Efficient Data Race Detection for C/C++ Programs Using Dynamic Granularity

Young Wn Song and Yann-Hang Lee
Computer Science and Engineering
Arizona State University
Tempe, AZ, 85281
ywsong@asu.edu, yhlee@asu.edu

***Abstract*—To detect races precisely without false alarms, vector clock based race detectors can be applied if the overhead in time and space can be contained. This is indeed the case for the applications developed in object-oriented programming language where objects can be used as detection units. On the other hand, embedded applications, often written in C/C++, necessitate the use of fine-grained detection approaches that lead to significant execution overhead. In this paper, we present a dynamic granularity algorithm for vector clock based data race detectors. The algorithm exploits the fact that neighboring memory locations tend to be accessed together and can share the same vector clock archiving dynamic granularity of detection. The proposed heuristic for sharing vector clock is simple but robust, can result in performance improvement in time and space, and is with minimal loss in detection precision. The algorithm is implemented on top of FastTrack and uses Intel PIN tool for dynamic binary instrumentation. Experimental results on benchmarks show that, on average, the race detection tool using the dynamic granularity algorithm is 43% faster than the FastTrack with byte granularity and is with 60% less memory usage. Comparison with existing industrial tools, Valgrind DRD and Intel Inspector XE, also suggests that the proposed dynamic granularity approach is very viable.**

Keywords-Race detection; Concurrent bug; Multithreaded programs

I. INTRODUCTION

Most embedded applications are constructed with multiple threads to handle concurrent events. Threads are synchronized for proper sharing of resource and data. Unfortunately, it is easy to misuse synchronization operations in multithreaded programming and threads may be vulnerable to race conditions and deadlocks. Ever increasing demands of multi-core processors aggravate this problem not only for embedded applications but for general desktop applications.

A data race occurs when a memory location is accessed concurrently by two different threads and at least one of the accesses is a write. Data races are hard to reproduce, find, and fix since a data race may only occur in a particular execution of the program and the race does not necessarily always cause observable errors in the program execution.

Over the past few years, several techniques have been developed to detect data races. Static analysis techniques [8, 20, 26] consider all execution paths for possible data races providing a better detection coverage than dynamic techniques but they suffer from excessive number of false alarms. On the other hands, dynamic techniques detect data races when execution paths are exercised and they largely fall into two categories: LockSet algorithms [23, 27] and happens-before algorithms [7, 19, 21, 22, 25]. In Eraser's LockSet algorithm [23], data races are reported when shared variable accesses violate a specified locking discipline, i.e., the variable is not protected by the same lock consistently. For a given execution path, checking a lock discipline enables Eraser to detect potential data races as well as ones that actually happened in the program execution. Eraser may report many false alarms which hinder developers' focus on fixing real problems. Eraser may also report false alarms by not being able to recognize synchronization idioms, e.g., a semaphore implementation using mutex locks.

Happens-before detectors are based on Lamport's happens-before relation [12] and don't report false alarms as the approach only checks any happens-before relation that actually occurs during the given execution paths. Based on the execution of a multithreaded program, a partial ordering of memory and synchronization operations can be defined. A pair of accesses to the same variable is concurrent when neither of the accesses happens before the other. The happens-before relation is realized by the use of vector clock [5] and there are largely two happens-before detection methods based on how shared accesses are represented and compared for the detection.

In the first method [21, 22, 25], a segment is defined as a code block between two successive synchronization operations and shared memory accesses are collected in a bitmap for each segment. In each thread a vector clock is collected to uncover any concurrent segments of running threads. If two concurrent segments contain shared memory accesses, the accesses are reported as data races. This method may incur a significant overhead in time despite of several optimization techniques such as *clock snooping* and *merging segments* [21, 22] as it requires set operations with the collected shared memory accesses.

In the second method [7, 19], other than a vector clock for each thread, each shared variable has two vector clocks for reads and writes which record access history of the variable by every thread. When a thread reads from a shared variable, the write vector clock of the shared variable is compared with the vector clock of the thread. A write-read data race is reported if the vector clock comparison reveals that the write and the read are not ordered by the happens-before relation. A similar protocol can be performed for a

write access. Having vector clocks for each shared variable can result in huge memory consumption. However, for applications developed in object-oriented languages, the memory requirement may be tolerable as large detection units such as fields or objects can be used.

Data race detection for embedded applications which are mostly written in C/C++ needs to consider fine granularity of data access (e.g., byte). It would seem to be a better choice to use the first happens-before detection method since having a vector clock for every shared read or write byte may make the detection infeasible. On the other hand, as shown in FastTrack [7], the second method can reduce the space and time overheads of vector clocks from $O(n)$ (where n is the number of threads) to nearly $O(1)$ with no loss in detection precision. While it may become feasible to detect races in C/C++ programs, the overheads using the FastTrack with fine granularity is still high for C/C++ programs, as illustrated in our experiment results.

In this paper, we present a dynamic granularity algorithm for vector clock based race detection. The detection granularity starts from byte and is dynamically adjusted as shared memory locations are accessed. A large detection granularity is adopted when neighboring bytes have the same vector clock. Thus, instead of multiple copies, a single copy of vector clock is shared among these neighboring bytes. Sharing the same vector clock among neighboring memory locations become feasible since (1) neighboring memory locations belonging to array or struct tend to be accessed together, (2) data structures are often accessed together during initialization even if they are separately protected afterward, and (3) some groups of shared memory locations are accessed only for one code block.

In the algorithm, a state machine is associated with a vector clock for read or write of a memory location and the state can be *Init*, *Shared*, *Private*, or *Race*. To minimize analysis overhead, the sharing decision for each read or write location is made at most twice for the lifetime of the location. Peak memory consumption is further reduced by temporarily sharing vector clock at the *Init* state. In addition, the possibility of false alarms caused by sharing vector clock is minimized as new sharing decision is made after the *Init* state.

