HeapMon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection

R. Shetty
M. Kharbutli
Y. Solihin
M. Prvulovic

The ability to detect and pinpoint memory-related bugs in production runs is important because in-house testing may miss bugs. This paper presents HeapMon, a heap memory bug-detection scheme that has a very low performance overhead, is automatic, and is easy to deploy. HeapMon relies on two new techniques. First, it decouples application execution from bug monitoring, which executes as a helper thread on a separate core in a chip multiprocessor system. Second, it associates a filter bit with each cached word to safely and significantly reduce bug checking frequency—by 95% on average. We test the effectiveness of these techniques using existing and injected memory bugs in SPEC®2000 applications and show that HeapMon effectively detects and identifies most forms of heap memory bugs. Our results also indicate that the HeapMon performance overhead is only 5%, on average—orders of magnitude less than existing tools. Its overhead is also modest: 3.1% of the cache size and a 32-KB victim cache for on-chip filter bits and 6.2% of the allocated heap memory size for state bits, which are maintained by the helper thread as a software data structure.

Introduction

Many software tools have been developed to find errors through the static analysis of code or to monitor an aspect of program behavior at runtime and detect problems. Static methods can be used without affecting performance, but imperfect memory disambiguation and input-dependent program behavior severely limit their scope. Runtime tools can significantly improve programmer productivity and reduce development costs [1], but they suffer from large performance overheads that preclude their use in production runs of deployed applications. As a result, problems that remain in deployed code can create errors, crashes, and security vulnerabilities.

This is especially true for memory bugs such as reads from uninitialized memory, reads or writes using dangling pointers, and memory leaks, which are common problems in C and C++ programs. Memory bugs are difficult to detect by code inspection because they may involve different code fragments and exist in different modules or source code files. The compiler is of little help because it typically fails to accurately disambiguate pointers [2]. Thus, in practice, memory bug detection relies on runtime checkers [3–13] that insert monitoring into the application using a compiler or binary instrumentation, but the resulting performance overhead is tolerable only in debugging runs. In deployed software, instrumentation is removed and bugs that occur in production runs are not detected.

Detection of memory-related bugs in production runs can help pinpoint surviving bugs. Left undetected, such bugs may cause behavior that is difficult to discern, such as occasional wrong computation outputs, late, obscure, or intermittent system crashes, security attacks, and subtle performance loss. For example, reading from an uninitialized variable may result in crash-free execution with wrong computation outputs, while undetected memory leaks can induce excessive page faults or much-

0018-8646/06/$5.00 © 2006 IBM

delayed crashes when the available memory is eventually exhausted. Security vulnerabilities due to memory bugs may become apparent only when an attack is successful. Popular programs such as Microsoft Internet Explorer**, Microsoft Internet Information Services, and Macromedia Flash** and ColdFusion**, among others, are known to be vulnerable to heap-related attacks [14–16]. The Code Red worm [17] is a recent example.

In this paper, we focus on architecture support for efficient detection of memory bugs in production runs. We carefully designed the hardware support so that it is not specific to the type of bugs to be detected or the type of checkers used for detection. We have avoided bug- or checker-specific hardware extensions because different users, applications, programming languages, and environments may require different checks.

The bug checks in our design are performed in software by a *helper thread*. Our memory bug checker, *HeapMon*, monitors application heap space to detect heap memory bugs. This approach takes advantage of chip multiprocessing (CMP) and multithreading and allows checks to overlap with regular program execution in the application thread. In HeapMon, the status of each word on the heap is monitored by associating with it a state that indicates whether the word is *unallocated* (U), *allocated but uninitialized* (AU), or *allocated and initialized* (AI). Each state defines the accesses to that word that are legal and those that are illegal (bugs). When a bug is detected, its type, program counter, and data address are logged so that developers can determine the nature of the bug and its precise location.

To facilitate this approach, we use three hardware mechanisms that focus on efficient and effective communication between threads. The first is a communication queue for forwarding events from an application thread to its helper thread, together with instruction set architecture (ISA) extensions to insert and extract events from this queue. This avoids long-latency communication of events through shared memory. The second mechanism automatically forwards memory access events to the helper thread. This avoids the need to instrument load/store instructions, which must be checked in most monitoring and bug-detection schemes. The third mechanism filters out redundant or unnecessary memory access events using a set of bits in the application processor cache. Because the definition of *redundant* or *unnecessary* depends on the kind of checking being done by the helper thread, the helper-thread code has full control over these filter bits.

With this design, HeapMon offers the following benefits:

- *Low-overhead:* The helper-thread approach completely decouples bug monitoring from application execution, and the filtering mechanism reduces bug-check frequency. Consequently, HeapMon achieves a very low performance overhead (see the section on evaluation results).
- *Automatic:* HeapMon automatically monitors the entire heap region of the application; there is no need to insert watchpoints or specify memory regions for monitoring in the application code.
- *Deployable:* HeapMon monitors existing program object files. It is deployed by relinking the application with a new static memory allocation library, or simply running it with a new dynamically linked memory allocation library.[1]

Architectural support for HeapMon is quite inexpensive. The communication queues between the application and the HeapMon helper thread are small, and the filter bits require modest storage overhead: 3.1% of the cache size and a 32-KB victim cache for the on-chip filter bits. Software storage overhead of HeapMon includes 6.2% of the allocated heap memory size for storing the per-word state information, which is incurred only for applications that are monitored by HeapMon. Although HeapMon uses an extra CMP processor to run the helper thread, there is still a significant savings of total processor utilization (i.e., *processor count* $\times$ *time*). The HeapMon thread is running only 15% of the overall application execution time, indicating the possibility of time-sharing the processor with other tasks. Its low overhead allows more prerelease software testing and the use of more realistic test inputs, improving the productivity of testing and debugging. It may also affect whether monitoring and checking will be always used, even in production runs of deployed software.

Related work

Our work focuses on low-overhead architectural support for runtime bug detection using a helper-thread approach. Detection of memory bugs can also be performed statically by explicit model checking [18, 19] and program analysis [20–22]. Static methods do not affect performance, but their scope is severely limited by input-dependent program behavior and difficulties in disambiguating memory references. Detection of memory bugs can also be performed at runtime by instrumenting the code with checks, as in IBM Purify* [11], Valgrind [13], Intel Thread Checker [6], DIDUCE [5], Eraser [12], CCured [8], and others [3, 4, 7, 9, 10]. Such instrumentation typically introduces large performance overheads because instrumented memory references (loads and stores) execute often, and execution of the instrumentation code is

[1]Some applications manage their own heap through custom allocation and deallocation routines. However, we found that custom routines are typically localized in very few functions. They can be modified to work with HeapMon without a significant programming effort.

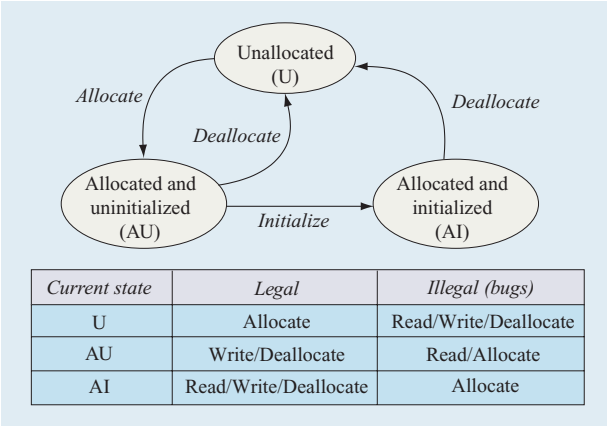
interleaved with normal program execution. Extending data structure definition (such as pointers) can greatly improve the ability of runtime checkers to detect bugs [3]. Such extensions are orthogonal and largely complementary to our HeapMon approach.

