Pesticide: Using SMT to Improve Performance of Pointer-Bug Detection

Jin-Yi Wang, Yen-Shiang Shue, T. N. Vijaykumar, and Saurabh Bagchi
School of Electrical and Computer Engineering, Purdue University
{jywang, shue, vijay, sbagchi}@ecn.purdue.edu

***Abstract*—Pointer bugs associated with dynamically-allocated objects resulting in out-of-bounds memory access are an important class of software bugs. Because such bugs cannot be detected easily via static-checking techniques, dynamic monitoring schemes have been proposed. However, the key challenge with dynamic monitoring schemes is the runtime overhead (slowdowns of the order of 10x are common). Previous approaches have used thread-level speculation (TLS) to reduce the overhead. However, the approaches still incur substantial slowdowns while requiring complex TLS hardware. We make the key observation that because the monitor code and user code are largely and unambiguously independent, TLS hardware with all its complexity to handle speculative parallelism is unnecessary. We explicitly multithread the monitor code in which a thread checks one access and use SMT to exploit the parallelism in the monitor code.**

Despite multithreading the monitor code on SMT, dynamic monitoring slows down the user thread due to two problems: instruction overhead and insufficient overlap among the monitor threads. To address instruction overhead, we exploit the natural locality in the user thread addresses and memoize recent checks in a small table called the allocation-record-cache (ARC). However, programs making and accessing many small memory allocations cause many ARC misses and incur significant runtime overhead. To address this issue, we make a second key observation that because adjacent memory objects result in ARC entries with contiguous address ranges, the entries can be merged into one by simply merging the ranges into one. This merging increases the effective size of the ARC. Finally, insufficient overlap among monitor threads occurs because of inefficient synchronization to protect the allocation data structure updated by the user thread and read by the monitor threads. We make the third key observation that because monitor-thread reads occur for every check but user-thread writes occur only in allocations and deallocations, monitor reads are much more frequent than user writes. We propose a locking strategy, called biased lock, which puts the locking overhead on the writer away from the readers. We show that starting from a runtime overhead of 414% our scheme reduces this overhead to a respectable 24% running three monitor threads on an SMT using a 256-entry ARC with merging and biased lock.

I. INTRODUCTION

According to the National Institute of Standards (NIST) "software developers spend approximately 80% of development costs on identifying and correcting defects and yet few products of any type other than software are shipped with so many errors" [1]. To address this problem, static checking tool, such as [9], [11], [15], [17] attempt to detect and remove software bugs in the testing and verification phase. However, NIST reports that more than half of bugs are not found until "downstream" in the development process or during post-sale use of software. This figure will worsen as software becomes more complex. To address the large percentage of bugs slipping through static checking, dynamic monitoring schemes, such as [10], [16], [25], [26], [19] and [8] attempt to detect bugs at runtime.

A detailed study of software defects in commercial database management systems and operating systems [21] reports that as many as half of the "high-impact" bugs are in dynamic memory allocation and pointer management. The study defines high-impact bugs as those that often results in system unavailability. The memory allocation and pointer management bugs cause out-of-bounds memory access. Accordingly, we target pointer bugs associated with dynamically-allocated objects resulting in out-of-bounds memory access. Because static checking cannot detect such bugs easily, we explore a dynamic monitoring scheme.

While implementing dynamic monitoring is relatively straightforward, the key difficulty is the runtime overhead. Though dynamic monitoring is powerful and can catch hard-to-find bugs, its considerable overhead has limited its applicability to in-house testing. As such, dynamic monitoring is too slow to be used in production runs. For instance, though Purify, which uses software for monitoring without any support from hardware, has been extremely successful in catching pointer bugs, 1000% slowdowns are common. Recent dynamic-monitoring schemes, such as [16], iWatcher [26], and AccMon [25], propose to address this performance problem by leveraging thread-level speculation (TLS). These schemes use TLS to overlap speculative threads spawned from the user computation as well as the monitoring code. [16] uses Dynamic Multi Threading (DMT) [2] as its TLS architecture, and iWatcher and AccMon use [20] as their TLS architectures.

However, there are two shortcomings with these TLS-based schemes: (1) TLS schemes introduce considerable complexity: DMT uses associative searches through large trace buffers for dependence tracking and flash copying of register values. [20] pushes speculative state into the cache hierarchy and requires complicated dependence tracking through the cache hierarchy. (2) Despite using TLS, dynamic monitoring still inflicts considerable performance loss. [16] reports 700% runtime overhead over no monitoring, and [25] incurs 200% for three SPEC2000 programs. While [16] targets dynamic monitoring during production runs, [25] targets the debugging phase of software development. Irrespective of the intended target, their high performance degradation coupled with their reliance on complex TLS hardware makes it hard to deploy them in production runs.

In this paper we address these shortcomings in the context of monitoring out-of-bounds memory access due to pointer bugs based on the following key observations: (1) The main computation and the monitoring code are unambiguously and truly independent except that the addresses of the main computation's memory accesses need to be passed to the monitoring code. More importantly, each dynamic instance of the monitoring code meant to check one memory access is unambiguously and truly independent of other instances. (2) Exploiting ambiguous, speculative parallelism and incurring its complexity is unnecessary when explicit,

unambiguous parallelism exists. Accordingly, we propose to multi-thread the monitoring code in which a thread checks one memory access, and to use SMT [23] to exploit the explicit parallelism in our monitoring code. Because SMT exploits explicit parallelism instead of speculative parallelism, SMT is considerably simpler than TLS.