We have developed a race detector based on FastTrack for C/C++ program and the dynamic granularity algorithm is implemented on top of the FastTrack implementation. Our experimental results on several benchmark programs show that the race detector using our dynamic granularity provides 43% speedup and 60% less memory over the FastTrack with byte granularity. Also we provide case studies on two popular data race detection tools: Valgrind DRD [16, 25] and Intel Inspector XE [11]. Our dynamic-granularity is about 2.2x and 1.4x faster than Valgrind DRD and Inspector XE, and consumes about 2.8x less memory than Inspector XE.

The rest of the paper is organized as follows. In the following section, a brief review of vector clock based race detectors is presented. Section III presents the proposed dynamic granularity algorithm. The implementation of data race detector using dynamic granularity are explained in Section IV. Section V shows experimental results on the tool with FastTrack as well as comparisons with Valgrind DRD and Intel Inspector XE. A concise survey of related works is described in Section VI and, in Section VII, we conclude the paper with future work.

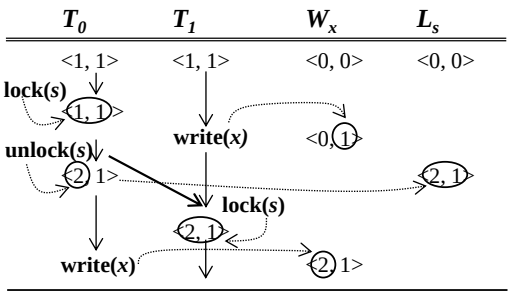

Figure 1. An example execution of DJIT+ is shown. T_0 and T_1 are vector clocks of thread *0* and thread *1*, respectively. W_x and L_s are vector clocks for write *x* and lock *s*, respectively. Solid arrows show happens-before relations and dotted arrows indicate vector clock updates by the operations.

II. Vector Clock Based Race Detectors

A. Preliminaries

The happens-before relation [12] over the set of memory and synchronization operations, denoted "→", is the smallest relation satisfying,

- **Program order:** If a and b are in the same thread and a occurs before b, then $a \rightarrow b$.
- **Locking**[1]**:** If a is a lock release and b is the immediately successive lock acquire on the same lock, then $a \rightarrow b$.
- **Transitivity:** If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two operations are concurrent if they are not ordered by the happens-before relation. A data race occurs when two memory operations are concurrent and at least one of them is a write.

A vector clock [5] is a vector of logical clocks for all threads and used to realize the happens-before relation. A vector clock is indexed with thread ids. For instance, in Figure 1, $T_1[0]$ gives the logical clock of thread *0* in the thread *1*'s vector clock and after the lock(*s*) $T_1[0] = 2$. The execution of a thread is logically divided into code blocks by synchronization operations and the logical clock for the thread is incremented in every new code block. The logical clocks of other threads in a thread's vector clock are updated through synchronization operations. Similarly, the access history of a shared memory location is recorded in vector clocks for the location. Then, the happens-before relation for the program execution is realized by the combination of the vector clocks of threads and shared memory locations.

B. DJIT+

DJIT+ [19] defines the execution of a thread as a sequence of *epochs*[2] and a new epoch starts from every lock

[1] Other synchronizations such as fork-join can be similarly defined. In this paper, we discuss synchronizations only with locking for clarity.

[2] It is defined as *timeframes* in the paper. But we use *epochs* for the consistency of discussion.

release operation. DJIT+ detects only the first race for each memory location. For consecutive reads of a memory location in the same epoch, it is sufficient to check only the first read for the detection of the first race. This property is also true for consecutive write operations.

A thread i has a vector clock T_i in which $T_i[i]$ (i.e., its' own clock) is incremented upon every new epoch. A vector clock L_s of a synchronization object s is updated when thread i performs a lock release operation on s and is set to the element-wise maximum of L_s and T_i. Upon a lock acquire operation of s by thread i, T_i is updated as the element-wise maximum of L_s and T_i. By the vector clock updates, happens-before relations for the program execution are constructed and a thread's logical clock is updated. As synchronization operations play a role in conveying a thread's clock to other threads, the thread's vector clock is known to other threads.

For a memory location x, the access history of read and write is represented with vector clocks R_x and W_x respectively. Upon the first read of x in an epoch at thread i,

1) A write-read data race is reported if there is another thread j whose write to x is not known to thread i, i.e.,$W_x[j] \geq T_i[j]$. If there was a synchronization from thread j to thread i between the write and the read, then the accesses should have been ordered and $W_x[j] < T_i[j]$.
2) Thread i updates R_x such that, $R_x[i] = T_i[i]$.

A similar protocol can be applied to write operations. Figure 1 shows an example of how DJIT+ detects a data race. In the example, the write on x in thread *0* is a data race since $W_x[1] \geq T_0[1]$.

By the property of checking only the first read and write in an epoch, run-time performance can be significantly improved. However, there still exists a huge overhead in time and space for maintaining the vector clocks of shared memory locations.

C. FastTrack

FastTrack [7] is based on DJIT+ and provides a significant performance enhancement over DJIT+ with the same detection precision as DJIT+. FastTrack exploits the insight that, in most cases the last access of a memory location can provide enough information for detecting data races instead of using the full vector clock representation. An epoch representation denotes the last access of a memory location. If the last access was by a thread t at logical clock c, then the epoch is denoted $c@t$ using only two scalars. For all writes to a memory location, the epoch representation can be used instead of the full vector clock because all writes to the location are totally ordered by the happens-before relation before the first race on the location. This leads to a reduction in time and space overhead from $O(n)$ (where n is the number of threads in the execution) to $O(1)$. For read operations, the epoch representation cannot be used all the time since read operations can be performed without locking (i.e., read shared). Thus, read vector clock, R_x, is replaced with an adaptive representation which uses a full vector clock only when the read is shared with other threads without protection. Based on the adaptive representation, the overhead for reads can be reduced from $O(n)$ to nearly $O(1)$.