Recently, hardware support concepts for detecting memory bugs or to support debugging—such as iWatcher [2], AccMon [23], SafeMem [24], DISE [25], and that by Oplinger and Lam [26]—have been proposed. iWatcher, AccMon, and DISE essentially extend hardware watchpoints and breakpoints to an arbitrary number of locations. In iWatcher and AccMon, a load or a store address is checked against watched addresses maintained in hardware tables. A match triggers an exception and transfers control to bug-checking code, which returns control to the application when the check is complete. These exceptions disrupt application execution and occur even if the check is redundant or unnecessary. To avoid these overheads, AccMon introduces a bloom filter that avoids the exception for certain loads or stores. However, the filter is specific to the particular checking technique proposed. In DISE, store instructions are instrumented by dynamically replacing each with a set of instructions at fetch time. These instructions check the store address against watched addresses and call a debugging function when there is a match. This address matching is performed in software and is interleaved with the application execution; it is unsuitable for watching a large number of addresses. All three schemes (iWatcher, AccMon, and DISE) interleave application execution with bug-checking, address-matching, or debugging functions. In contrast, our helper-thread mechanisms are unique in that they execute checks in parallel with application execution and eliminate most unnecessary checks on loads and stores without using a bug- or checker-specific hardware extension. Note that our helper-thread approach is not specific to HeapMon, and may help hide bug-checking latencies in other schemes.

In past studies, helper threads have been used mainly for prefetching [27–32] and branch prediction [31]; HeapMon uses them for bug detection. Our checkers are also related to hardware tags that store and manage the state of main memory locations [33–37] and, more recently, Mondrian memory protection [38]. In contrast to hardware tags, our helper thread maintains tags in software data structures without any special hardware support for storing, managing, checking, and updating them. This allows HeapMon to be adapted to different kinds of checks with different tags, or even without tags, by implementing different code in the helper thread.

Bug detection: Coverage and limitations

This section discusses the types of bugs that can be detected by HeapMon and the limitations of our current implementation.



Current state	Legal	Illegal (bugs)
U	Allocate	Read/Write/Deallocate
AU	Write/Deallocate	Read/Allocate
AI	Read/Write/Deallocate	Allocate

Figure 1

State transition for each word. The table shows the legal requests and the illegal requests (detectable bugs) for each state.

Bug-detection coverage

To detect bugs, HeapMon helper thread allocates and maintains two bits of state for each word in the heap area (**Figure 1**). All free words in the heap region have a U state. When an object is allocated (via `malloc` or an equivalent function), the state of all words of the object changes to AU. When a word in the object is written or initialized, the state of the word changes to AI. Finally, when an object is deallocated (via free or equivalent functions), the state of its words changes back to U. The states and the state transition diagram are adopted from Purify [11]. Consequently, HeapMon inherits some of the Purify bug-detection capability and some of its limitations. The main advantage of HeapMon over Purify lies in the decoupling of the application code and the bug monitoring, and the use of architecture support to efficiently reduce bug-monitoring frequency, which results in orders of magnitude lower execution time overhead.

As in Purify, bug-checking conditions shown in Figure 1 can detect six types of memory-related bugs at runtime. Memory leaks are detected when, at the end of program execution, some words in the heap region are still in one of the allocated state.[2] Note, however, that HeapMon does not distinguish between true memory leaks with heap objects that are intentionally left allocated by the programmer.

Like Purify, HeapMon can detect heap buffer overflows. To accomplish this, the memory allocation and deallocation routine is modified to leave a small unallocated block between each pair of consecutive allocated regions. This would detect buffer overflow

[2]Note that Purify employs a more accurate and timely memory-leak detection that relies on mark and sweep-conservative garbage collection.

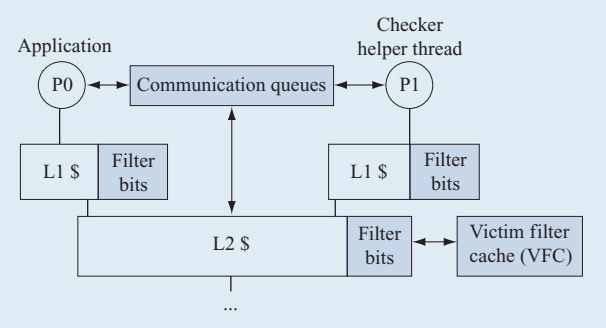


Figure 2

HeapMon hardware support.

attacks, such as the Code Red worm [17], because the attack attempts to write to such unallocated blocks. It would also detect some array-out-of-bounds errors.

Bug-detection limitations

Because HeapMon monitors the heap memory, where most pointer-related bugs occur, it cannot detect bugs in the global data and stack segments. Since we track states at the granularity of 32-bit words, bugs on accesses to byte-sized locations may not always be detected.

Bug-detection mechanisms

Basic implementation

Like Purify, HeapMon can be implemented in software by intercepting memory allocation and deallocation library calls and load/store instructions, and adding code for checks and state updates. Unfortunately, each check and state update may result in executing many instructions, some of which are memory references and difficult-to-predict branches. Because most applications frequently execute load and store instructions, the checks and state updates they trigger may introduce significant performance overheads. Although static analyses in some cases can determine when checks or state updates are unnecessary, they are limited by imperfect memory disambiguation and unavailability of the source code.

HeapMon operates on application binaries and does not use any compile-time information. As a result, all static load and store instructions must trigger appropriate checks and state updates. However, instead of interleaving checks and state updates with regular application execution, HeapMon performs them in a separate thread that runs in parallel with the application thread. This approach is possible without any hardware support but is impractical because of its performance impact.

Optimizing helper-thread communication

To allow fast communication and synchronization between the two threads, HeapMon hardware support includes a set of hardware communication queues (**Figure 2**) and new instructions that insert event notifications into and extract them from the queues. There are two types of first-in first-out (FIFO) queues: a *request queue* stores the bug-check requests generated by the application processor; a *reply queue* stores the bug-check results computed by the helper thread. Queue-insertion instructions stall when the queue is full, which provides synchronization when the helper thread cannot keep up but allows the application thread to work at full speed when the helper thread is able to keep up.

We note that any helper-thread checker that monitors frequent events in the application thread would have similar communication and synchronization problems. Our communication and synchronization support is not specific to HeapMon and can be used by any other checker implemented as a helper thread.

Reducing instrumentation overhead

Even with special instructions for inserting events into the request queue, instrumentation of all load and store instructions in the application would have a heavy impact on both performance and code size. To eliminate this instrumentation, HeapMon includes support for automatically placing load and store events in the request queue. Such support is needed only for frequent events, while infrequent events are still inserted into the queue in software. This approach simplifies our hardware support because it forwards only a few classes of frequent events, such as loads and stores. In HeapMon, memory allocation and deallocation events are still forwarded using software instrumentation and our ISA extensions, but such instrumentation is now required only in the heap memory-management library. As a result, to use HeapMon with an application, we need to use only a modified dynamic library or relink it with a modified static library.