Another choice to exploit explicit parallelism would be chip multiprocessors (CMPs), as [19] does. However, memory addresses need to be passed from the user thread to the monitor threads. In CMPs, the user and monitor threads would run on different cores, requiring high-bandwidth (potentially specialized) communication paths between the cores. To avoid this problem, we choose SMT where the user and monitor threads run within one core and can communicate easily.

Despite running multiple monitor threads overlapped with the user thread, dynamic monitoring slows down the user thread due to two problems: instruction overhead and insufficient overlap among the monitor threads.

Because each check involves tens of instructions, monitoring incurs the first the problem of instruction overhead. Though user and monitor threads are independent, because all threads run on one SMT core the monitor threads compete with the user thread for execution resources causing substantial performance degradation. To reduce the volume of monitor instructions, we exploit the natural locality in the user addresses and memoize recent checks in a small table called the *allocation-record-cache (ARC)*. When a user address hits in the ARC, the hardware checks the access without invoking a monitor thread, avoiding extra instructions.

This locality was also exploited by AccMon in its check-lookaside-buffer (CLB). However, because each entry in CLB (and ARC) correspond to a memory object, programs allocating and accessing many objects need a large CLB (and ARC). In an attempt to reduce the size of the CLB, AccMon implements a bloom filter which results in false positives. However, while AccMon is used for debugging where the false positives will not reach the user, pesticide is for production user runs where users will not tolerate false positives unnecessarily terminating their program. Consequently, we propose a scheme without false positives. We make the key observation that because adjacent memory objects result in ARC entries with contiguous address ranges, the entries can be merged into one by simply merging the ranges into one. Because the set of valid address ranges are derived from memory allocation functions in software, we perform this merging in software in the monitoring code. Merging enables ARC to cover more memory objects with fewer entries.

The second performance problem is the insufficient overlap among the monitor threads. Although monitor threads are largely independent of the user program there still exists some synchronization. The monitor threads have to read the data structure holding the current set of valid memory allocations where as the user thread writes to the structure when allocating or deallocating memory (e.g., malloc and free). Using the standard reader-writer-lock [6] for this synchronization causes inordinate contention among the monitor threads. To address this issue, we make the key observation that because monitor-thread reads occur for every check but user-thread writes occur only in malloc and free, monitor reads are much more frequent than user writes. Accordingly, we employ a novel locking strategy, called *biased lock*, in which each monitor thread has its own lock for reading whereas the user thread has to obtain all the monitor-thread locks before writing. This biased strategy makes readers fast at the cost of the writer which fits our context of frequent monitor reads and infrequent user writes.

The key novelty of this paper are the range merging ARC and biased lock. Our simulation results show that starting from an average runtime overhead of 414% incurred by monitoring over no monitoring, pesticide reduces this overhead to a respectable 24% using a 256-entry ARC and three monitor threads. This 24% overhead compares well with [16]'s 700% and [25]'s 200% and also with the fact the Java which performs bounds-checks in-line in the user code incurs about 100% runtime overhead [4], [24]. Because pesticide checks all heap accesses, it covers all out-of-bounds heap accesses without any false positives.

The roadmap for the rest of the paper is as follows. In Section II we describe our software and hardware architecture. In Section III, we describe the ARC, the biased locks and the merging scheme. We present our evaluation methodology in Section IV and our results in Section V. Section VI describes related work and we conclude in Section VII.

II. OUR SCHEME

In general, pointer bugs can be associated with heap, stack, or static objects, However, because the study on commercial software, mentioned in Section I, shows that heap objects account for about half of all the "high-impact" bugs [21], we monitor only heap objects and not the stack or static objects. Moreover, there are efficient schemes to monitor the stack (e.g., [7] protects against buffer overflow with minimal performance degradation). As we explain later, pesticide can apply to static objects.

A. Overview

To achieve our goal of monitoring user-thread heap accesses, we need to track user-thread heap allocations and deallocations, and check whether user-thread accesses fall within a valid allocation. We track the memory allocations and deallocations in a hash table called the book-keeping-structure (BKS). To perform the checking, our monitor threads run concurrently with the user thread on a SMT processor. Upon a load or store instruction in the user thread, pesticide triggers a monitor thread to check the address and the length of the access. Multiple instances of the monitor thread run on the SMT processor to check multiple accesses in parallel (the number of monitor threads is fixed). Each monitoring thread matches the address and length of the access against the BKS entries. A match (i.e., access is to a valid address and access length is within allocated size) indicates that the access is legal. A mismatch indicates a pointer bug. Figure 1 shows a block diagram of pesticide.

We now give the details of our software in Section II-B and hardware structures in Section II-C.

B. Software support

In order for application programmers not having to worry about monitoring routines, we propose that library functions be instrumented with monitoring capabilities. We augment library calls to memory-management routines (e.g., *malloc*, *calloc*, *realloc*, and *free*) with code to maintain the BKS.

The BKS is a hash table that tracks valid memory address ranges by recording memory allocations and deallocations (the addresses are virtual addresses annotated with process ID to allow for multiple concurrent user processes). Every allocation creates a

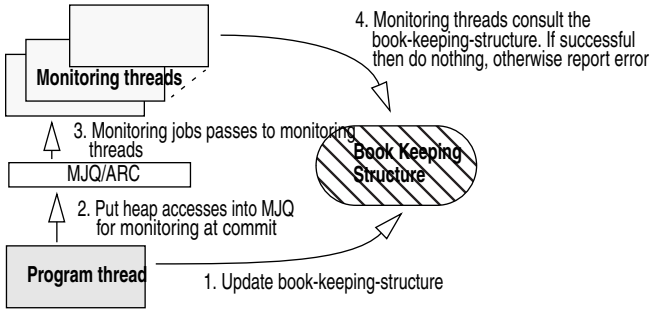

Figure 1: Proposed monitoring scheme

new BKS entry which is removed at the corresponding deallocation. Each entry contains the start address and the length of the allocation.