III. Dynamic Granularity Algorithm

FastTrack is a fast and space-efficient race-detection tool but it still needs to keep vector clocks for each memory location. This is not problematic for object-oriented programming languages since detection unit can be either a field or an object. For C/C++ programs, it is not easy to detect data structure boundaries (e.g., dynamically allocated struct or array) and moreover data are often protected in fine grained (e.g., a byte or a word). A simple way to rectify the problem is to use a fixed granularity. However using a granularity larger than a word would produce a large number of false alarms and does not help reducing the overheads for many cases as shown in our evaluation results.

In this section, we present a dynamic granularity algorithm which enables vector clock based race detectors to use detection granularity as large as possible with minimal false alarms. The algorithm is described on the assumption of using DJIT+ or FastTrack detectors; a thread's execution is defined as a sequence of epochs; and for consecutive accesses of a memory location in an epoch, only the first read and write are checked. In the description, *two vector clocks are the same* when they are the same size and their contents are of equal value, and both a vector clock and an epoch representation are referred to as *a vector clock*.

The dynamic granularity is realized by sharing vector clock with neighboring memory locations. The sharing heuristic is based on the following observations:

1. Neighboring memory locations (e.g., array, struct) tend to be accessed together whether the locations have data races or not. Hence, they can have the same vector clock.
2. At initialization, a data structure is often accessed in its entirety, e.g., zero-out an array, even if its elements are protected separately afterward.
3. There are groups of memory locations that are accessed only in one epoch for the entire lifetime of the location, e.g., dynamically allocated memory locations that are used temporarily.

With these observations, the dynamic granularity algorithm is realized with a vector clock state machine that is described in the following subsection.

A. Vector Clock State Machine

For a memory location, we maintain a read location and a write location separately. Hence, only the same access type (read or write) of vector clocks can be shared. Let L be a location which can be either a read or write location. When L is accessed for the first time, a vector clock is created for it. The sharing state of each location is maintained by a state machine attached to its vector clock as shown in Figure 2. The state machine basically has four states. In the first epoch access, the vector clock is temporarily shared with its neighbor if the neighbor has the same vector clock. When the second epoch access begins at a location, the shared vector clock at the location is split and new sharing decision is made.

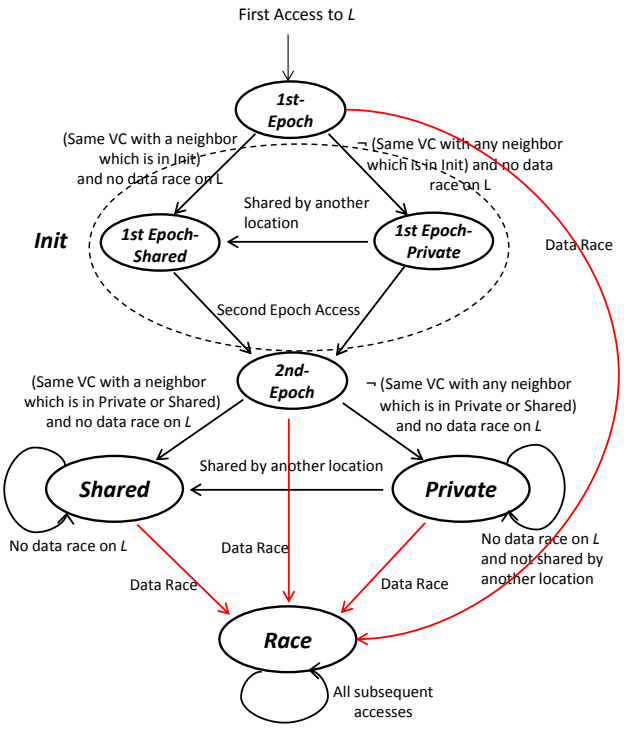

Figure 2. Vector clock state machine for each read or write location.

A *neighbor* of *L* is a memory location adjacent to *L* that is considered to share potentially a vector clock with *L*. A location *L* can have two neighbors that one is located left (a predecessor of *L*) and the other is located right (a successor of *L*). During the first epoch, the neighbors are the nearest predecessor and successor that have valid vector clocks. A new access to a location *L* initiates a vector clock in the *Init* state. This vector clock can be shared with *L*'s neighbors if they have the same clock value and are in the *Init* state as well. For an access to a location *L* in the 2nd epoch, the neighbors are at locations *L-size* and *L+size* where *size* is the data size of the access. As long as the neighbors are not in the *Init* or *Race* state, we compare the vector clock of *L* with those of its neighbors. If the vector clocks are equal, the state is set to *Shared*. Otherwise, it is *Private*.

Each vector clock can be in one of the following 4 states:

Init: When *L* is accessed first time, its vector clock is initiated and is set to this state until the next epoch access. This *Init* state is intended to approximate the initialization process. Note that elements of a data structure may be initialized together and have the same vector clock during the first epoch. However, starting from the 2nd epoch, the elements may be accessed separately and have their own private vector clocks.

While in the *Init* state**,** *L* can be in the *1st-Epoch-Shared* sub-state if one of the neighbors has the same vector clock and are in the *Init* state. Thus, *L* shares temporarily its vector clock with its neighbors during the first epoch. When there is no neighbor that has the same vector clock as the location *L*, the state of *L* becomes *1st-Epoch-Private*. The state of *L* can transition to *1st-Epoch-Shared* when a new neighbor location *L′* is initiated and *L′* has the same vector clock as *L*. The rationale behind the temporary sharing is that there could be many memory locations that are accessed only in one epoch. Examples include dynamically allocated memory or groups of memory locations in a big data structure that are used only in one epoch. As our experimental results show, having this *Init* state saves a considerable amount of memory for some applications. Upon the next epoch access, *L* has its own vector clock and state, and the new sharing decision is made for the rest lifetime of the location *L*.