We note that this support for automatic forwarding of load and store events is, again, not specific to HeapMon. Other checkers that must observe memory accesses can enable the forwarding of loads, stores, or both to avoid heavy instrumentation of the application code and the resulting performance overheads.

Reducing helper-thread workload

One important factor in maintaining good application performance is to keep the helper thread less busy than the main thread. Otherwise, the helper thread cannot keep up, the request queue becomes full, and the application thread stalls. Helper-thread workload can be reduced by optimizing its code, but even then, the

processing time per load or store event is longer than the typical time the application thread spends between consecutive load or store instructions. Fortunately, many events forwarded to the helper thread are redundant or unnecessary. For example, HeapMon detects heap bugs, so only heap-related events have to be forwarded to its helper thread. There are other checker-specific requests that can be omitted: An example is a store to an already allocated and initialized heap memory location that always results in a successful HeapMon check and does not change the HeapMon state for that location. However, forwarding of such events to the helper thread still results in a state lookup to determine that no action is needed.

We have introduced two filtering mechanisms to prevent some forwarding of unnecessary events to the helper thread. Because we wanted to avoid checker-specific hardware support, both filtering mechanisms are not checker-specific and can be programmed by the checker helper thread. First, we use an *address filter* that specifies the start and end of the virtual address range in the application for which events should be forwarded to the helper thread. For example, the helper thread may program the address filter to forward only events related to the heap, the stack, another range, or the entire address space.

The second mechanism provides fine-grain filtering that consists of a filter bit for every word in the application processor cache. If a filter bit is set to 1, load or store events for that memory location are not forwarded to the helper thread. If the filter bit is set to 0 (cleared), events for that memory location are forwarded to the helper thread. Filter bits are examined by the forwarding hardware when a decision is being made about whether to place an event in the request queue.

The decision about when to set or clear a particular filter bit depends on the kind of checking performed by the helper thread. Different checkers have different definitions of *redundant* and *unnecessary* events. Consequently, our filter bits are under the control of the helper-thread software. For this purpose, the helper thread puts filter bit set and clear operations into the reply queue. These operations are extracted from the queue by the application processor hardware, which sets and clears the filter bits accordingly.

If every word were associated with a filter bit, these bits would be too numerous for all of them to be kept on chip. To simplify our HeapMon hardware, filter bits are kept only for cached data and are discarded when data is displaced from the cache. Fetching a block from memory into the cache triggers a request to the helper thread for generating the filter bits for the entire cache block. The performance impact of keeping filter bits only on chip can be further reduced by introducing another level of on-chip caches for filter bits only, which we call the victim filter cache (VFC) (Figure 2). Because filter bits are much smaller than the corresponding data, the VFC can be small and fast, yet still provide filter bits for most data accesses that miss in on-chip caches and consequently, eliminate most of the extra event forwarding.

Finally, our event-filtering mechanism is not specific to HeapMon—another checker can control these filter bits to safely filter out memory read/write events that are redundant or unnecessary for that particular checker.

Reporting modes

We envision two major scenarios for using HeapMon-based checkers. Our primary focus is the always-on scenario, in which checking is done in production runs of software that is already deployed. In this scenario, the application is not debugged interactively; problems are logged and sent to the software vendor for analysis. The second scenario is in prerelease testing and debugging, where it is best if each detected problem triggers an exception at the instruction for which the problem is detected in order to facilitate interactive debugging. In the debugging mode, after completing the check for an event, the helper thread uses the ISA extensions to insert a response into the reply queue. The application processor hardware consumes these responses automatically. Responses indicating that no error is detected are used to allow retirement of the event-causing instruction. Responses that indicate the detection of errors trigger an exception in the event-causing instruction. To maintain precise exceptions, the application processor delays retirement of instructions for which responses from the helper thread are still pending.

There are two requirements for the correctness of bug detection in HeapMon.[3] First, the order in which requests are inserted into the request queue must correspond to the program order of the instructions that generate the requests. Without following the program order, false bugs may be detected whereas true bugs may not be detected. The second requirement is that speculative instructions that will eventually be squashed should not update any state maintained in the helper thread.

Always-on mode

In the always-on mode, both requirements are met by forwarding events to the helper thread as each instruction retires, which occurs in program order and only for nonspeculative instructions. For a load or store instruction, we check whether the filter bit for the accessed word is available (i.e., has been looked up from the caches). If it is available and its value is 1, the

[3]There are also requirements for the order in which a filter bit is looked up or cleared. For clarity and space, we omit a discussion on how this is handled, but it is taken into account in our experiments.

instruction is immediately retired. If it is available and its value is 0, an event corresponding to the instruction is inserted into the request queue and the instruction is retired. If the request queue is full, the retirement of the instruction stalls until an entry in the request queue becomes available. If the filter bit is not available because it is not found in the cache, an event is also inserted into the request queue, conservatively assuming that checking is needed for this instruction. Because a load instruction must access its data before it can be executed, by the time it is ready to retire, the filter bit will have been looked up from the cache. However, a store instruction is typically retired before it accesses the caches. To avoid generating an event because the filter bit has not been looked up, we issue a filter bit access for a store instruction as soon as its address becomes available. In most cases, by the time the store is ready to retire, its filter bit has been looked up. If it has not, the store conservatively assumes that the filter bit is 0 and places an event in the request queue. Thus, in the always-on mode, instruction retirement stalls only if the request queue is full.

Debugging mode

In the debugging mode, the retirement of a check-causing instruction is delayed until the checker response is received. If, as in our always-on mode, a check is requested just before an instruction commits, retirement in the application thread stalls until the check is complete. With this in mind, checks in the debugging mode are forwarded to the helper thread when the request-triggering instruction is executed. Because instruction execution can be speculative and out of order, the actual exception processing is delayed until retirement. To achieve this, each load, store, or queue-insertion instruction in the processor is tagged with a *no-bug-found* bit, which, if set to 1, indicates that the instruction check has been completed without any bug being detected, and therefore the instruction can retire without raising an exception.

We note that the program-order requirement in the request queue is too strong and can be relaxed to dependence-based ordering, in which the order in which requests are inserted into the request queue must correspond to the program order of the corresponding instructions only if they access the same or overlapping addresses. However, instructions that access non-overlapping addresses can be inserted into the request queue out of order. To enforce this ordering, a load, store, or queue-insertion instruction waits until all preceding load, store, or queue-insertion instructions have their addresses disambiguated. After that, the request is inserted into the request queue if there is no address overlap or if the only address overlap is with instructions whose requests are filtered out or already inserted in the request queue.

The second requirement is to prevent speculative instructions from causing permanent state updates in the helper thread. To satisfy this requirement, each request in the request queue is tagged with a *speculative* bit that is set for an instruction that generates a request while it is still speculative. The bit is 0 if a request is generated by an instruction that is about to retire. Upon receiving a request, the helper thread checks the bit and may perform a state update only when the request is not speculative. For speculative requests, the checker determines whether an update request will be needed when the instruction retires. Details of this mechanism are omitted because of space limitations.