The BKS is fundamentally different from conventional hash tables. In a conventional hash table, an object that is hashed into the hash table is found by using the *same* object as the key. In BKS, while we hash in allocations' address ranges, accesses to a specific address probe the BKS to determine if a given address falls within a valid range. Thus there is a disparity between what is stored (i.e., address ranges) and what is used to probe (i.e., a specific address). This disparity implies that if we hash in a long address range using the range's start address, and an access far from the start but within the range occurs, then the range and the access may fall into different hash buckets resulting in the access not finding the range though the access is valid. One way to solve this problem is that we could repeatedly hash each byte of the whole range of the allocation and store all of them in the BKS. However, this approach would result in inordinately many copies of the same range and blow up the hash table size. Instead, we break the original allocation's address range into many small ranges called hash-blocks (e.g, 512-byte ranges). We hash all the hash-blocks of the original allocation into the BKS using the hash-block's start address. Consecutive hash-blocks fall into consecutive buckets, and a BKS entry corresponds to one hash-block. Upon an access, we use the hash-block number part of the access's address (e.g., hash-block of 512 bytes and a 32-bit address mean that the upper 23 bits of the address are the hash-block numbers) to probe the BKS. Because the addresses within a hash-block have the same hash-block number as the hash-block's start address, accesses to a hash-block map to the same bucket as that holding the hash-block. Though our solution allows the access to find its corresponding range, we break up long address ranges into many hash-blocks, each of which repeatedly store the original range, increasing the hash table size.

We use simple open chains to handle collisions. Because access probes need to traverse the chains to ascertain validity of the access, the longer the chains the more the monitoring overhead. Therefore, it is important to keep the chains short. While a good hash function is necessary for this purpose, it is not sufficient. Hash-block size has a considerable impact on the chain length. Both small and large hash-blocks result in long chaining but in different scenarios. Small hash-blocks break up larger allocations into many BKS entries (albeit in different buckets) resulting in much chaining. Large hash-blocks imply that many smaller allocations fall into the same bucket resulting in much chaining. Thus, the hash-block size has to match the allocation size commonly found in programs.

One simple optimization we can do is that, upon an check, we move a hash element to the top of its hash chain in anticipation that the hash element will be accessed again due to locality. This move reduces the chain traversal in subsequent accesses.

C. Hardware support

Because the volume of memory accesses to be checked is high and the check itself is fairly short especially if the BKS chains are short (e.g., a few tens of instructions), using software to spawn monitor threads or to pass the addresses from user to monitor thread would incur considerable overhead. Instead, we employ hardware support in the form of the *monitor-job-queue (MJQ)* which captures the user thread's addresses off the pipeline and triggers a monitor thread to check the access.

The MJQ is a FIFO queue built in hardware. The queue buffers address (virtual address and process ID) and the length of the heap accesses to be monitored. As we mentioned before, we check only heap accesses, and not stack and static accesses. The MJQ determines an address to belong to the heap if the address lies between the heap bottom and heap ceiling. Address and length of the access whether it is a byte, word, or quadword are then taken from the load-store queue.

Monitoring could potentially be triggered at any point in the execution of loads and stores. Monitoring before commit would include misspeculate loads and stores along with the correct loads and stores leading to wastage of SMT resources as much of the monitoring would be unnecessarily triggered. Therefore, we check at commit. Because loads and stores stay in the load-store queue until commit, we readily obtain the addresses and the access lengths from the load-store queue.

While the checking itself is independent of the user thread once the access address and length are given to the monitor thread, the decision of whether to allow the access to commit before the check completes or not impacts the parallelism between the monitor and user threads. While loads does not cause bugs to spread to other programs, stores may propagate bugs via I/O. Delaying the commit till the check completes prevents a buggy store from propagating further. Because stores are frequent enough, this option would curtail the parallelism between user and monitor threads and would severely slow down the user threads. Instead, to retain the parallelism, we do not hold up store commits till the check completes. Thus, accesses may commit before the check completes. (e.g., a few hundreds of cycles). To prevent bug propagation, we ensure that all checks pending in the MJQ are complete before any system call, including I/O call, is committed. Because system calls are infrequent this delay in commit does not significantly impact performance.

Apart from system calls, the other point where the user thread waits for the monitor thread is upon memory deallocations. Deallocating a heap object while there are pending checks of accesses to the object would cause us to flag a bug incorrectly. Consequently, we ensure that all pending checks complete before the deallocation starts. While hardware can easily detect system calls as special opcodes and trigger the draining of the MJQ, deallocation functions are indistinguishable from other functions in the user thread. To that end, we use a special NOP to signal the beginning of a deallocation. Because deallocations are also infrequent, delaying the deallocation till all pending checks are complete does not significantly impact performance.

We mentioned earlier that we can easily extend pesticide to static objects. Because static objects' address ranges are known at link time, the linker can insert the ranges into the BKS.

III. SUPPORTING EFFICIENT MONITORING

The key reasons for performance degradation in the basic scheme described so far are instruction overhead and insufficient overlap among monitor threads. We alleviate these problems via our optimizations.

A. Allocation-record-cache (ARC)

To reduce the instruction overhead of monitoring, we exploit the locality in the user-thread accesses to memoize checks to recently-accessed heap objects so that future checks to the same objects are elided completely and the instruction count overhead of monitoring is reduced. We use a hardware cache, called the allocation-record-cache (ARC), for this memoization.

Before inserting an access into the MJQ, the address of the access is first checked in the ARC. Upon a hit, the ARC performs the check in hardware. Consequently we do not place the access in the MJQ, saving the instructions of the check. A miss launches a monitor thread which performs the check in software but also loads the ARC with the BKS entry used to perform the check.