Shared: On the second epoch access of *L*, if there is no data race on *L* (and no read-read conflict for a read location) and there exists a neighbor that has the same vector clock as *L* and is in either *Shared* or *Private*, the location *L* shares its vector clock with the neighbor. Also, the state of *L* can transition from *Private* to *Shared* when *L* becomes a neighbor of another location *L′* that has the same vector clock as *L*.

Private: When there is no neighbor that has the same vector clock as *L*, the state of *L* becomes *Private* on the second epoch access.

Race: On a data race, the state of *L* becomes *Race*. If there are memory locations sharing the same vector clock with *L*, the sharing is terminated and each of these locations become *Race* and is assigned with a private vector clock.

B. *Dynamic Granularity*

The dynamic granularity is achieved by sharing vector clocks with neighboring address locations. The detection starts with byte granularity for every location and the granularity is increased as more neighboring locations share the same vector clock. The vector comparison to determine vector clock sharing can be an expensive operation. However, following the vector clock state machine, there can be at most two sharing decisions for the lifetime of a memory location and it requires only *O*(1) time overhead in most cases when the FastTrack algorithm is used. In fact, we can have a significant performance enhancement by the use of dynamic granularity since, as we change to a large granularity, multiple accesses may be treated as the same epoch accesses.

IV. IMPLEMENTATION

We have implemented the FastTrack algorithm for data race detection of C/C++ programs with fixed (byte and word) and dynamic granularities. Intel PIN tool 2.11 [13] is used for dynamic instrumentation of the programs.

A. *Instrumentation*

To trace all shared memory accesses, every data access operation is instrumented. If an instruction accesses non-shared memory (e.g., stack), the instrumentation routine returns immediately. Figure 3 shows pseudocode for memory read instructions. Memory write can be similarly described and we omit the FastTrack algorithm for clarity.

When an access is not the first read or write in an epoch, vector clock updates and data race checking on that access can be skipped according to the FastTrack algorithm.

```
void memoryRead(uint addr, uint size, uint tid)
{
   if (nonSharedRead(addr) || sameEpoch(tid, addr))
      return;

   Location L = findReadAccess(addr);
   if (!L) {                                  // The first access of addr
      L = insertRead(addr, size);
      shareFirstEpoch(L, addr, size);       // Temporary sharing
      L→ state = Init;
   } else if (L→state==Init) {               // Second epoch access
      split(L, addr, size);                  // Split for new sharing
      shareSecondEpoch(L, addr, size);   // New sharing decision
      if (L→count>1)
         L→state = Shared;
      else
         L→state = Private;
   }

   //if race found on addr, split all vectors sharing with L
   //  and set states of locations to Race
   if (raceFound(addr))
      splitAndSetRace(L, addr);

   //remember access L into bitmap for this thread
   //the bitmap is reset at the next epoch of this thread
   insertEpochAccess(tid, addr);
}
```

Figure 3. Instrumentation code for memory read.

Checking the same epoch access can be costly since looking up a vector clock from a global data structure requires synchronization of threads. To reduce overhead, a per-thread bitmap is implemented. When the first access is made in an epoch, the access is set in the bitmap and the bitmap is reset for every lock release operation. Because the bitmap is a thread local data structure, checking the same epoch is more efficient than looking up a global data structure.

The mechanism for dynamic granularity is invoked at first two epochs for each read or write location. Thus, the overhead can be negligible. Also it will be straightforward to apply dynamic granularity into existing data race detection tools.

B. Indexing Structure

To find the vector clock of each read or write location, a chained hash table is implemented as shown in Figure 4. For efficient sequential processing such as deleting vector clock entries from free() and vector clock sharing process, each hash chain entry has an indexing array which can contain up to *m* pointers for vector clock entries. For a 32-bit address, the upper address (upper 32-$\log_2 m$ bits of the address) is hashed into the table to locate the corresponding hash entry. The vector clock entry for the address is indexed using the lower address (lower $\log_2 m$ bits of the address).

The size of the indexing array in the hash entry changes according to memory access patterns, thus saving memory on the indexing array. When a new hash entry is created, it starts with an array of *m*/4 pointers since the most common access pattern is word access. When a byte access (i.e., the address is not word or half-word aligned) is detected, the array is expended to have *m* pointers.

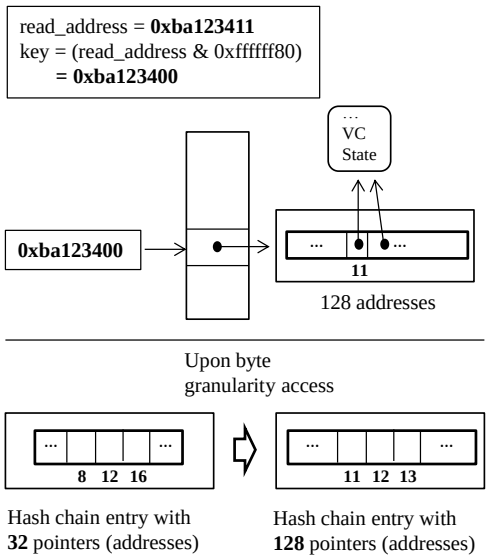

Figure 4. *Top*: A separate chaining hash table implementation. Each entry can contain *m* addresses (shown a case *m*=128). *Bottom*: The size of indexing array in a hash entry is changed from *m*/4 to *m* when byte granularity access begins in the entry.

