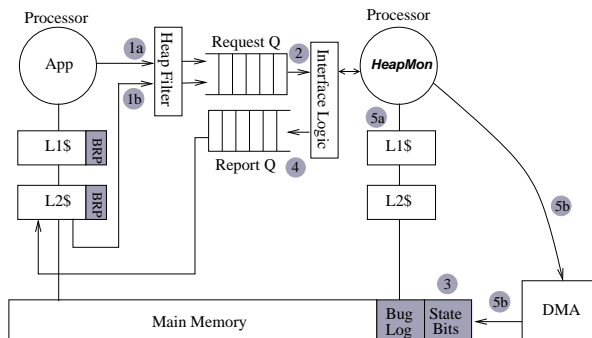


Figure 5. HeapMon operation and communication. For illustration purposes, the figure shows each processor with a private L2 cache, although physically the processors may share a single L2 cache.

on the HeapMon processor, and the HeapMon thread proceeds with the bug check. Once the check is complete and the HeapMon thread is ready to process a new request, it first attempts to get the next request from the queue. If no request is available, the thread becomes inactive (e.g. puts its processor into sleep mode). It would also be possible to busy-wait by polling the queue until a request arrives. However, busy-waiting would needlessly expend power and increase heat dissipation problems. Such energy efficiency concerns are beyond the scope of this paper. Regardless, we use the sleep/wake model in our simulation experiments because a busy-waiting processor slows down simulation considerably.

Reading and Maintaining the State Bits. When the helper thread obtains a bug check request, it reads the state bits for the requested words. HeapMon stores these state bits in memory, in a *software data structure*. This data structure maintains two state bits per word in the application’s heap area. With two bits of state per word, the storage overhead of using HeapMon is 6.25% of the total allocated memory in the heap area. HeapMon allocates the state bit structure in large chunks incrementally and dynamically. It detects the application’s request to increase its heap segment size by intercepting the application’s call to `brk` and `sbrk`.

When HeapMon receives a bug check request, it uses simple indexing to locate the state bits for the requested address. Then it reads the state of the word from its data structure (circle 4 in Figure 5) and, based on the type of the request and the state of the word, determines whether or not the access is a bug. If the access is not a bug, HeapMon puts the BRP bits to the Report Queue (circle 4 in Figure 5). These bits will be placed alongside the cache line in the application processor’s cache. Finally, if the access results in a state change, the helper thread performs a state change (circle 5a in Figure 5) and sends a signal to update the BRP bits of the application processor caches. Since the updated state bits were recently read by the helper thread, they are most likely still resident in the cache of the HeapMon processor.

3.3 HeapMon Optimizations

DMA Bulk State Update. In some cases, many state bits have to be updated at once. This is especially the case for memory allocation/deallocation requests when large blocks of memory may be affected. If the helper thread performs the state changes

completely in software, it may have a large latency. To avoid that, when the helper thread needs to initialize/update more than one page of state bits, the HeapMon helper thread programs the DMA to do that (circle 5b in Figure 5) [12]. It first specifies the physical pages (or virtual pages with some DMAs) that should be mapped as the DMA’s memory. Then it supplies a small *write string pattern*, representing the state that should be written for the entire pages, to the DMA, and suspends itself. The DMA repeatedly transfers the write string pattern and writes it directly to the memory. Once the transfer is completed, it sends an interrupt to wake up the HeapMon helper thread. This functionality already exists since many systems already have DMAs. We model DMA overheads in the evaluation.

Separate L2 Cache. Sharing the L2 cache between the HeapMon processor and the application processor may produce an undesirable cache contention that invariably slows down the application. Unfortunately, the impact of cache contention is easily magnified due to the inter-dependencies between the application and the HeapMon helper thread. For example, if due to cache contention the application suffers extra cache misses, the helper thread will have to perform more bug checks, and since the application needs to wait for BRP bits to be produced, it is slowed down further, and so on. Therefore, we also evaluate the case where the HeapMon processor has a small private L2 cache.

3.4 Limitations of HeapMon

Since HeapMon only monitors heap memory requests, it cannot detect bugs in stack and global data segments. However, this is not a fundamental limitation. Since the bug detection is implemented in software, the helper thread can be modified to also track other segments. This functionality can be useful to detect other classes of bugs, such as accessing non-initialized stack variables.

