A Case Study on Context Maintenance in Dynamic Hybrid Race Detectors †

Jialin Yang
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
jialiyang4-c@my.cityu.edu.hk

Ernest Bota Pobee
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
ebpobee2-c@my.cityu.edu.hk

W.K. Chan‡
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cityu.edu.hk

***Abstract*—Many dynamic hybrid race detectors aim at detecting violations of the lockset discipline in execution traces of multithreaded programs. They are designed to abstract memory accesses appearing in traces as contexts. Nonetheless, they keep these contexts in different extents and partition the sets of contexts into equivalent classes of different granularity. In our case study, we compare three detectors using the PARSEC benchmark suite to examine the impact of using unrestricted strategy or restricted strategy for keeping these contexts in sequence on detection effectiveness, and the impact of partitioning context sequences by different granularities on scalability in time cost. The case study results indicate that using restricted context sequences sufficed to detect very high proportions of locking discipline violations detectable by using unrestricted context sequences, and the partitioning of context sets into finer equivalent classes significantly lowers the scalability in time cost with increasing number of threads to handle the same input workload.**

***Keyword*—Multithreaded program, data race, concurrency bug detection, dynamic analysis, case study**

I. INTRODUCTION

A data race is a concurrency bug that may appear in an execution trace of a multithreaded program. It occurs when two threads in the same trace access the same shared memory location without any protection by synchronization operations, and at least one of these accesses is a write [6]. Its occurrence may result in program state corruption or produce a crashing run. Previous case studies on open-source multithreaded programs show that the presence of a data race often indicates the presence of other higher-level concurrency bugs, such as atomicity violation or linearizability problem, in the same program [2][13].

Dynamic race detection techniques track events (i.e., memory accesses and synchronization operations) performed by various threads in execution traces. Most of them analyze these events and their ordering on-the-fly. These on-the-fly race detectors can be further classified into *happens-before* (HB) based detectors [6][9], *locking discipline* (LD) based detectors [17], and *hybrid* detectors [10][15][22][23][24].

The HB relation [11] is a causal order of events in a trace, and HB based detectors deem any two accesses to the same shared memory location (and at least one of them being a write) failing to be separated by such a relation over synchronization operations as a data race. The *locking discipline* (*LD*) [17] is a heuristic rule that every access to the same shared memory location should be protected by some common lock (i.e., mutex), and detectors based on LD use the notion of *lockset* to check if the set of common locks being held by different threads along a trace when accessing the same shared memory location becomes empty [17].

Hybrid detectors [10][15][22][23][24] combine both HB based and LD based techniques. Different emphases and strategies result in techniques with different scope of memory accesses to check for race occurrences, memory cost, time efficiency, and scalability with increasing number of threads. To support the above checking, these techniques commonly abstract each past access event as a *context*, which is a tuple of information extracted from the event. The warning reported by such detectors is referred to as ∅-race.

In this paper, we present and compare three hybrid detectors on the PARSEC benchmark suite [1] to examine (1) the impact of using unrestricted strategy or restricted strategy for keeping context sequences on detection effectiveness, and (2) the impact of partitioning context sequences by different granularities on scalability in time cost, through two research questions. Our experiment includes three latest hybrid race detectors that detect ∅-races: AccuLock [19][20], MultiLock-HB [20], and HistLock [21]. Each detector homogenously keeps a past event as a pair of epoch [6] and lockset [17] (i.e., *context*). Both AccuLock and HistLock keep sequences of such contexts with length restriction; whereas MultiLock-HB has no such restriction. All three detectors set different granularities on partitioning the sets of contexts into equivalent classes (*context partitioning* for short). In our case study, we have studied whether context sequence with length restriction can compete well with context sequence without such length restriction in terms of detection effectiveness, and whether using a finer granularity on context partitioning is scalable enough compared to using a coarser one.