V. EVALUATION

In this section, we present the effectiveness of our dynamic granularity algorithm. First, we show our experimental results of the FastTrack with fixed (byte and word) and dynamic granularities. Second, analysis results on the state machine are presented. Lastly, performance measures of two popular data race detection tools, Valgrind DRD [16, 25] and Intel Inspector XE [11], are compared with the FastTrack using dynamic granularity. All experiments were performed on Ubuntu 12.04 with kernel version 3.2.0 and Intel Core Duo CPU with 4 GB of RAM.

All experiments were run with 11 benchmarks: 8 from the PARSEC-2.1 benchmark suite [1] that are implemented with the POSIX thread library and 3 from popular multithreaded applications: *FFmpeg* [4], a multimedia encoder/decoder; *pbzip2* [9], a parallel version of *bzip2*; and *hmmsearch* [6], a sequence search tool in bioinformatics. For input sets of the PARSEC benchmark programs, the *simsmall* input set is used for *raytrace* while the *simlarge* is used for the rest 7 programs. Input for the other three programs is chosen to have similar run times as the PARSEC benchmark programs.

A. Performance and Detection Precision

Table 1 shows overall experimental results of the FastTrack with the three different granularities. "Total shared accesses" column shows the total number of shared reads and writes during each benchmark program run. "Max. # of vectors in byte granularity" column indicates the maximum number of vector clocks present for the execution of each program in byte granularity run. These two columns, combined with the number of threads, give us a general idea of the instrumentation overhead in the detection. "Slowdown" and "Memory overhead" columns report runtime and memory overheads of each detection mechanism

Table 1. Overall experimental results.

	Benchmark Program	Total shared accesses (million)	Max. # of vectors in byte granularity	# of threads	Base time (sec)	Base memory(MB)	Slowdown			Memory Overhead			# of Detected Data Races		
							Byte granularity	Word granularity	Dynamic granularity	Byte granularity	Word granularity	Dynamic granularity	Byte granularity	Word granularity	Dynamic granularity
PARSEC	facesim	8033	93,930,447	2	6.1	288	138	138	102	8.8	8.8	4.6	8909	8909	8909
	ferret	3856	83,678,104	11	6.7	146	65	57	52	16.6	11.7	8.9	2	2	2
	fluidanimate	2443	11,220,394	3	2.0	248	87	87	81	2.6	2.6	2.2	1	1	1
	raytrace	18	3,291,927	3	9.5	170	27	27	27	2.1	2.1	2.0	13	13	13
	x264	3392	12,550,683	256	2.2	49	75	55	64	20.8	9.6	9.0	1300	993	1313
	canneal	359	7,141,372	3	6.5	104	13	13	13	5.3	5.3	5.1	0	0	0
	Dedup	10003	9,208,539	7	7.7	2682	152	76	85	1.0	1.0	1.0	0	0	0
	streamcluster	8030	2,277,958	5	3.8	30	245	245	137	4.8	4.8	3.7	1053	1053	1079
	ffmpeg	5790	7,195,586	9	3.0	95	121	106	109	4.0	3.0	3.1	1	9	1
	pbzip2	7239	8,842,583	6	5.7	67	64	49	39	5.4	4.2	3.4	0	0	0
	hmmsearch	38050	961,831	3	26.6	23	84	83	45	4.9	4.9	4.3	1	1	1
Average							97	85	68	6.9	5.3	4.3			

as the ratios to the run time and maximum memory used in the un-instrumented program execution. "# of Detected Data Races" columns show the number of data races detected by each granularity detector.

Overall Results. The results show that the dynamic granularity detector is on average 1.43x and 1.25x faster than the byte-granularity and the word-granularity detectors, respectively. For memory overhead, on average the dynamic granularity detector consumes 60% less memory than the byte granularity detector and 23% less memory than the word granularity detector.

For benchmarks *facesim*, *fluidanimate*, *raytrace*, *canneal*, *streamcluster*, and *hmmsearch*, memory consumption is neither reduced nor does detection become faster when we switch from byte granularity to word granularity. Since the sizes of most accesses in those benchmarks are equal to or greater than a word, no vector clock is created for non-word-aligned locations. Thus, using word granularity does not help to reduce the overhead of vector clock operations. However, except for *raytrace*, *canneal*, the use of dynamic granularity enhances the detection both in time and memory space. This suggests the advantage of using a large granularity crossing word boundaries. The results from *ferret* and *pbzip2* show improvements both in word granularity and dynamic granularity, but the use of dynamic granularity has more benefits than the use of word granularity. It may be strange to see that the factor of memory overhead for *dedup* is 1.0 for all detectors. Note that the maximum overhead does not always occur when the maximum memory is used in the benchmark. In fact, *dedup* uses a large amount of memory (about 2.7 GB) at the beginning of the execution when the detection overhead is close to zero. Then, the memory usage is gradually decreased while the peak memory overhead from the detectors occurs afterward.

For detection precision, there are few discrepancies among the detectors as shown in Table 1. With word granularity, 993 data races are reported for *x264* while the dynamic granularity detector reported more data races. When word granularity is used, non-word-aligned addresses are masked to word boundary and data races for those locations are detected as one race. That is how the word granularity detector reported less number of data races for *x264*. We carefully inspected data races from *x264* found by the dynamic granularity detector and noticed that there were 4 write locations which were sharing a vector clock with one location having a data race. More data races from *ffmpeg* by the word granularity detector and from *streamcluster* by the dynamic granularity detector are found to be false alarms due to inaccurate updates of vector clocks when large detection granularities are used.