Each ARC entry holds a BKS entry: the start address and the length of the allocation. However, there is a key difference between the BKS and ARC entry. To avoid the danger of an address not finding its range in the BKS, both the address and range are hashed by their hash-block number. Consequently, each hash entry can cover only a hash-block implying that large allocations be broken up into multiple hash-blocks, introducing repetition in the hash entries. Because the ARC is a small cache, such repetition would be wasteful. Instead we use a fully-associative cache so that there is no indexing into the ARC. Because every access searches through all of ARC's entries, there is no danger of an access not finding its range in the ARC. Consequently, each entry in the ARC is not restricted to covering one hash-block, implying that an ARC entry can cover an entire allocation without breaking up the allocation across multiple ARC entries.

Thus, the address of an access is matched against all the ARC's entries in parallel, checking if the address fall within an entry's start address and the entry's allocation length.

Because the ARC is essentially a cache of the BKS, any modifications to the BKS need to be handled by the ARC for maintaining coherence between the BKS and the ARC. Consequently, the ARC is flushed upon deallocations which are identified by the special NOPs described in Section II-C. As mentioned earlier, deallocations are infrequent, so the flushes do not significantly impact the ARC.

B. Range merging

While the ARC works well for many programs, a few programs make and access many (small) allocations. Because one ARC entry can hold only one allocation, small allocations imply that the ARC can reach only a small part of the user thread's working set; and many allocations imply that many ARC entries would be needed. The net effect is many misses in the ARC. Many allocations also implies long hash collision chains in the BKS. Here again, the long collision chains is not due to an ineffective hash function, but rather due to the fact the one hash entry can hold only one allocation.

We exploit the fact that the BKS entries for adjacent heap objects can be merged to increase the effective capacity of the ARC. This merging has the additional benefit of shortening the hash collision chains.

Because memory allocations and deallocations are tracked by the BKS, we perform this merging in software in the BKS code. To implement merging, we keep the BKS entries in the collision chains in ascending order of starting addresses. Upon new allocations, an insert into a chain merges entries if two entries contain contiguous ranges. Upon deallocations, a previously-merged entry may be broken into two entries. As such, merging increases the overhead of the BKS code. Because allocations and deallocations are relatively less frequent than accesses which benefit from the improved effective ARC size and shorter chain lengths, merging improves performance (allocations need not be infrequent in the absolute just fewer than accesses). Also, we do not perform the locality optimization of moving the hash element to the top of the hash chain as such moving will violate the ordering of the start addresses.

There is one difficulty with merging: Because memory allocators often allocate memory objects padded to a size larger than requested for reducing memory-management overhead, storing heap-management-related meta information, and alignment reasons, merging padded ranges would result in letting some bugs go undetected. If an access falls in the padding which is in the middle of a merged range then the access is invalid, but we cannot detect it to be so. To address this problem, we first make the key observation that same-sized objects are adjacent in the common case. This observation implies that in the common case padding would exhibit a repetitious pattern in the merged ranges. Consequently, recording the padding just once for the entire merged range would suffice. Therefore, we merge two ranges only if they are adjacent and they are of the same size and have the same padding. We record the start address of the first allocation, the size of the allocation, and end address of the merged entry. The ARC caches these merged entries.

Ignoring the above observation and merging different-sized objects would mean recording all the paddings within the merged range. This recording even if done via bit vectors would add substantial space overhead, defeating merging's purpose.

C. Biased Locks

Because checking of one access is independent of checking of other accesses, we employ multiple monitor threads in parallel. However, because memory allocation and deallocation routines in the user thread share the BKS with the monitor threads, it is necessary to protect the shared data via proper synchronization. Specifically, the user thread writes and the monitor threads read. However, the two commonly-used locking strategies lead to heavy contention among the monitor threads. The first strategy, called the basic lock, uses one global lock contended by all threads (i.e., user thread and multiple monitor threads), as shown in Figure 2a. There is heavy contention among the multiple monitor threads leading to complete serialization.

The second strategy, called RW, is for multiple readers and writers involved in producer-consumer scenarios (user thread is the producer and the monitor threads are the consumers). As shown in Figure 2b, RW also has a global lock between the reader and the writer. RW attempts to reduce the overhead on the readers by requiring only the first reader to grab the global lock, allowing later

a) Basic lock

Reader:

```
lock (MUTEX);
...critical section...
unlock (MUTEX);
```

Writer:

```
lock (MUTEX);
...critical section...
unlock (MUTEX);
```

b) Reader-Writer lock

Reader:

```
lock(R);
readcount=readcount+1;
if (readcount==1)
  then lock (MUTEX)
unlock(R)
...critical section...
lock(R);
readcount=readcount-1;
if (readcount==0)
  then unlock (MUTEX)
unlock(R)
```

Writer:

```
lock(MUTEX);
...critical section...
unlock(MUTEX);
```

c) Biased lock

Reader # i (1<= i <=n):

```
lock (MUTEXi);
...critical section...
unlock (MUTEXi);
```

Writer:

```
for (i=1; i<=n; i++)
lock (MUTEXi)
...critical section...
for (i=1; i<=n; i++)
unlock (MUTEXi)
```

Figure 2: Three lock schemes

TABLE I: Simulation parameters

Simulator Parameters	
fetch width	8
decode width	8
issue width	8
commit width	8
active list size (per thread)	256
LSQ size (per thread)	128
issue queue	64
L1 I-cache	64K, 2way, 3cycle
L1 D-cache	64K, 2way, 3cycle
L2 unified	2M, 8way, 12cycle
Memory Latency	300 cycles
Branch prediction	2-level hybrid
MJQ size	1000
ARC size	0,8,256,1024

readers to avoid grabbing the global lock. However, the readers among themselves have to grab a local lock to identify the first reader. Because the local lock protects just an increment operation, the local lock does not serialize the readers much. If the original critical section to be protected from the writer is long, the local lock overhead is amortized. Unfortunately, because our monitor threads are very short, the local lock overhead is not amortized.