It should also be noted that currently HeapMon does not detect buffer overflow bugs. However, it can be easily and directly enhanced to detect them, without extra hardware support. For example, the memory allocation and deallocation routine can be modified to leave a small unallocated block between two consecutive allocated regions. A buffer overflow would result in an access to this unallocated block and would be detected by HeapMon. In addition to helping find programming errors, detection of buffer overflow in production code would also boost security by detecting attacks that exploit such errors.

Finally, keeping one state per 32-bit word imposes a limitation in tracking byte-sized locations. For example, a read to an uninitialized 1-byte `char` may not be detected if the neighboring byte has been initialized. However, since most memory bugs involve uninitialized pointers instead of smaller structures, and that the size of a pointer is 32 bits in our platform, we can still detect most uninitialized pointer bugs.

4 Evaluation Setup

Applications.

To evaluate HeapMon, we use 14 applications, mostly from Spec2000 [15]. The applications, their sources, input sets, L1 and L2 cache miss rates, number of allocations, average allocation size, and the percentage of memory accesses that go to

the heap region are shown in Table 1. We omitted FORTRAN benchmarks since they do not use dynamic memory allocation, and C++ benchmarks due to limitations of our infrastructure.

Simulation Environment.

The evaluation is performed using SESC, a detailed cycle-accurate execution-driven simulator capable of simulating CMP and intelligent memory architectures. This simulator has recently been developed at the University of Illinois and is based on MINT [17] and other published work [5]. Table 2 shows the parameters used for each component of the architecture. Each CMP core is an aggressive out-of-order superscalar processor with private L1 instruction and data caches. In CMPs, the L2 cache can be configured to be per-processor private or shared among processors. The memory hierarchy below the L2 cache is always shared by all CMP cores.

Four different configurations are used to evaluate the performance of HeapMon. *IM* represents an intelligent memory with the memory processor integrated into the memory controller. It has half the memory latency of the main processor, but does not have any bandwidth advantage due to its location outside the DRAM chips. *CMP+L2shr* is a dual-core CMP with a shared L2 cache. *CMP+L2prv* is a dual-core CMP with a private L2 cache for each processor. However, we vary the HeapMon processor’s L2 cache from small (64KB) to as large as the application processor’s L2 cache (1MB). None of the configurations uses DMA support, except *CMP+enh*, which uses DMA to perform bulk state updates, and includes a 128KB private L2 cache for the HeapMon helper thread.

Bug Injection. Since Spec2K applications have been heavily debugged, at least for the supplied input sets, we inject memory bugs to evaluate whether HeapMon is able to detect them. We inject three types of bugs. *alloc-bug* is an injected bug where at random memory allocation requests, we reduce the allocation size by 8 bytes. *dealloc-bug* is an injected bug where at random time, we deallocate the randomly selected heap objects. Finally, *noinit-bug* is an injected bug where we skip random initialization writes to a location. No memory leak bugs were injected to the benchmarks as they already existed in most cases. All benchmarks were run from start to completion without skipping or fast-forwarding through any instructions.

5 Evaluation Results

In this section we evaluate the execution time overhead of HeapMon (Section 5.1) and its bug detection capability (Section 5.2). Finally, we characterize HeapMon’s performance (Section 5.3).

5.1 Performance Overhead

To be useful in production runs, a bug detection mechanism must not slow down programs too much. This is a limitation in existing tools, where various dynamic checkers are reported to slow down programs between 6 to 30 times [3, 13, 18]. By using thread level speculation and custom hardware, iWatcher gets the overhead down to the 66-174% range, when it monitors 20% of all dynamic loads. However, even this lower overhead may still be too much to allow use in production runs. We will now discuss the execution time overhead of HeapMon.