Memory Overhead. Table 2 shows the details of memory overhead. For each granularity, three major overhead factors are shown. The "Hash" column indicates the maximum memory used for hash tables and hash entries to index vector clocks. The "Vector clock" column gives the maximum memory used to store vector clocks. The third column, "Bitmap", is the maximum memory used for bitmap data structures for checking same epoch accesses. The overhead is measured based on object size and is slightly underestimated since the size of memory allocated for a data object is usually little more than the actual size of the type.

The dynamic granularity algorithm saves a substantial amount of memory used for vector clock allocations (as shown in "Vector clock" columns). Another view of memory savings on vector clocks is shown in Table 3 which lists the maximum numbers of vector clocks during program executions. On average, the dynamic granularity detector uses roughly 4x and 3x less memory for vector clocks than the byte granularity and the word granularity detectors, respectively. Indexing costs of the byte granularity and the dynamic granularity detectors are almost same since the use of dynamic granularity does not save memory on indexing vector clocks (as shown in "Hash" columns). The use of word granularity saves memory on indexing for some benchmark programs because the addresses are mostly word-aligned, thus using smaller indexing arrays in hash entries.

Table 2. Memory overhead of FastTrack detection with different granularities

Benchmark program	Base Memory (MB)	Byte Granularity				Word Granularity				Dynamic Granularity			
		Hash(MB)	Vector Clock (MB)	Bitmap (MB)	Overhead total (MB)	Hash(MB)	Vector Clock (MB)	Bitmap (MB)	Overhead total (MB)	Hash(MB)	Vector Clock (MB)	Bitmap (MB)	Overhead total (MB)
facesim	288	513	1505	132	2149	514	1503	132	2148	517	273	131	921
ferret	146	458	1573	66	2097	259	1072	57	1388	454	469	64	988
fluidanimate	248	132	180	27	339	132	180	27	338	132	74	27	233
raytrace	170	35	53	15	103	30	53	15	97	35	22	15	72
x264	49	77	233	7	317	33	89	7	129	77	44	7	128
canneal	104	87	176	52	315	87	176	52	315	87	155	52	294
dedup	2682	212	148	57	417	147	100	58	305	214	88	56	358
streamcluster	30	11	37	6	54	11	36	6	54	11	3	6	21
ffmpeg	95	37	118	7	162	18	43	7	68	37	28	7	72
pbzip2	67	49	141	17	207	37	89	17	143	50	4	16	70
hmmsearch	23	12	15	3	30	12	15	3	30	13	1	3	17
Average		148	380	35	563	116	305	35	456	148	105	35	288

Table 3. Maximum number of vector clocks present

Benchmark program	Max. # of vector clocks			Avg. sharing count
	Byte Granularity (thousand)	Word Granularity (thousand)	Dynamic Granularity (thousand)	
facesim	93,930	93,808	16,991	5.5
ferret	83,678	52,375	24,689	3.4
fluidanimate	11,220	11,174	4,590	2.4
raytrace	3,291	3,285	1,319	2.5
x264	12,550	4,803	2,019	6.2
canneal	7,141	7,141	5,812	1.2
dedup	9,208	6,226	5,474	1.7
streamcluster	2,277	2,245	193	11.8
ffmpeg	7,195	2,620	1,696	4.2
pbzip2	8,842	5,570	265	33.3
hmmsearch	961	950	53	17.9

Slowdown. Speedups can be achieved in two ways by the use of a large granularity. Firstly, in DJIT+ based race detectors including FastTrack, the vector clock operations are performed only for the first read and write of a shared memory location during each epoch. The use of a large granularity makes multiple accesses as one access. Thus, there are more same epoch accesses enhancing the detection performance. In Table 4, we show the percentage of same epoch accesses along with the slowdown for each benchmark program. The results suggest that in most cases the performance gains from a large granularity are consistent with the percentage of same epoch accesses. For the cases of *canneal* and *raytrace*, as the percentages of same epoch accesses do not vary noticeably among different granularities, there is no performance enhancement by the use of a large granularity.

Second, speedup comes from the reduction of vector clock allocation and de-allocation operations. For the case of *pbzip2*, the dynamic granularity detector is 1.6x faster than the byte granularity detector while the percentages of same epoch accesses are same. On the other hand, the average number of locations that share a vector clocks is 33.3 under dynamic granularity as shown in Table 3. This implies that there will be about 33 times less vector clock creation and deletion operations. The other interesting case is *dedup*. The program has the same percentage of same epoch accesses for both byte and dynamic granularities and the average number of sharing vector clocks is only 1.7. However, the dynamic granularity detector is 1.78x faster than the byte granularity detector. The reason is that there are an excessive number of dynamic memory locations in *dedup*. On average, there is about 1.7 GB of memory allocated and de-allocated in the 11 benchmark programs whereas it is 14GB in *dedup*.

B. Analysis of State Machine

The sharing decision for realizing dynamic granularity is made twice for the lifetime of a location *L* (read or write). In the first epoch, *L* tries to share a vector clock with its neighbors temporarily. In the second epoch, a new sharing decision is made for the rest lifetime of *L*. We make a firm sharing decision at the second epoch (after initialization of *L*) since some groups of data structures can be initialized at the same segment of code even if they are accessed separately afterward. This design makes the sharing decision accurate. The temporary sharing at the first epoch may save a considerable amount of memory because there could be groups of locations that are accessed together only once in the same epoch and if that is the case, we do not have to keep a separate vector clock for each of them. Notice that there is no possibility of false alarms by the temporary sharing at the *Init* state. Table 5 shows the effectiveness of this design. Column 2 and 3 show the maximum memory used without and with temporarily sharing at the first epoch. Column 4