We make the key observation that our writer (i.e., user thread) is much less frequent than readers (i.e., monitor thread). Therefore we bias the overhead away from the readers and towards the writer in the lock called biased-lock, as shown in Figure 2c. In the biased-lock each reader is given its own lock. So there is no contention among the readers. The writer on the other hand needs to grab the locks of *all* the readers. This ensure mutual exclusion and at the same time allows the monitors to be accessed without too much overhead. With biased lock, we do not perform the locality optimization of moving the hash element to the top of the hash chain as such moving will make monitor threads also writers of BKS, breaking our assumption that monitor threads can execute in parallel completely.

In SMT, resources are shared across threads. It is wasteful for a thread to be spin-waiting on a lock because instructions which go repeatedly into the pipeline will only confirm that the lock is still not available. We want the thread waiting for this lock to stall so the thread will not eat up resources which would be allocated to other threads which may make progress. Such a stalling scheme is implemented in Lock-box [22] which we use. The lock-box stalls a thread on a busy bit when the lock is already taken. Upon the unlock instruction, the bit is cleared and the waiting thread is signaled to go ahead.

IV. EVALUATION METHODOLOGY

We use a SMT simulator based on the Simplescalar 3.0c [5] running the Alpha instruction set to simulate pesticide. Our simulation parameters are shown in Table I.

We use SPEC2000CPU benchmark set. Because we focus on heap accesses, we do not consider Fortran-77 benchmarks which does not have dynamic allocations. Due to time constraint, we simulate only C benchmarks and not C++. We create benchmark binaries with and without monitoring incorporated into the memory management libraries. To ensure that both versions have the same level of compiler optimizations, we compile the benchmarks using gcc2.97 on a DEC Alpha running OSF.

The key software parameters are hash-table size and hash-block size. We use a hash-table with 64K buckets which are sufficient for our benchmarks. We found that the best hash-block size is 512 bytes which we use in all experiments except while varying the hash-block size.

We incorporate early SimPoints [18] in our simulations. Because of the instruction-count overhead of monitoring, the no-monitoring and monitoring versions of the benchmarks execute different *total* number of instructions for the same Simpoints. We ensured that the two versions run the same *user* instructions as intended by SimPoints.

V. RESULTS

Because performance is the key concern for dynamic monitoring, we present performance achieved by pesticide. We do not show coverage because by design pesticide covers all out-of-bounds heap accesses. Also, we do not incur any false positives.

Section V-A presents the unoptimized, raw impact of monitoring on performance. Section V-B shows how running multiple monitor threads impacts performance—with different locking strategies. These numbers show the benefit of using explicit parallelism. Because the hash-block size impacts the hash chain lengths which directly impacts the instruction overhead of monitoring, we vary the hash-block size in Section V-C. Section V-D shows the benefit of eliding checks via ARC's memoization. Section V-E shows how much merging improves performance by increasing the ARC's reach and also shortening the hash chains. Finally, Section V-F summarizes our results.

A. Runtime overhead due to monitoring

In Figure 3, we show the runtime overhead of monitoring. The Y axis shows as percent, the run time of the user thread with one monitor thread normalized to the run time of the user thread with no monitoring. We show a line at the 100% mark which represents no performance degradation due to monitoring. The higher the bars above this line, the more the performance degradation. The X axis shows the benchmarks. Low IPC (instructions per cycle) in the case of no monitoring implies that the pipeline can absorb the extra monitoring instructions. To show this trend, we order the X axis in increasing order of no-monitoring IPC. There are two numbers shown on top of each bar. The top number is the ratio of the dynamic instruction counts with monitoring over the counts without monitoring. The bottom number is the IPC of the benchmarks without monitoring.

With monitoring the benchmarks' runtime overhead range from 5% to 1634% with an average of 414%. Most benchmarks incur significant runtime overhead. There are two factors that

determine performance with monitoring: (1) the dynamic instruction overhead due to monitoring (the top number on top of the bars) and (2) the IPC of the no-monitoring case (the bottom number). Because each check adds about 33 instructions to probe the BKS and determine the validity of the access and because heap accesses are frequent in general, the instruction overhead is usually high. If the instruction overhead is low, as is the case in gzip and crafty, then there is little increase in the runtime due to monitoring. However, if the instruction overhead is high, then there is substantial increase in the runtime even if the no-monitoring IPC is low allowing SMT to absorb the instruction overhead. This trend is true for most of the benchmarks on the left side of the graph such as art, equake, twolf, vpr, parser, and ammp. The only exception to this trend is mcf whose no-monitoring IPC is so low that even a high instruction overhead does not hurt performance. If the no-monitoring IPC is higher, then SMT can absorb the overhead only to a lesser extent, resulting in higher increase in runtime with monitoring. gap, perl, and mesa show this effect.

Thus, monitoring introduces substantial runtime overhead. While programs with low performance such as mcf can absorb monitoring's instruction overhead, we want programs with high performance not to degrade. Therefore, we apply our optimizations to reduce the overhead, both by overlapping monitor threads and by eliminating software checks by memoizing in hardware.

B. Locking strategy

We show the improvements achieved by better locking strategies from Section III-C. Figure 4 shows the runtime normalized to that of no monitoring. For each benchmark, the bars going from left to right represent one monitor thread (same as Figure 3), seven monitor threads with basic lock, with reader-writer lock, and with biased-lock, respectively.