Figure 6 shows the execution time overhead of HeapMon for various machine configurations. The figure shows four bars

Benchmark	Source	Input Set	L1 Cache Miss Rate	L2 Cache Miss Rate	Num Alloc.	Avg. Alloc. Size (Bytes)	% Heap Accesses
ammp	Specfp2000	test	17.90%	0.43%	34,764	2,676	36.52%
art	Specfp2000	test	8.29%	5.42%	30,488	60	27.75%
bzip2	SpecInt2000	test	0.85%	1.06%	10	1,138,631	3.95%
crafty	SpecInt2000	test	2.95%	0.16%	40	21,001	0.42%
equake	Specfp2000	test	1.01%	48.42%	316,851	25	34.63%
gcc	SpecInt2000	test	1.38%	0.78%	4,302	3,509	18.84%
gzip	SpecInt2000	test	2.91%	2.57%	243	27,944	4.26%
mcf	SpecInt2000	test	8.66%	13.51%	6	12,074,701	24.15%
mesa	Specfp2000	test	0.08%	37.21%	66	307,088	15.87%
mst	Olden	1024 nodes	1.75%	36.22%	419	32,487	41.6%
perlbmk	SpecInt2000	test	1.19%	13.98%	421	19,954	37.37%
twolf	SpecInt2000	test	1.10%	0.08%	9,400	56	28.62%
vortex	SpecInt2000	test	0.91%	6.15%	186,436	357	19.42%
vpr	SpecInt2000	test	2.29%	0.01%	1,589	106	25.55%

Table 1. Applications used in our evaluation.

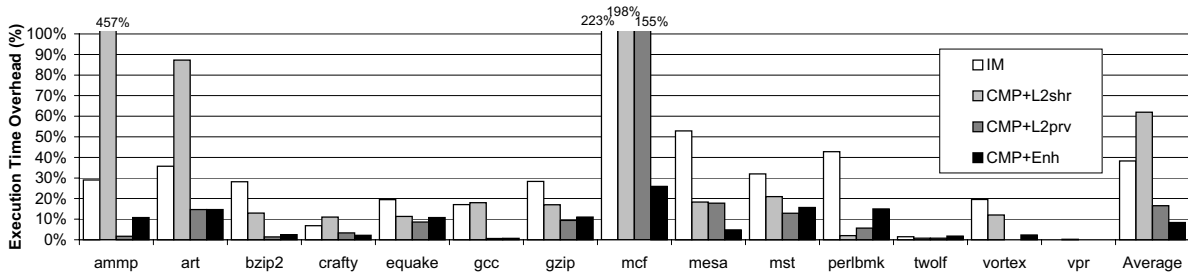


Figure 6. Overhead imposed by HeapMon on various machine configurations.

per application and their average, where each bar represents the execution time overhead for a different machine configuration: Intelligent Memory (*IM*), CMP with a shared 1-MB L2 cache (*CMP+L2shr*), CMP with a per-processor private 1-MB L2 cache (*CMP+L2prv*), and CMP with enhancements (*CMP+enh*) which include a 128-KB private L2 cache for the HeapMon processor, use of DMA for bulk state updates, and using 8-byte state update granularity when possible.

CMP+enh is clearly the best scheme, delivering an average execution time overhead of only 8%, with the worst case of slightly less than 26% (*mcf*). *CMP+L2prv*, followed by *IM* and *CMP+L2shr* have higher average execution overheads (17%, 38%, and 62%, respectively). Comparing results from different configurations gives us insight into the execution overhead. For example, the difference in performance overheads of *CMP+L2shr* and *CMP+L2prv* comes from the fact that in *CMP+L2prv*, the application and the helper thread do not contend for the L2 cache. The figure shows that cache contention has a very strong effect on overhead: the average overhead in *CMP+L2shr* is considerably higher than in *CMP+L2prv*. In three applications, the difference in execution overhead is very large: 457% vs. 2% in *ammp*, 87% vs. 15% in *art*, and 198% vs. 155% in *mcf*. The reason for such large differences is that the impact of cache contention can easily be magnified due to the inter-dependencies between the application and the HeapMon helper thread. For example, if due to cache contention the application suffers extra cache misses, the helper thread is also slower because it has to perform more bug checks, which in turn stalls the application’s processor more because read misses from the processor wait for both the data and the BRP bits to be produced. However, note that even in *CMP+L2prv*, *mcf* still suffers a very large overhead (155%). Therefore, there is still at least one performance problem in addition to cache

contention. We found that this additional source of overhead is due to large memory allocations which occupy the helper thread for a long time and stall the application’s processor significantly (see Section 5.3 for a detailed analysis).