Overall recommendation

Delayed instruction retirement and separate check and update requests in the debugging mode make it more complex and can adversely affect its performance. The always-on mode is likely to be used in most systems, while the debugging mode will be used primarily in software development. Consequently, the principle of *optimizing the common case first* leads us to believe that the always-on mode with simple error logging should be implemented first. The debugging mode with precise exceptions can be added if cost and complexity considerations permit.

HeapMon operation

To illustrate how HeapMon interacts with our hardware support, we present a timeline of a check-triggering instruction from execution to retirement in the application thread.

HeapMon helper-thread implementation

Figure 3(a) shows a simplified view of the helper-thread main loop. The thread operates in a tight loop that retrieves and processes bug-check requests, using lightweight sleep to wait when the request queue is empty. Allocation requests in the figure correspond to `malloc()`, `calloc()`, and similar library calls.

Figure 3(b) shows HeapMon helper-thread code for processing a check request for a read event. The code in lines 02–04 locates the byte that contains state bits in the HeapMon thread data structure, and the code on line 06 extracts these bits. Lines 07 and 08 show the code for detecting and reporting reads to unallocated or uninitialized words. A bug report contains the type of event (a load, store, allocation, or deallocation), the current state and virtual address of the heap word in question, and the program counter that corresponds to the event. Some statements for statistics collection and filter-bit manipulation are implemented but are not shown in Figure 3(b).

```
01 while(!exitApp){
02   getNextEvent(&event);
03   switch (event.eventType) {
04     case HM_MALLOC, HM_CALLOC_, HM_REALLOC : alloc_handler( addr_info, event); break;
05     case HM_FREE : free_handler( addr_info, event); break;
06     case HM_READ : read_handler( addr_info, event); break;
07     case HM_WRITE : write_handler( addr_info, event); break;
08     case HM_GET_FILTER_BITS: get_filter_bit_handler( addr_info, event); break;
09     case HM_PAUSE_CHECKING: pause_handler(); break;
10     case HM_RESUME_CHECKING: resume_handler(); break;
11     case HM_NOEVENT: lightsleep(); break; // Request Queue empty
12     case HM_EXIT : check_for_leaks(); exitApp=true; break;
13   }
14 }
```

(a)

```
01 void read_handler(AddrSpace_t addr_info, Event_t event) {
02   int byteOffset = (event.dAddr - addr_info.heapStart) >> LOG2_NUMBYTES_PER_TAGBYTE;
03   int bitOffset = ((((event.dAddr - addr_info.heapStart) % NUMBYTES_PER_TAGBYTE)
04                     >> LOG2_BLOCK_SIZE)) << LOG2_NUM_TAG_BITS;
05
06   unsigned char heapState = (heapStateBits[byteOffset] >> bitOffset) & 0x03;
07   if (heapState != INIT_ALLOC)
08     reportBug(event.eventType, heapState, event.dAddr, event.iAddr);
09 }
```

(b)

Figure 3

(a) HeapMon helper-thread main loop code. (b) HeapMon helper-thread code for processing a bug-check request from a read.

In implementing the helper thread, we started with a clean, high-level implementation. However, we found the response time of the helper thread to be unsatisfactory. We applied more aggressive compiler and manual optimizations on the code, including using immediates (defines) for some values, constant propagation, strength reduction, function inlining, loop unrolling, etc. To efficiently handle allocation and deallocation requests to a large region (128 bytes or more), the helper thread updates the state bits using 8-byte writes. Overall, the optimizations reduce the helper-thread response time by 80–95% compared with the unoptimized version.

Request timeline in HeapMon

Figure 4 shows the sequence of events for a bug check in HeapMon running in the always-on mode. Each allocation or deallocation function is augmented with an instruction that inserts an event into the request queue. When the application calls such a function, a request event is placed in the request queue (circle la in Figure 4). The event also requests that the helper-thread bug checking pause while it is executing the function, so that loads and stores needed for heap management are not detected as bugs. While paused, the helper thread ignores bug-check requests until it is resumed. For a read or write instruction, the filter bit for the accessed word in on-chip caches is checked. If the filter bit is found to be set to 1 for an access, the access is safe (i.e., it will not result in bug detection), and no bug-check request is generated. If the filter bit is found (or assumed) to be 0, a bug check is generated for that access (circle 1b or 1c) by placing a request in the request queue (circle 2), if the access is to an address in the address range specified in the address filter. When the filter bit is not found in the caches, the request also requests that the filter bits of the entire cache block be generated by the helper thread. A request in the request queue contains the application process identification, program counter of the requesting instruction, request type, starting virtual address, request size, and speculative bit. Virtual addresses are preferred in a request because regions of memory (heap, stack, code, etc.) are contiguous in the virtual address space, which allows easier tracking of state for each location in the helper-thread code and avoids issues that would arise because of paging and disk swapping.

After processing a request, the helper thread checks the queue and retrieves the next request (circle 3). Alternatively, it may have been in a lightweight sleep because there were no more requests to process, in which case it receives a *wake up* signal when a new request is

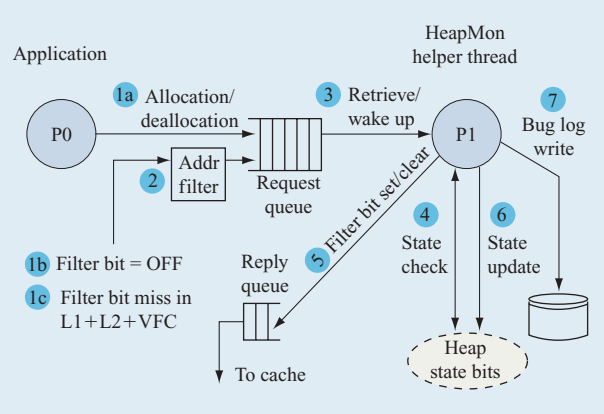


Figure 4

HeapMon bug-check protocol in the always-on mode. For brevity, we show only one pair of request and reply queues and do not show the memory hierarchy. Heap state bits are a software data structure maintained by the helper thread.

placed in the queue. In both cases, the thread obtains the request and reads the *heap state bits* (circle 4) to determine whether the request corresponds to a valid heap access or a bug. On the basis of the request type and the current heap state for the accessed word, the thread may reply with a *filter-bit-set* or a *filter-bit-clear* signal (circle 5) to turn the filter bit on or off. With the checker, the signal is sent in two cases. In the first case, the request is from a read or a write that finds the word in the AI state or from a write that puts the word in that state. In this case, the thread sends a filter-bit-set signal to turn the filter bit on so that future reads/writes to the word will not generate bug-check requests. In the second case, the request is a deallocation; here, a filter-bit-clear signal is sent to enable bug-check requests on future accesses to those words.

If a request results in a state transition, the thread performs a state update on the heap state bits (circle 6). Because the state bits were recently read by the thread, they probably still reside in the cache, and the update is a cache hit. Finally, if a bug is found, it is logged, together with all relevant information that can be reported to the programmer or user (circle 7). In the always-on mode, bug detection, if desired, can also raise an (imprecise) exception in the application, for example to terminate the application and prevent corruption of sensitive data.