While one would expect runtime to improve with multiple monitor threads, that is not the case for basic locks (e.g., equake). Basic lock incurs contention which offsets the benefits of multiple monitor threads. Comparing basic lock with biased lock, we see that biased lock performs significantly better due to the reduced contention for the readers. This improvement is despite the fact that the monitor code using biased lock requires 42 instructions for each check compared to the 33 required by basic lock. This instruction count increase is because biased locks do not perform the locality optimization done by basic lock of moving the hash element to the top of the hash chain (Section III-C).

While the RW lock performs better than the basic lock, biased lock is better than RW. In RW, the readers incur the overhead of its local lock (Section III-C). Apart from the serialization due to the local lock, RW requires 61 instructions for each check compared to biased-lock's 42. This increase in instruction count is due to the local lock because neither biased lock and RW perform the locality optimization. Thus, the local lock overhead is high.

Comparing one monitor thread with the biased lock, runtime overhead decreases from an average of 414% to 157%.

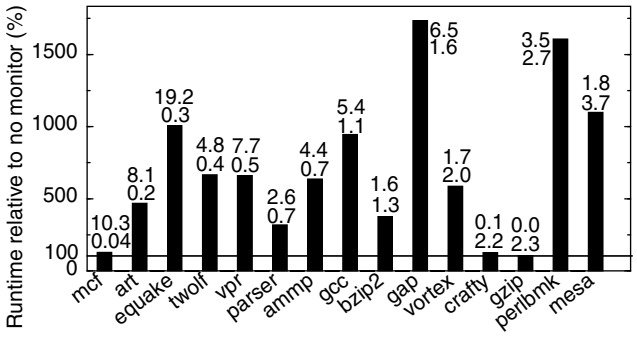

Figure 3: Runtime overhead of monitoring

C. Hash-block size

As discussed in Section II-B, a small hash-block implies long hash chains due to breaking up larger allocation into many hash-blocks and a large hash-block implies long hash chains due to many smaller allocations falling into the same bucket. We varied the hash block size from 256 bytes to 4KB and found that 512 bytes is the best hash-block size for our benchmarks.

D. ARC

To reduce the runtime overhead further, we now use the ARC which exploits locality to reduce the number of checks in software by memoizing recent checks in hardware.

Figure 5 shows the runtime for seven monitor threads using biased locks and 512-byte hash-blocks normalized to no-monitoring case. For each benchmark, the bars going from left to right vary ARC sizes as 0, 8, 256, and 1024 entries. Note that the y-axis scale is different than that of the previous graphs. There are two numbers on top of the bars for each benchmark. The top number is the ratio of the instruction count of monitoring over that of no monitoring, and the bottom number is the ARC miss rate, both for 1024-entry ARC.

We see that even an 8-entry ARC significantly improves runtime over the no-ARC case. For many benchmarks, even 8 entries suffice. For these benchmarks, the ARC miss rates (bottom numbers) are low allowing many checks to be memoized resulting in low instruction overhead (top number). Comparing this overhead with the overhead without the ARC (the top numbers in Figure 3), we see a large reduction. The only exceptions are equake and twolf, both of which have large miss rates even with 1024 entries. In equake's case, there are over 1 million less-than-32-bytes memory allocations which overwhelm the ARC. An 1024-entry ARC could only reach 32KB of equake's 32MB memory footprint.

E. Merging

To increase the effective size of the ARC and to shorten hash chains we employ our merging scheme which merges BKS entries of contiguous allocations of the same size. Figure 6 shows the runtime for seven monitor threads using biased lock, 512-byte hash-blocks, 256- and 1024-entry ARC, and merging and no-merging normalized to no-monitoring case.

Equake and twolf are the two benchmarks which have high

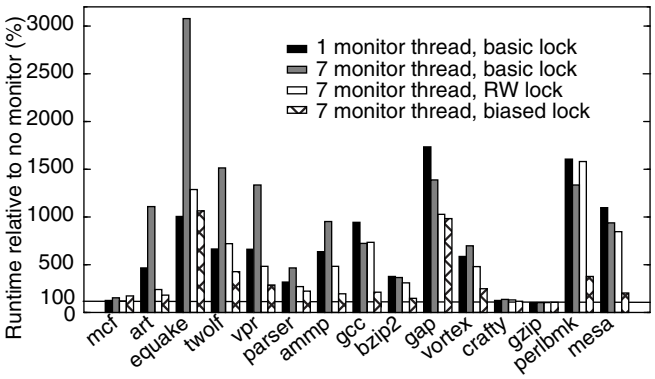

Figure 4: Effectiveness of different locks

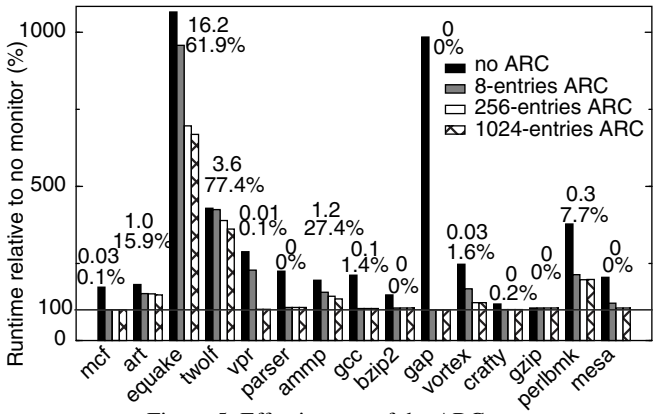

Figure 5: Effectiveness of the ARC

ARC miss rates in Figure 5. Because equake allocates same-sized objects (about 24 bytes), merging works well. Equake's ARC miss rate improves from 62% to 11%, resulting in the runtime overhead almost vanishing. Because twolf's allocation is not as regular as equake's, twolf's improvement is less drastic. As explained in Section III-B, different-sized objects are not meged due to difficulties with padding. By reducing the rutime overhead of equake and twolf, merging provides peformance-robustness to the ARC.