Although both *IM* and *CMP+L2prv* configurations avoid pollution of the application processor’s L2 cache, *IM* does not perform as well as the *CMP+L2prv* configuration. This is mainly because the memory processor in *IM* is a narrow-issue processor without an L2 cache. Furthermore, the memory processor’s memory access latency advantage is not fully realized because state bits are stored in a very compact form – each cache line can store 512 state bits (enough for 256 words). Lacking enough spatial locality in the application’s thread, the L1 cache of the *IM* processor is not used efficiently (many of the cached bits are not used). With no L2 cache, no bandwidth advantage due to its location outside the DRAM chips, and inefficiently used L1 cache, the *IM* processor performs worse than a more powerful CMP processor with an L2 cache. Therefore, this kind of *IM* is not a good choice of processor to run the HeapMon thread.

Finally, HeapMon characterization in Section 5.3 helps us to find optimizations that are useful for HeapMon, resulting in the *CMP+enh* configuration that has a very low execution overhead across all the benchmarks tested.

5.2 Bug Detection Capability

We now present an evaluation of HeapMon’s bug detection capability. We ran the benchmarks through HeapMon and only found memory leak bugs. This is not surprising since SPEC2K benchmarks have existed for years and have gone through rigorous debugging and testing. Memory allocation bugs or reads

System Parameters

Each Main Processor (Core)	4 GHz, 6-way OOO, 248-entry ROB Int, fp, ld/st FUs: 4, 4, 4 Branch penalty: 17 cycles
Main Processor (Memory)	IL1: WB, LRU, 16KB, 2-way, 64B line, RT: 3 cyc DL1: WB, LRU, 16KB, 2-way, 64B line, RT: 3 cyc L2: WB, LRU, 1MB, 8-way, 64-B line, RT: 13 cyc Outstanding ld/st: 24/24 (DL1), 48/48 (L2). Mem bus: split-trans., 1GHz, 64-bit, 8 GB/sec peak RT memory latency: 400 cyc DMA (optional): 2000 cyc per 4-Kbyte page
Memory Processor (Core)	4 GHz 2-way OOO, 104-entry ROB Int, fp, ld/st FUs: 2, 2, 2 Branch penalty: 17 cycles
Memory Processor (Memory)	IL1: WB, LRU, 16KB, 2-way, 64B line, RT: 3 cyc DL1: WB, LRU, 16KB, 2-way, 64B line, RT: 3 cyc Outstanding ld/st: 24/24 (DL1), no L2 cache Uses the same memory bus with the Main Proc RT memory latency: 200 cyc

Configurations

IM	1 Main Proc with L2 + 1 Mem Proc without L2
CMP+L2shr	2 Main Proc cores, one L2 shared by both procs
CMP+L2prv	2 Main Proc cores, App proc. with 1MB L2 cache HeapMon proc. with various L2 sizes
CMP+enh	2 Main Proc cores, App proc. with 1MB L2 cache HeapMon proc. with 128KB L2 (9-cyc RT) DMA is used for bulk state updates

Table 2. Parameters and configurations of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip latency from the processor.

to uninitialized location most likely result in crash or wrong results, and therefore they would have been removed after rigorous testing, at least for the input sets provided with the benchmarks. What is surprising, however, is the extent of memory leak problem in these heavily tested SPEC2K benchmarks.

Table 3 shows, for each benchmark, how many memory allocations are made, how many of them are not deallocated (both as number and as a percentage of allocations), and the total leak (number of bytes that remain allocated at the end of the program execution). The table shows that, except for *mcf*, every one of the benchmarks has a leak. In fact, three benchmarks (*bzip2*, *equake*, and *mst*) never perform any deallocation. Three other benchmarks (*ammp*, *art*, and *crafty*) deallocate fewer than 5% of their allocations. This prevalence of memory leaks is due to several factors. First, they are hard to detect because, with fixed and relatively small inputs used for benchmarking, the system never runs out of memory due to these leaks. Second, developers tend to be conservative in deallocating heap objects, for fear that doing so will result in dangling pointers or deallocating an object multiple times, which could crash the application. Third, developers may assume that their code would not be re-used and hence choose not to de-allocate pointers at the end of the application’s execution. Finally, developers often work with tight deadlines and are not giving a priority to resolving bugs, such as memory leaks, which only produce intermittent or subtle effects.