Other implementation issues

- *Communication queue implementation:* In both always-on and debugging modes, a request-generating instruction stalls if the request queue is full. Reducing this stall time is an important issue. In practice, we found that with a sufficiently large request queue (64 entries), the application processor rarely stalls because of transient increases in the request rate. The processor can still stall when the helper thread is busy for a long time, such as when it is processing a large allocation in HeapMon, while the application thread keeps generating and queuing up requests that eventually fill the request queue. In contrast, the reply queue can be very small (we use eight entries in our experiments) because replies are quickly consumed by the application processor hardware.
- *Helper-thread scheduling:* To be effective, the helper thread should be gang-scheduled with the application so that event-check requests will be serviced in a reasonable amount of time. In our evaluation, we assume that the helper-thread processor is not multitasked.

Evaluation setup

Applications

We use two sets of applications in our evaluation of HeapMon. Set A is used to evaluate the performance overheads and characteristics. We use 14 applications, mostly from SPEC**2000 [39]. The applications, their sources, input sets, L1 and L2 cache miss rates, number of allocations, average allocation size, and percentage of memory accesses that go to the heap region are shown in **Table 1**. We omitted Fortran benchmarks because they use no heap memory and can neither produce HeapMon checks nor benefit from them. Among C/C++ benchmarks, `vortex`, `gap`, and `parser` are not included because they currently do not run on our simulator. To correctly detect bugs, the program must be monitored from the beginning to track, for example, exactly which heap locations have been initialized. Consequently, all benchmarks are simulated from the beginning to the end using primarily test input sets to keep the simulation times reasonable.

Set B is used to evaluate HeapMon bug-detection capability. We use applications and input sets supplied by the authors of AccMon [23]. We were able to compile and run three of these applications (ncompress-4.2.4, polymorph-0.4.0, and bc-1.06). Two other applications (gzip-1.2.41 and man-1.5h1) could not be compiled because of limitations in our cross-compiler infrastructure. Of the three compiled applications, only bc-1.06 was reported to have heap-related bugs; thus, we expected to detect bugs only in bc-1.06. We did not use set B applications to evaluate performance because their input sets were designed to make bugs manifest, but are very small and unrealistic for performance evaluation.

In set A applications, HeapMon identifies only memory leaks. This is not surprising: They have gone through rigorous debugging and testing for the standard

Table 1 The applications used in our performance evaluation (set A).

Benchmark	*Source*	*Input set*	*L1 cache miss rate* (%)	*L2 cache miss rate* (%)	*Number of allocations*	*Average allocation size* (bytes)	*Heap accesses* (%)
ammp	SPECfp**2000	test	19.44	33.7	34,766	399	32.8
art	SPECfp2000	test	9.65	48.03	30,490	75	27.45
bzip2	SPECint**2000	test	0.84	1.66	13	1,051,360	3.86
crafty	SPECint2000	test	2.15	0.06	44	20,140	0.38
eon	SPECint2000	test	0.14	0.13	2,595	85	5.03
equake	SPECfp2000	test	1.24	57.59	316,854	36	32.37
gcc	SPECint2000	test	1.29	0.89	4,305	3,529	17.37
gzip	SPECint2000	test	2.82	3.64	246	27,866	4.11
mcf	SPECint2000	test	7.16	13.35	10	9,660,180	22.42
mesa	SPECfp2000	test	0.13	61.71	69	302,464	14.03
mst	Olden	1,024 nodes	1.7	37.21	422	32,420	40.12
perlbmk	SPECint2000	test	0.36	14.46	423	19,948	34.34
twolf	SPECint2000	test	1.09	0.08	9,413	73	26.28
vpr	SPECint2000	test	2.17	0.01	1,594	127	25.21

input sets over the years. To further evaluate our bug-detection capability, we injected bugs into these applications.

Simulation environment

The evaluation is performed on an extension of SESC [40], a detailed cycle-accurate execution-driven CMP simulator. **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. We assume that the helper thread can obtain a request from its request queue with a single-cycle latency. We note, however, that this has little effect on the overall request-processing latency, which is of the order of hundreds of cycles.

Evaluation results

In this section we present execution time overheads of HeapMon running in two modes. We discuss in detail its performance characteristics and the sensitivity of the overhead to different cache parameters, and we evaluate its bug-detection effectiveness.

Performance overhead and characteristics

Figure 5 shows execution time overhead of HeapMon running in the always-on mode and in the debug mode with precise exceptions for bug detection. We use a 64-entry request queue, an eight-entry reply queue, a filter bit for each cached word, and a 32-KB VFC that caches filter bits for displaced L2 blocks. Each bar represents the additional execution time with HeapMon, relative to baseline execution time without HeapMon.

The execution time overheads are quite low, with averages of 4.9% for the always-on mode and 10.8% for the debug mode. In the always-on mode, HeapMon overhead is above 10% only for equake (28.6%). In the debug mode, the overhead is less than 10% in nine of 14 benchmarks. The largest overhead is that of perlbmk (49%). The low overhead in the always-on mode supports the argument that, unlike existing bug-detection tools, HeapMon can be used in production runs.

To better understand HeapMon overhead, we characterize its always-on performance. **Figure 6(a)** shows the helper-thread's execution time, broken down into the time it is busy servicing allocation requests, deallocation requests, read requests, write requests, and computing filter bits. Each bar is normalized to the application execution time. Therefore, the height of the entire bar for each application shows the fraction of the execution time during which the helper thread is busy.[4]

The figure shows that, on average, HeapMon processor services requests only 15% of the time and is busy more than 30% of the time in only two applications (ammp and equake). This shows that although the HeapMon helper thread runs on a separate CMP processor, it does not

[4]This "busy" time still includes various pipeline stalls in the HeapMon thread processor.

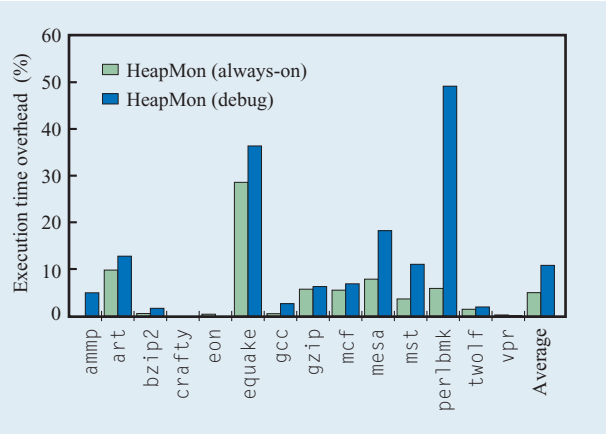


Figure 5

Execution time overhead of HeapMon in two configurations.

fully occupy the processor and could be interleaved with other activity to maximize system throughput.

In most benchmarks, the HeapMon helper thread spends most of its time computing filter bits. There are two reasons for this. First, HeapMon computes filter bits for an entire cache line (16 words) at a time, which requires more time than to service reads or writes to individual memory locations. Second, such filter-bit computation helps eliminate later read and write check requests, which reduces the time spent servicing such requests. In `mcf`, however, most of the time is spent on servicing allocation and deallocation requests, owing to very large allocations and deallocations (averaging 9.7 MB).