Perl's runtime worsens because of its memory allocation characteristics: perl invokes the realloc() library function often, which not only reduces the possibility of merging, but also causes high instruction overhead when merging is incorporated. This high overhead is the result of realloc() performing both frees and mallocs, both of which incur instruction overhead due to merging.

For the rest of the benchmarks merging is not needed as their ARC miss rates are good to start with. Consequently merging does not improve them, but merging does not hurt them either.

F. Performance summary

Summarizing our results in Figure 7, we show the normalized runtime averaged over the benchmarks. The three groups from left to right, show monitoring with the basic lock, with the biased lock but no merging, and with the biased lock and merging. In each group from left to right we show 1, 3 and 7 monitor threads.

Going from basic lock to biased lock corrects the disadvantageous trend of worsening performance with more threads. In all the groups, the ARC significantly improves performance, and a 256-entry ARC is enough for most benchmarks. Merging improves over the ARC by providing performance-robustness.

We see that we started with a 414% runtime overhead which we reduced to 24% using a 256-entry ARC and 3 monitor threads (the overhead is 18% for 7 monitor threads but using 7 SMT contexts for pointer bugs may be too aggressive, so we highlight 3 threads). This 24% overhead compares well with the 700% overhead for [16] and 200% overhead for [25] which also incurs false positives, and also with the fact the Java which performs bounds-checks inline in the user code incurs about 100% runtime overhead [4], [24]. Because pesticide checks all heap accesses, it covers all out-of-bounds heap accesses without false positives.

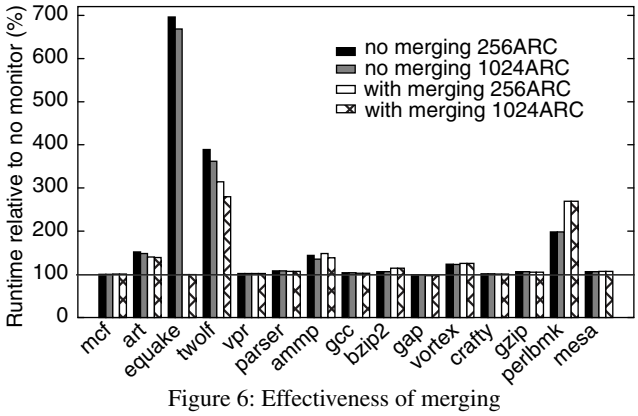

Figure 6: Effectiveness of merging

VI. RELATED WORK

We have discussed [16], iWatcher [26], and Accmon [25]. Previous work on bug detection is broadly divided into two classes, static and dynamic checking. Static checking and analysis for bugs include work from [9], [11], [15] and [17]. We define static checking as those schemes that do not impose any runtime overhead. However, for languages like C, pointer alias problems prevent thorough checking of code during compile time.

In dynamic checking, the earlier proposals are mainly software solutions (e.g., BCC [13] and SafePointer [3]). However both these schemes have substantial runtime overhead (30 times for BCC and 5.4 times for SafePointer). More sophisticated dynamic checking schemes check for program-invariant violations [12] but incur high runtime overhead (e.g., 500%).

Next we discuss dynamic schemes which use some hardware support. DISE [14] is designed for checking whether accesses fall within *coarse-grain contiguous address space* rather than *fine-grain objects*. DISE checks all accesses against the same bound (which can be kept in two registers for the whole execution) whereas we check accesses against individual object boundaries (need to be kept in memory, not registers).

HeapMon [19] also targets out-of-bounds bugs but checks at word granularity whereas we check at byte granularity. Thus, HeapMon would miss out-of-bounds accesses for object sizes that are not multiple of words. As [21] shows, "high-impact" bugs access a few bytes past objects. HeapMon would miss these important bugs. Adding byte granularity to HeapMon would increase its overhead. Additionally, HeapMon uses an extra 128KB cache (much larger than our 256-entry ARC), without which its performance overhead is 17%. However, this 17% cannot be compared with our 24% overhead because Heapmon uses small, unrealistic SPEC2000 *test* inputs while we use realistic *ref* inputs. HeapMon's performance would be worse with larger *ref* inputs.

Mondrian Memory Protection [8] checks memory protection for arbitrary-sized memory blocks, and could be used for pointer bugs. However, Mondrian has no ability to overlap checking with user thread. Implementing Mondrian with additional features (non-

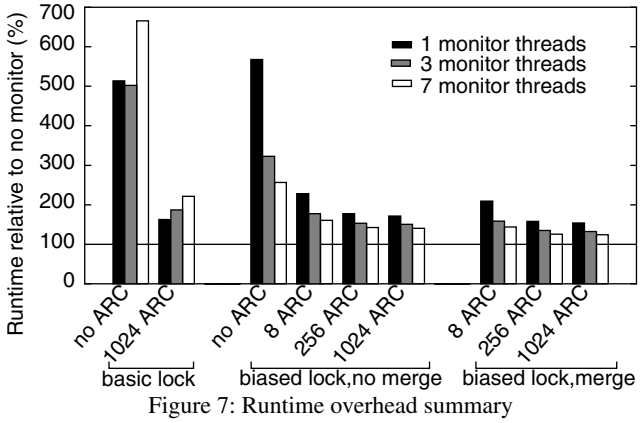

Figure 7: Runtime overhead summary

existent in [8]) to allow this overlap would need significantly more complex and dedicated hardware to service or buffer checks while checker cache misses are being serviced. [8]'s performance numbers cannot be compared to ours because [8] gives an overhead on the number of memory accesses using SPEC2000 *test* and *train* inputs but not runtime overhead with *ref* inputs.