Benchmark	Allocation	Not Deallocated	(%)	Total Leak (Bytes)
ammp	34,764	34,762	99.9%	13,263,552
art	30,488	30,487	99.9%	1,824,544
bzip2	10	10	100%	13,662,920
crafty	40	38	95%	872,824
equake	316,851	316,851	100%	8,190,248
gcc	4,302	1,178	27.4%	2,974,624
gzip	243	4	1.6%	6,600,704
mcf	6	0	0%	0
mesa	66	6	9.1%	17,232
mst	419	419	100%	13,676,552
perlbmk	421	325	77.2%	4,233,160
twolf	9,400	1,407	15%	51,728
vortex	186,436	40,600	21.8%	16,001,384
vpr	1,589	48	3%	69,352

Table 3. Detected memory leak in the benchmarks tested.

In order to evaluate HeapMon’s capability of detecting other bugs, we inject bugs into selected SPEC 2000 benchmark applications. We use the data in Table 1 to choose a representative application that has few but very large allocations (*mcf*), few and small allocations (*crafty*), many small allocations (*art*), and a very high number of allocations (*equake*). As mentioned in Section 4, we introduce three types of bugs into the benchmarks: *alloc-bug* reduces randomly selected memory allocation requests by 8 bytes, *dealloc-bug* deallocates some randomly selected heap objects, and *noinit-bug* removes randomly selected initialization writes. We inject multiple bugs on each run. When a bug is detected, we reverse the impact of the bug such that it does not crash the application, allowing the HeapMon to detect other bugs that occur later.

Benchmark	<i>alloc-bug</i>	<i>dealloc-bug</i>	<i>noinit-bug</i>
art	100% (of 20)	100% (of 5)	100% (of 20)
crafty	61% (of 13)	100% (of 4)	100% (of 20)
equake	100% (of 20)	100% (of 4)	100% (of 20)
mcf	100% (of 3)	100% (of 3)	100% (of 20)

Table 4. Bug detection capability of HeapMon, showing how many number of bugs of each type is injected, and the percentage of them that are detected.

Table 4 shows the percentage of bugs detected by HeapMon for each type of injected bugs. HeapMon detected all injected bugs except five *alloc-bugs* in *crafty*. The reason for this is that the *alloc-bugs* reduce the allocation size by 8 bytes from heap objects that are used as string character buffers. These buffers have a fixed allocated size, but unless they store long strings, most of the bytes at the end of the allocated region will not be accessed, at least in the input set that we use. This illustrates the difficulty of detecting memory bugs because they only occur with some input combinations that may occur only in production environments that stress the application to its limits. This emphasizes the importance of dynamic bug detection that has low enough overhead to be used in production environments.

5.3 HeapMon Characterization

This subsection presents characterization results of the HeapMon helper thread, running on the *Cmp+L2shr* configuration. Figure 7 shows HeapMon’s execution time, broken down into the time it is busy servicing allocation requests (*Alloc*), deallo-

cation requests (*Dealloc*), as well as read requests (*Read*) and write requests (*Write*). Each bar is normalized to the application's execution time. Therefore, the height of the entire bar for each application shows the fraction of time the HeapMon helper thread is busy servicing requests.

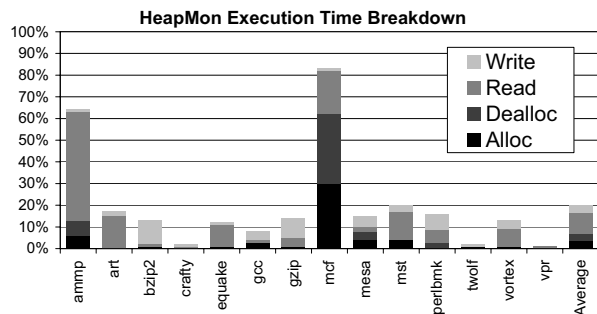


Figure 7. Breakdown of the HeapMon's execution time for servicing each request type.

The figure shows that for all but two applications, the HeapMon processor is busy for less than 20% of the time. This illustrates that, although HeapMon helper thread runs on a separate CMP processor, it does not occupy the processor all the time. It can interleave its execution with other jobs, if the system aims to maximize its throughput. However, it has to remain a high priority thread because the application's execution time overhead is highly dependent on the helper thread's service time.