Table 4. Measures of same epoch accesses

Benchmark Program	Slowdown			Same epoch		
	Byte Granularity	Word Granularity	Dynamic Granularity	Byte Granularity	Word Granularity	Dynamic Granularity
facesim	138	138	102	74%	74%	94%
ferret	65	57	52	78%	83%	87%
fluidanimate	87	87	81	89%	89%	94%
raytrace	27	27	27	65%	65%	68%
x264	75	55	64	67%	90%	78%
canneal	13	13	13	97%	97%	97%
dedup	152	76	85	85%	93%	85%
streamcluster	245	245	137	50%	51%	97%
ffmpeg	121	106	109	68%	90%	84%
pbzip2	64	49	39	95%	97%	95%
hmmsearch	84	83	45	83%	83%	98%
Average	97	85	68	77%	83%	89%

Table 5. Comparisons of state machines with different configurations

Benchmark program	Max. Memory(MB)		# of Detected Data Races	
	No sharing at Init	Sharing at Init	No Init state	With Init state
facesim	2180	1317	9210	8909
ferret	1808	1302	2	2
fluidanimate	604	551	18529	1
raytrace	348	334	13	13
x264	470	442	1315	1313
canneal	550	530	0	0
dedup	2729	2730	0	0
streamcluster	142	111	1079	1079
ffmpeg	301	294	1	1
pbzip2	359	225	2	0
hmmsearch	107	99	1	1
Average	873	721		

shows the number of detected data races without the *Init* state, i.e., no temporary sharing and the sharing decision is made only once in the first epoch. Comparing with column 5 in which the *Init* state is added, there could be many false alarms as the consequence of improper sharing decisions made only in the first epoch. In addition, the results suggest that there are considerable numbers of memory locations that are used only in one epoch.

C. Case Studies

In this section, we present experimental results on two popular data race detection tools, DRD in Valgrind-3.8.1 [25] and Intel Inspector XE 2013 [11] update5. Also comparison results with our dynamic granularity on FastTrack are given. DRD, a tool for programs written with the POSIX library, detects various errors including data races, lock contention delays, and misuses of the POSIX library. The race detection algorithm in DRD is based on RecPlay [21]. Since DRD does not keep vector clocks for each memory location in its segment comparison approach, we expect that DRD uses less memory but is slower than FastTrack. Intel Inspector XE is a memory and thread error checker that is capable of detecting various errors including data races, deadlocks, and cross-thread stack access.

Inspector XE provides a GUI with comprehensive analysis reports, including the source code location of an error, calling stack analyses, and suggestion for fixing any detected errors. Likewise, DRD provides execution context for each error as well as the location that the error occurs. Race report from our implementation of FastTrack is not as comprehensive as the two tools, but we provide the location of a race along with the previous access location, thread ids, and the race memory address. The information should be sufficient for developers to fix the problems easily.

For Inspector XE, the command-line version was used and only data race detection is enabled. The two tools used byte granularity and all tools, including our dynamic granularity version of FastTrack, traced all modules including shared libraries. For the dynamic granularity detector, we applied the similar suppression rules as in DRD, e.g., suppressed data races detected from libc and ld. The dynamic granularity detector and DRD report the first race for each memory location while Inspector XE uses a combination of instruction pointers and timeline when a race occurs to distinguish races. Thus Inspector XE may report the same accesses on a specific memory location as multiple races or multiple accesses issued at the same instruction points as one race. DRD and Inspector XE classify the detected data races with execution context, but in the experimental results we list the raw number of data races before the classifications.

The comparison results are shown in Table 6. Both Inspector XE and DRD exited on *dedup* runs with out of memory warnings. DRD on *fluidanimate* and Inspector XE on *ffmpeg* ran for more than 24 hours before we stopped the analyses.

As we expected before the experiment, DRD is slower than the FastTrack with dynamic granularity and even slower than the FastTrack with byte granularity, but DRD consumes less memory than the FastTrack with dynamic granularity. The comparison results also suggest that the dynamic granularity detector is as accurate as the other two tools. All three tools detected the same race for *hmmsearch* (Inspector XE reported the same race one more time in a different timeline). The dynamic granularity detector and Inspector XE reported the same races for three benchmarks *ferret*, *fluidanimate*, and *streamcluster*, For *raytrace*, the dynamic granularity detector and DRD reported the same races, but DRD reported more races from pthread library which was suppressed by the dynamic granularity detector. DRD detected no race for *ffmpeg* while the dynamic granularity detector reported one race. We manually inspected the source code and found that it was a data race by the two worker threads accessing a shared variable without protection.

Table 6. Performance comparison of Valgraind DRD, Intel Inspector XE and FastTrack with dynamic granularity

Benchmark program	Base time (sec)	Base Memory (MB)	Slowdown			Memory Overhead			# of Detected Data Races		
			Valgrind DRD	Intel Inspector XE	Dynamic granularity	Valgrind DRD	Intel Inspector XE	Dynamic granularity	Valgrind DRD	Intel Inspector XE	Dynamic granularity
facesim	6.1	288	59	128	102	2.2	6.0	4.6	8909	31	8909
ferret	6.7	146	748	87	52	2.6	5.0	8.9	108	4	2
fluidanimate	2.0	248	--	89	81	--	12.4	2.2	--	7	1
raytrace	9.5	170	42	17	27	1.9	4.1	2.0	16	0	13
x264	2.2	49	143	246	64	3.2	22.1	9.0	988	218	1313
canneal	6.5	104	31	41	13	8.2	11.9	5.1	0	0	0
dedup	7.7	2682	--	--	85	--	--	1.0	--	--	0
streamcluster	3.8	30	66	108	137	4.2	17.5	3.7	1067	61	1079
ffmpeg	3.0	95	120	--	109	2.6	--	3.1	0	--	1
pbzip2	5.7	67	64	99	39	2.9	8.6	3.4	0	0	0
hmmsearch	26.6	23	74	64	45	4.4	21.9	4.3	1	2	1
Average			150	98	68	3.6	12.2	4.3			