VII. CONCLUSION

The key challenge with dynamic monitoring schemes for detecting pointer bugs is the runtime overhead. Previous approaches have used thread-level speculation (TLS) to reduce the overhead. However, the approaches still incur substantial slowdowns while requiring complex TLS hardware. We explicitly multithreaded the monitor code and use SMT to exploit the parallelism in the monitor code, avoiding TLS's complexity.

Out scheme still slows down the user thread due to two problems: instruction overhead and insufficient overlap among the monitor threads. To address instruction overhead, we exploited the natural locality in the user thread addresses and memoized recent checks in a small table called the allocation-record-cache (ARC). However, programs making and accessing many small memory allocations cause many ARC misses and reduce the effectiveness of ARC. To address this issue, we make the key observation that because adjacent memory objects result in ARC entries with contiguous address ranges, the entries can be merged into one by simply merging the ranges into one. This merging increases the effective size of the ARC. Finally, insufficient overlap among monitor threads occurs because of inefficient synchronization to protect the allocation data structure updated by the user thread and read by the monitor threads. We made the key observation that because monitor-thread reads occur for every check but user-thread writes occur only in allocations and deallocations, monitor reads are much more frequent than user writes. We proposed a locking strategy, called biased lock, which puts the locking overhead on the writer away from the readers.

We show that starting from a runtime overhead of 414% pesticide reduces this overhead to a respectable 24% running three monitor threads on an SMT using a 256-entry ARC with merging and biased lock. This 24% overhead compares well with previous schemes' 700% and 200% and also with the fact the Java which performs bounds-checks in-line in the user code incurs about 100% runtime overhead. Because pesticide checks all heap accesses, it covers all out-of-bounds heap accesses without any false positives.

REFERENCES

[1] Software errors cost us economy $59.5 billion annually. http://www.nist.gov/public_affairs/releases/n02-10.htm, 2002.

[2] Haithm Akkary and Michael Driscoll. A Dynamic Multithreading Processor. *Proceedings of 31st Int'l Symposium on Microarchitecture*, Dec 1998.

[3] Todd Austin, Scott Breach, and Gurindar Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation,* June 1994.

[4] Chris Bentley, Scott Watterson, David Lowenthal, and Barry Rountree. Implicit Java Array Bounds Checking on 64-bit Architecture. In *Proceedings of the 18th annual Int'l Conference on Supercomputing,* June 2004.

[5] Doug Burger, Todd Austin, and Steve Bennett. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin, 1996.

[6] P. Courtois, F.Heymans, and D. Parnas. Concurrent Control with readers and writers. In *Comminication of the ACM, Vol 14, No10, pp.667-668*, Oct 1971.

[7] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Summer Conference (USENIX 98)*, Jan. 1998.

[8] Josh Cates, Emmett Witchel and Krste Asanovic. "Mondrian Memory Protection". In *Proceedings of the 10th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[9] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation,* May 1996.

[10] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th Int'l Conference on Software Engineering*, May 2002.

[11] David Hovemeyer and William Pugh. Finding Bugs is Easy. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, Dec. 2004.

[12] Richard Jones and Paul Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the 3rd Int'l Workshop on Automated Debugging*, May 1997.

[13] Samuel Kendall. BCC: Run-time Checking for Cprograms. In *Proceedings of the USENIX Summer Conference (USENIX 83)*, Summer 1983.

[14] E. Christopher Marc Corliss and Amir Roth. "DISE: A Programmable Macro Engine for Customizing Applications". In *Proceedings of the 30th Int'l Symposium on Computer Architecture*, June 2003.

[15] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. Cmc: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[16] Jeffery Oplinger and Monica Lam. Enhancing Software Reliability with-Speculative Threads. In *Proceedings of the 10th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[17] D. Park, U. Stern, J. Skakkebask, and D. Dill. Java Model Checking. In *15th IEEE Int'l Conference on Automated Software Engineering*, Sept. 2000.

[18] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the Int'l Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 03)*, June 2003.

[19] Yan Solihin, Rithin Shetty, Mazen Kharbutli and Milos Prvulovic. HeapMon: a Low Overhead, Automatic, and Programmable Memory Bug Detector. In *Proceedings of the 1st Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC2 04)*, Oct. 2004.

[20] J. Gergory Steffan, Christopher Colohan, Antonia Zhai, and Todd Mowry. A Scalable Approach to Thread-level Speculation. In *Proceedings of the 27th Int'l Symposium on Computer Architecture*, June 2000.

[21] M. Sullivan and R. Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *Proceedings of 21st Int'l Symposium on Fault-Tolerant Computing, 1991*, June 1991.

[22] Dean Tullsen, Jack Lo, Susan Eggers, and Henry Levy. Supporting Fine-grained synchronization on a Simultaneous Multithreading Processor. In *5th Int'l Symposium on High Performance Computer Architecture*, 1999.

[23] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *Proceedings of the 22nd Int'l Symposium on Computer Architecture*, June 1995.

[24] Hongwei Xi and Songtao Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proceedings of the 1999 Conference of the IBM Center for Advanced Studies on Collaborative Research*, 1999.

[25] Pin Zhou, Wei Lin, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Jose Torrellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th Int'l Symposium on Microarchitecture*, Dec 2004.

[26] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Joseph Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Int'l Symposium on Computer Architecture*, June 2004.