For two applications (*ammp* and *mcf*), HeapMon helper thread is busy 65% and 82% of the time, respectively. In *mcf*, most of the helper thread's time is spent on servicing allocation and deallocation requests, which is why in Figure 6, the execution time overhead for *mcf* is still very large even when we use private L2 caches in *CMP+L2prv* configuration. Upon closer examination, it turns out that *mcf* has allocations that are very large, which occupy the handler for a long time. Since bulk state updates are involved, the handler can utilize DMA to accelerate it. In *ammp*, the time spent in servicing read requests is large, due to low hit rates of the BRP bits (Figure 8).

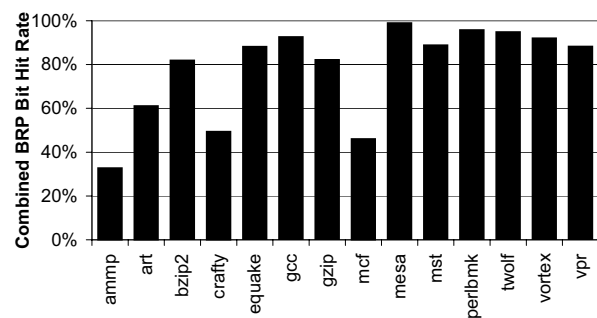


Figure 8. Percentage of BRP bit accesses that hit in L1 or L2.

Figure 8 shows the percentage of BRP bit accesses that are L1 or L2 cache hits. Since most of the time the BRP bits are not zero, these hits do not usually result in bug checks. Therefore, in general, the BRP bits are very effective in filtering/reducing the number of bug check requests. In all but four applications, more than 80% of heap reads or writes do not result in bug check requests to the HeapMon helper thread. This explains why the helper thread is only busy less than 20% of the time

for those applications (Figure 7). Three out of four applications that have low BRP hit rates (*ammp*, *art*, and *mcf*) also have high execution overheads (Figure 6). This is due to the shared L2 cache contention where the application's data competes with the helper thread's data for cache space, producing undesirable effects where the helper thread's state bit data replaces some of the application's data. This increases the application's data and BRP-bit miss rates, which in turn increases the helper thread's occupancy. On the other hand, although *crafty* suffers from a low BRP bit hit rate, HeapMon's execution time remains extremely small. This is due to the fact that fewer than 0.5% of *crafty*'s memory accesses are to the heap region as seen in Table 1.

6 Conclusions

This paper presents *HeapMon*, a user-level helper thread that runs on a separate CMP processor and effectively detects memory bugs, such as reads from uninitialized or unallocated memory locations, and memory leaks. This new approach does not require human intervention and has only minor storage and execution time overheads. The storage overhead is 3.1% of the cache size and 6.2% of the allocated heap memory size. Although architectural support for HeapMon is simple, its execution time overhead is only 8% on average, and less than 26% in the worst case. Such a low overhead is due to an efficient bug check filtering mechanism, using a small L2 cache to reduce contention between the application and the helper, and using DMA for bulk state updates.

References

- [1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 290–301, 1994.
- [2] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conf.*, pages 63–78, San Antonio, Texas, 1998.
- [3] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Intl. Conf. on Software Engineering*, 2002.
- [4] KAI-Intel Corporation. Intel thread checker. URL: <http://developer.intel.com/software/products/threading/tcwin>. 2004.
- [5] V. Krishnan and J. Torrellas. A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 286–293, 1998.
- [6] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2029, 2001.
- [7] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symp. on Principles of Programming Languages*, pages 128–139, 2002.
- [8] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs, 1997.
- [9] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.
- [10] Rational Software. IBM Rational PurifyPlus for Unix. ROI Analysis. URL: <http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/tp613.pdf>. 2002.
- [11] Rational Software. http://www.rational.com/products/purify_unix/index.jsp. 2004.

- [12] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly, 2nd ed, 2001.
- [13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. 1997.
- [14] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. URL:<http://valgrind.kde.org/>. 2004.
- [15] Standard Performance Evaluation Corporation. Spec benchmarks. <http://www.spec.org>, 2000.
- [16] G. Tasse. The Economic Impacts of Inadequate Infrastructure for Software Testing. *NIST Report 02-3*, 2002.
- [17] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *2nd Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, 1994.
- [18] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torellas. iWatcher: Efficient architectural support for software debugging. In *31st Intl. Symp. on Computer Architecture (ISCA)*, 2004.