Figure 6(b) shows the percentage of memory accesses that find the filter bits in the L1 cache, L2 cache, or the VFC. We also show whether the filter bit is set or clear. Several observations can be made from this figure. First, the HeapMon filtering mechanism is very effective in reducing the frequency of bug checks. On average, 95% of all accesses find their filter bit set in the cache (Hit + on). Consequently, only 5% of the accesses generate bug-check requests, which is key to the low performance overheads. Lower filter-bit hit rates are observed in only `ammp` and `art`, which are also the only applications with very high L2 miss rates (34% and 48%, respectively), L1 miss rates (19% and 10%, respectively), and heap accesses (33% and 27%, respectively), indicating a large and active working set in the heap. However, even for these applications, the execution time overhead is low (0.4% and 9.8%, respectively). Finally, we observe that, on average, only 1.5% of all accesses find the filter bits in one of the caches, but the bits are clear (Hit + off). This confirms that a heap object is accessed many times after it is allocated and initialized, and that filter-bit caching almost always results in avoiding bug-check requests.

Table 2 Parameters of the simulated architecture. Latencies correspond to contention-free conditions. RT = round trip from the processor.

Processor core	4-GHz, six-way out-of-order issue
	Integer, floating-point, and load/store functional units: 4, 4, 3
	Branch penalty: 17 cycles
	Reorder buffer size: 248
Memory hierarchy	L1 instruction (per processor): write-back, 16-KB, two-way, 64-B line, RT: three cycles, least recently used (LRU) replacement, Miss Status Handling Register (MSHR) size: 24.
	L1 data (per processor): WB, 16 KB, two-way, 64-B line, RT: three cycles, LRU repl., MSHR size: 24.
	L2 unified (shared): 1-MB, eight-way, 64-B line, RT: 13 cycles, LRU repl., MSHR size: 48.
	Memory bus: split-transaction, 8 B wide, 1 GHz, 8-GB/s peak
	RT memory latency: 400 cycles
Additional hardware	Request/reply queues: 64/8 entries, FIFO, one-cycle access time
	Filter-bit storage in L1 and L2: 3.1% of cache sizes (512 B at the L1 + 32 KB at the L2)
	Victim filter cache: 32 KB, eight-way, 2-B line, four-cycle access time

Table 3 shows the helper-thread's average service time (in cycles) for different types of activity. The table shows that a single read or write request is usually processed in less than 100 cycles, and even cache-block filter bits are typically computed in less than 400 cycles. However, allocations and deallocations may involve a large memory region, and their service time varies greatly. Ten large allocations in `mcf` average 9.7 MB each (Table 1) and have the longest average service time of 1.85 million cycles per allocation and 2.6 million cycles per deallocation. Note that allocation and deallocation requests result in changing the state of each word in the affected region, so HeapMon processing time for these requests grows in proportion with request size.

Figure 6(c) shows the time during which the request queue is full as a percentage of the application execution time. The figure shows that on average the queue is full only 1.3% of the time. The worst case is `mcf`, for which

the request queue is full 4.6% of the time. For most of this time, the helper thread is busy processing a large allocation or deallocation request, and the application thread fills the queue with subsequent load and store requests. We find that even substantially larger queues (thousands of entries) are insufficient to prevent application stall in such situations. Overall, however, these results indicate that the queue size of 64 entries is sufficient for most programs, and that most of the HeapMon execution time overhead can be attributed to other factors.

Sensitivity study

We tested the sensitivity of HeapMon performance to changes in the L2 cache and VFC parameters. **Figure 7(a)** shows the performance overhead when no VFC is used, a 1-MB private L2 cache for the helper thread in lieu of a VFC, and configurations with a 16-KB, a 32-KB, and a 64-KB VFC. Without a VFC, the HeapMon thread introduces an average overhead of 15.2%. We suspected that this performance overhead might be caused by cache contention between the application and the helper thread, so we tried adding a 1-MB private L2 cache for HeapMon. This is not a realistic configuration, but it helps us determine whether cache contention introduces a bottleneck. Although the average overhead with a separate L2 cache decreases to 11.8%, it is not much lower than with no VFC, suggesting that cache contention is not the main bottleneck. Because most of the helper-thread's time is spent computing filter bits, adding a VFC to prevent repeating such work is the more effective solution. The figure also shows that a small VFC helps reduce overhead considerably, but additional VFC space yields little additional benefit. This is because even a small VFC stores filter bits for a large amount of data.

Figure 7(b) shows the performance overhead given different L2 cache sizes (512-KB, 1-MB, and 2-MB) while keeping the cache associativity constant and a fixed 32-KB VFC. The overhead is always relative to the configuration with the same cache size, but without any bug checking. The average overheads are 6.9% for 512 KB, 5.0% for 1 MB, and 5.6% for 2 MB. In general, larger caches result in fewer cache misses, fewer filter bits "lost" due to cache replacements, fewer bug-check requests, and less HeapMon activity. However, since baseline execution speeds up significantly with larger caches, the relative HeapMon overhead can increase (e.g., to 5.6% for 2-MB caches). Overall, the results suggests that HeapMon performance overhead remains low as long as the application has good caching behavior.

Bug-detection capability

HeapMon tags heap words with states adopted from Purify; therefore, their bug-detection capability is similar, and we expect HeapMon to detect all bugs that we inject.

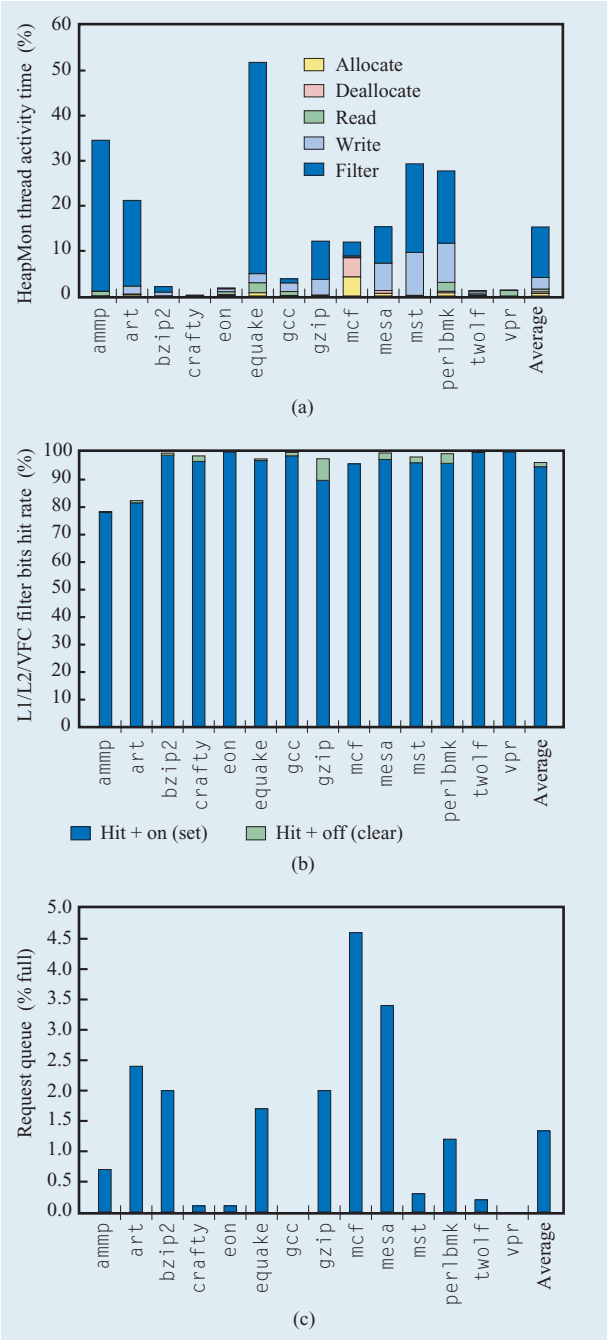


Figure 6

(a) Breakdown of the HeapMon helper-thread execution time by request type; (b) percentage of accesses that find their filter bit in the L1/L2/VFC; (c) percentage of time during which the request queue is full.