VI. RELATED WORK

Aside from the 3 basic approaches for race detections, [7, 19, 21, 22, 23, 25, 27], a variety of hybrid race detectors have been proposed in which Eraser's Lockset algorithm is combined with the happens-before algorithm to have better detection coverage and to avoid false alarms. O'Callahan and Choi [17] have proposed a race detection algorithm in which a subset of happens-before relations is added to a Lockset based detector. The detector is optimized by detecting redundant event accesses and by the use of a "two phase" approach of *detailed* and *simple* mode detections. MultiRace [19] combines DJIT+ and Lockset algorithm and only check the first access in each time frame. In MultiRace, the number of detection operations is reduced based on the information produced from LockSet and false alarms from LockSet are filtered out by happens-before relations. ThreadSanitizer [24] is a hybrid race detector for C++ programs that offers tunable options to users. Its dynamic annotations allow the detector to be aware of user defined synchronizations. Thus the tool hides certain false alarms and benign races. RaceTrack [27] incorporates the happens-before relation into the LockSet algorithm and only report races caused by concurrent accesses. One interesting idea in RaceTrack is the use of adaptive granularity. The detection granularity starts from object level and becomes field level when a potential race is detected. Unfortunately, the idea, based on object references, is not applicable to C/C++ programs.

LiteRace [14] is a sampling based race detector grounded in the *cold-region hypothesis* that infrequently accessed areas are more likely to have data races than frequently accessed areas. Accesses to code regions of different function units are sampled while all synchronization operations are collected. The sampler starts at a 100% sampling rate and the sampling rate is adaptively decreased until it reaches a lower bound. PACER [2] is another sampling based race detector that periodically samples all threads and offers a detection rate proportional to the sampling rate. These approaches offer reasonable detection rate with minimal overhead, but may miss critical data races.

As an alternative to software only race detectors, several hardware assisted race detectors have been proposed. In SigRace [15], data addresses are automatically encoded in hardware address signatures and in a hardware module. The signatures are intersected with those of other processors to detect data races. Greatehouse et al. [10] proposed a demand-driven race detector that utilizes cache performance counters to detect data sharing between threads. When the data sharing is detected, a software race detector is enabled and run until there is no more data sharing. These approaches are efficient but require specific hardware making them impractical.

While many researchers have focused on data race detection algorithms for Java programs, only a few of which have presented evaluation results for existing data race detection tools on C/C++ programs. Aikido [18] is a framework for shared data analysis in which sharing data is detected using per-thread page protection technique. The Aikido sharing detector is complementary to dynamic granularity and is effective to remove the instrumentation overhead of no-shared memory accesses. IFRit [3] is a dynamic race detection algorithm for C/C++ programs based on *interference-free region* which can limit the range of code instrumentation. IFRit has been compared with FastTrack and ThreadSanitizer [24]. Both researches applied the PARSEC benchmark suite in their performance evaluations, but only the *simsmall* input set was used and no memory overhead was reported. Moreover, none of them made an attempt to compare their tools with commercial grade data race detection tools.

VII. Conclusions

In this paper, we have presented a dynamic granularity algorithm for C/C++ programs that enables vector clock based race detectors to adjust detection granularity. A vector clock state machine is employed to determine when vector clocks can be shared. The state machine also considers the initialization patterns of data structures. Thus, possible false alarms due to vector clock sharing can be reduced.

Our experimental results show that the dynamic granularity detector outperforms the FastTrack with both byte or word granularity and also outperforms two existing data race detection tools, Valgrind DRD and Intel Inspector XE.

We plan to extend our work by optimizing the sharing algorithm. The current work maintains read and write vector clocks separately. The decision of sharing read vector clocks can be guided by the status of write vector clocks. We also plan to enhance the vector clock state machine to accommodate access behavior after the second epoch so that the detection granularity can be changed more dynamically.

Acknowledgment

This work was supported in part by the NSF I/UCRC Center for Embedded Systems, and from NSF grant #0856090.

References

[1] C. Bienia. Benchmarking Modern Multiprocessors. *Ph.D. Thesis. Princeton University*, 2011.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 255-268, 2010.

[3] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 467-484, 2012.

[4] FFmpeg. http://www.ffmpeg.org/.

[5] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28-33, 1991.

[6] R. D. Finn, J. Clements, and S. R. Eddy. HMMER web server: Interactive sequence similarity searching. *Nucleic Acids Research Web Server* Issue 39:W29-W37, 2011.

[7] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 121-133, 2009.

[8] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 219-232, 2000.

[9] J. Gilchrist. Parallel BZIP2, http://compression.ca/pbzip2/.

[10] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the 38th annual international symposium on Computer architecture (ISCA)*, pages 165-176, 2011.

[11] Intel Inspector XE 2013. http://software.intel.com/en-us/intel-inspector-xe.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.

[13] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G, Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 190-200, 2005.

[14] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 134-143, 2009.

[15] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, pages 337-348, 2009.

[16] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 89-100, 2007.

[17] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 167-178, 2003.

[18] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: Accelerating shared data dynamic analyses. In *Proceedings of the seventeenth international conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 173-184, 2012.

[19] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ninth ACM SIGPLAN symp. on Principles and practice of parallel programming (PPoPP)*, pages 179-190, 2003.

[20] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):1-55, 2011.

[21] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133-152, 1999.

[22] M. Ronsse, M. Christiaens, and K. D. Bosschere. Debugging shared memory parallel programs using record/replay. *Future Generation Computer Systems*, 19(5):679-687, 2003.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391-411, 1997.

[24] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, pages 62-71, 2009.

[25] DRD, Valgrind-3.8.1. http://valgrind.org/.

[26] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC-FSE)*, pages 205-214, 2007.

[27] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP)*, pages 221-234, 2005.