We use Table 1 to choose a representative application for each of the following behaviors: few but very large allocations (`mcf`), few and small allocations (`crafty`),

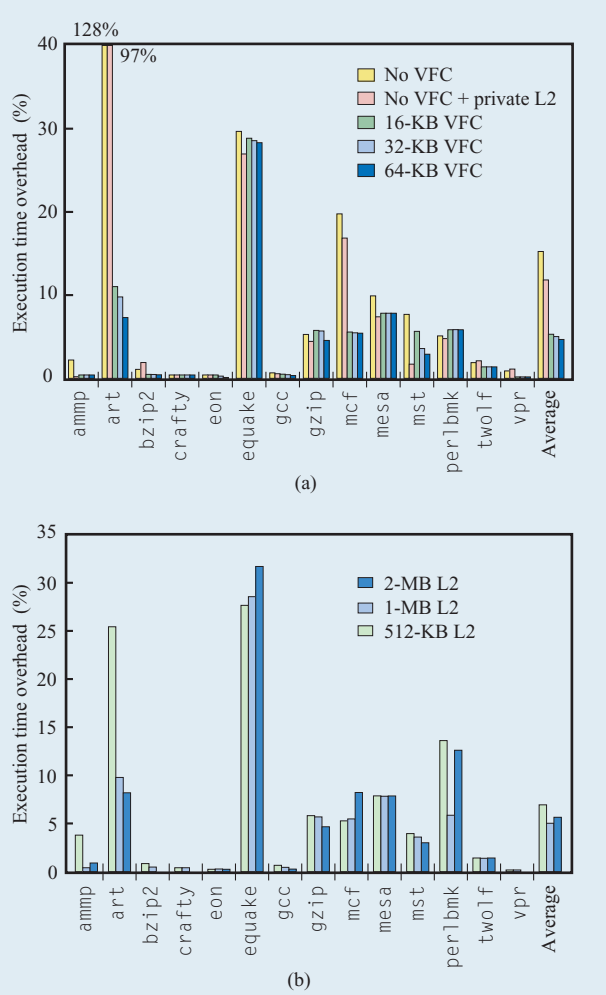


Figure 7

Sensitivity of execution time overhead of HeapMon in always-on mode for (a) different VFC sizes; (b) different L2 cache sizes.

many small allocations (`art`), and a very high number of allocations (`equake`). We introduce three types of bugs: alloc-bug reduces randomly selected memory allocation requests by 32 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 so it does not crash the application, allowing HeapMon to detect bugs that are injected later.

HeapMon detected all injected bugs except for five alloc-bugs in `crafty` (**Table 4**). These injections reduce allocation size for heap objects that are used as string buffers, and the 32 missing bytes are never used. As a result, the injected bug is never manifested and is not detected. This result once again illustrates the elusiveness of some bugs, which may be manifested only in production environments that stress the application to its limits. This observation supports the argument that a low-overhead bug detection deployable in production environments is needed.

We also tested bug-detection capability on set B applications using the input sets from [23]. The applications ncompress-4.2.4 and polymorph-0.4.0 have reported stack-related bugs but no heap-related bugs (that we know of), and HeapMon did not detect any bugs with the input sets used. However, HeapMon successfully detected the two previously reported [23] heap-related bugs in bc-1.06 as writes to unallocated locations.

We also ran set A applications (SPEC2000) through HeapMon without bug injection. HeapMon was able to

Table 3 Average service time (in cycles) of the helper thread for each type of heap request. N/A indicates that the helper thread never encounters the request. For example, `bzip2`, `equake`, and `mst` do not have deallocation requests because they never deallocate their heap objects, whereas in `bzip2` and `mst`, filter bits eliminate all read requests.

Application	*Allocation*	*Deallocation*	*Read*	*Write*	*Filter-bit requests*
`ammp`	179	1,791	19	28	305
`art`	69	3,789	21	24	181
`bzip2`	201,956	N/A	N/A	33	230
`crafty`	4,206	3,204	480	20	257
`eon`	159	155	70	76	532
`equake`	61	N/A	20	24	214
`gcc`	655	483	26	26	193
`gzip`	5,469	251	54	23	236
`mcf`	1,850,878	2,627,350	15	48	329
`mesa`	58,146	64,971	26	20	221
`mst`	6,242	N/A	N/A	38	264
`perlbmk`	3,825	7,484	48	20	225
`twolf`	72	36	18	27	157
`vpr`	83	42	47	28	178

Table 4 HeapMon bug-detection capability showing the percentage of inserted bugs detected and the number of injected bugs.

Benchmark	*alloc-bug* [% (no.)]	*dealloc-bug* [% (no.)]	*noinit-bug* [% (no.)]
`art`	100 (20)	100 (5)	100 (20)
`crafty`	61 (13)	100 (4)	100 (20)
`equake`	100 (20)	100 (4)	100 (20)
`mcf`	100 (3)	100 (3)	100 (20)

Table 5 Total memory not deallocated in the benchmarks tested. HeapMon lumps together true memory leaks and objects that are intentionally left allocated by the programmer.

Benchmark	*Allocations*	*Not deallocated*	*Not deallocated* (%)	*Total memory leak* (bytes)
ammp	34,766	34,764	99.9	13,867,154
art	30,490	30,489	99.9	2,276,734
bzip2	13	13	100.0	13,667,680
crafty	44	42	95.5	881,680
eon	2,595	632	24.4	53,720
equake	316,854	316,854	100.0	11,406,744
gcc	4,305	1,180	27.4	3,001,720
gzip	246	7	2.9	6,605,308
mcf	10	3	30.0	4,600
mesa	69	9	13.0	13,836
mst	422	422	100.0	13,681,240
parser	105	7	6.7	31,470,635
perlbmk	423	323	76.4	4,233,204
twolf	9,413	1,408	14.9	7,821,619
vpr	1,594	50	3.1	80,462

identify all objects that were left allocated when the applications terminated in each of the 15 benchmarks. **Table 5** shows, for each benchmark, the number of memory allocations made, the number of these that are not deallocated (both as an absolute number and as a percentage of allocations), and the total number of bytes that remain allocated at the end of program execution. Note that HeapMon does not distinguish between true memory leaks and heap objects that are intentionally left allocated by the programmer. HeapMon identifies that six benchmarks (ammp, art, bzip2, crafty, equake, and mst) either never deallocate their heap memory or deallocate fewer than 5% of their allocations.

Conclusions

This paper presents HeapMon, a heap memory bug-detection scheme that has a very low performance overhead, is automatic, and is easy to deploy. HeapMon relies on two new techniques. First, application execution is decoupled from bug monitoring, which executes as a helper thread on a separate core in a CMP system. The second new technique in HeapMon is to associate a filter bit with each cached word to safely reduce bug-checking frequency (by 95% on average). Our experimental results show that HeapMon effectively detects and identifies most forms of heap memory bugs and incurs an average performance overhead of only 5% on SPEC2000 applications. Such an overhead is orders of magnitude smaller than in existing tools. HeapMon requires modest on-chip storage overhead: 3.1% of the cache size and a 32-KB victim cache for on-chip filter bits. It also uses a software data structure for state bits, which uses only 6.2% of the allocated heap memory size.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Standard Performance Evaluation Corporation, Microsoft Corporation, or Macromedia, Inc. in the United States, other countries, or both.

References

1. IBM Corporation, IBM Rational PurifyPlus for UNIX; see *http://www-306.ibm.com/software/awdtools/purifyplus/*.
2. P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torellas, "iWatcher: Efficient Architectural Support for Software Debugging," *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004; see *http://opera.cs.uiuc.edu/paper/ZhouISCA04.pdf*.
3. T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,* 1994, pp. 290–301.
4. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proceedings of the 7th USENIX Security Symposium*, 1998, pp. 63–78.
5. S. Hangal and M. S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proceedings of the International Conference on Software Engineering*, 2002, pp. 291–301.
6. Intel Corporation, Intel Thread Checker; see *http://www.intel.com/cd/software/products/asmo-na/eng/threading/219783.htm*.

7. A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps, "Debugging via Run-Time Type Checking," *Lecture Notes in Computer Science* **2029**, 217–232 (2001).
8. G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 128–139.
9. H. Patil and C. Fischer, "Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs," *Software Pract. & Exper.* **27**, No. 1, 87–110 (1997).
10. H. Patil and C. N. Fischer, "Efficient Run-time Monitoring Using Shadow Processing," *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, 1995, pp. 119–132.
11. IBM Corporation, Rational software; *see http://www-306.ibm.com/software/rational/*.
12. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Syst.* **15**, No. 4, 391–411 (1997).
13. Valgrind Developers, Valgrind; see *http://valgrind.kde.org/*.
14. J. Boletta, SecurityFocus Newsletter No. 172, 2002; see *http://www.pantek.com/library/linux/lists/securityfocus.com/sf-news/msg00002.html*.
15. Symantec Corporation, "Microsoft IIS HTR Chunked Encoding Heap Overflow Allows Arbitrary Code"; see *http://securityresponse.symantec.com/avcenter/security/Content/2033.html*.
16. United States Computer Emergency Readiness Team, "Buffer Overflow in Microsoft Internet Explorer," *Technical Cyber Security Alert TA04-315A*; see *http://www.us-cert.gov/cas/techalerts/TA04-315A.html*.
17. United States Computer Emergency Readiness Team, "'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL," *FedCIRC Advisory FA-2001-19*; see *http://www.us-cert.gov/federal/archive/advisories/FA-2001-19.html*.
18. M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002, pp. 75–88.
19. U. Stern and D. L. Dill, "Automatic Verification of the SCI Cache Coherence Protocol," *Proceedings of the Conference on Correct Hardware Design and Verification Methods*, 1995, pp. 21–34.
20. J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 258–269.
21. D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 237–252.
22. S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A System and Language for Building System-Specific, Static Analyses," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 69–82.
23. P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torellas, "AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants," *Proceedings of the 37th International Symposium on Microarchitecture*, 2004, pp. 269–280.
24. F. Qin, S. Lu, and Y. Zhou, "SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs," *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005; see *http://www.hpcaconf.org/hpca11/papers/28_qin-safemem.pdf*.
25. M. L. Corliss, E. C. Lewis, and A. Roth, "Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE," *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 303–314.
26. J. Oplinger and M. S. Lam, "Enhancing Software Reliability with Speculative Threads," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 184–196.
27. J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic Speculative Precomputation," *Proceedings of the 34th International Symposium on Microarchitecture*, 2001, p. 306.
28. Y. H. Song and M. Dubois, "Assisted Execution," *Technical Report No. CENG 98-25*, Department of Electrical Engineering—Systems, University of Southern California, Los Angeles, CA 90089, 1998.
29. D. Kim, S. S.-W. Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors," *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, p. 27.
30. C.-K. Luk, "Tolerating Memory Latency Through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001, pp. 40–51.
31. A. Roth and G. Sohi, "Speculative Data-Driven Multithreading," *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, pp. 37–48.
32. Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," *Proceedings of the 29th International Symposium on Computer Architecture*, 2002, pp. 171–182.
33. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *Proceedings of the 4th International Conference on Supercomputing*, 1990, pp. 1–6.
34. W. Hillis and L. Tucker, "The CM-5 Connection Machine: A Scalable Supercomputer," *Commun. ACM* **36**, No. 11, 31–40 (1993).
35. S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994; see *http://www.eecs.umich.edu/~stever/pubs/isca94_typhoon.pdf*.
36. E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally, "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993; see *http://citeseer.csail.mit.edu/cache/papers/cs/1282/http:zSzzSzwww.cs.ucsb.eduzSz~schauserzSzpaperszSz93-isca.pdf/spertus93evaluation.pdf*.
37. W. J. Dally, L. Chao, A. Chein, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a Message-Driven Processor," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987, pp. 189–196.
38. E. Witchel, J. Cates, and K. Asanovic, "Mondrian Memory Protection," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 304–315.
39. SPEC Benchmarks, Standard Performance Evaluation Corporation; see *http://www.spec.org*.
40. SESC; see *http://sesc.sourceforge.net*.

Received June 21, 2005; accepted for publication July 25, 2005; Internet publication February 28, 2006

Rithin Shetty *Network Appliance, Inc., 495 East Java Drive, Sunnyvale, California 94089 (rithin@gmail.com)*. Mr. Shetty is a member of the technical staff, working with the Enterprise Data Management business unit on disk-based backup and mirroring technologies. He received a B.E. degree in computer science and engineering from Sri Jayachamarajendra College of Engineering, India, and an M.S. degree in computer science from North Carolina State University.

Mazen Kharbutli *Department of Electrical and Computer Engineering, North Carolina State University, P.O. Box 7256, Raleigh, North Carolina 27695 (mazen.kharbutli@gmail.com)*. Mr. Kharbutli received a B.S. degree in electrical and computer engineering from the Jordan University of Science and Technology, and an M.S. degree in electrical engineering from the University of Maryland. He is currently pursuing a Ph.D. degree in computer engineering at North Carolina State University.

Yan Solihin *Department of Electrical and Computer Engineering, North Carolina State University, P.O. Box 7256, Raleigh, North Carolina 27695 (solihin@ece.ncsu.edu)*. Dr. Solihin received a B.S. degree in computer science from the Institut Teknologi Bandung, Indonesia, an M.S. degree in computer engineering from Nanyang Technological University, Singapore, and M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana–Champaign. He is currently an assistant professor in the Department of Electrical and Computer Engineering, North Carolina State University. He has published more than 25 papers in computer architecture and image processing, which cover chip multiprocessor systems, performance modeling, interaction of architecture and systems software, and architecture support for security and software reliability.

Milos Prvulovic *College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, Georgia 30332 (milos@cc.gatech.edu)*. Dr. Prvulovic is an assistant professor at the College of Computing at the Georgia Institute of Technology. He received a B.Eng. degree in electrical engineering from the University of Belgrade, and M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana–Champaign. He has published numerous papers on computer architecture, primarily in the area of architectural support for thread-level speculation, reliability, and programmability of multiprocessors.