The main contribution of this paper is twofold: (1) This paper reports a case study to examine both context length restriction and context partitioning granularity as design

† This research is supported in part by the General Research Fund and Early Career Scheme of Research Grants Council of Hong Kong SAR (project numbers: 11201114, 11200015, and 11214116).
‡ Correspondence author.

factors in hybrid race detection. It is one of the few case studies on hybrid race detection currently in the literature. (2) It reports an empirical study that shows using context sequences with length restriction sufficient to detect high proportion of LD violations detectable by using context sequences without such length restriction, and a fine-grained context partitioning lowers the time efficacy with increasing number of threads to handle the same input workload.

The rest of this paper is organized as follows: Section II reviews the preliminaries related to our case study. Section III reports the experimental procedure and the result of our case study. Section IV discusses the closely related work. Section V concludes this paper.

II. PRELIMINARIES

A. Trace and Events

A *trace* is a sequence of events. Each event has a specific type, which can be a memory access or a synchronization operation. The type of memory accesses is further divided into two sub-types: *write*(x) and *read*(x), denoting to update and read the shared memory location x, respectively.

A synchronization operation is one of the following types: *fork*(t, u), *join*(u, t), *acquire*(m), *release*(m), *wait*(cv), *signal*(cv), *barrier_entry*(cv), and *barrier_exit*(cv). Operation *fork*(t, u) models that thread t forks a new thread u. Operation *join*(u, t) models that thread u joins thread t. Operations *acquire*(m) and *release*(m) model the lock acquisition and release operations that a thread acquires lock m, and releases m. Operation *wait*(cv) models that a thread is blocked until it receives a signal of the conditional variable cv. Operation *signal*(cv) models that a thread *s*ends a signal to each thread waiting on the conditional variable cv. Operations *barrier_entry*(cv) and *barrier_exit*(cv) model the barrier operations that a thread enters a barrier denoted by the conditional variable cv, and exits the barrier when all the threads required by cv has arrived at the barrier.

B. Happens-Before Relation and mhb Relation

The *happens-before* relation is a causal order over the events in a trace which satisfies one of the following three conditions and denoted by $\rightarrow_{hb}$. (1) [Program Order]: If α and β are events performed by the same thread and α precedes β, then $\alpha \rightarrow_{hb} \beta$. (2) [Synchronization Order]: If α and β are synchronization operations from two different threads and α precedes β, then $\alpha \rightarrow_{hb} \beta$. (3) [Transitivity]: If $\alpha \rightarrow_{hb} \beta$ and $\beta \rightarrow_{hb} \gamma$, then $\alpha \rightarrow_{hb} \gamma$.

A relaxation of the happens-before relation is to restrict the types of synchronization operations captured via Rule 2 [Synchronization Order] by only considering the operations *fork*(t, u), *join*(u, t), *wait*(cv), *signal*(cv), *barrier_entry*(cv), and *barrier_exit*(cv), which is popularly referred to as the ***must-happen-before*** relation [20][21] or the ***mhb*** relation for short, denoted as $\rightarrow_{mhb}$. If two events α and β that are neither $\alpha \rightarrow_{mhb} \beta$ nor $\beta \rightarrow_{mhb} \alpha$, then the two events are called *mhb-concurrent.*

C. LD-Based Detection and ∅-Race

Locking discipline (LD) violation is a heuristic that asserts all the accesses to the same shared memory location being protected by at least one lock in common. Specifically, on every event accessing on memory location x by thread t, the original lockset of x, denoted by CL_x, is updated to the intersection $CL_x \cap L_t$, where L_t denotes the set of locks currently held by thread t. If this updated CL_x is empty (i.e., $CL_x = \emptyset$), a LD violation is said to occur.

Definition 1 (∅-Race): Suppose α and β are two memory accesses on the memory location x in a trace with neither $\alpha \rightarrow_{mhb} \beta$ nor $\beta \rightarrow_{mhb} \alpha$, and α precedes β in the trace. An *∅-Race* involving α and β is said to occur if and only if at least one of α and β is a write and a LD violation occurs on β.

D. Vector Clock and Epoch

A *vector clock* (VC) [14] is a tuple of timestamps (i.e., integers) to record a clock for an entity (e.g. thread). We refer to the vector clock being currently held by a thread t as C_t. It is popular to use VCs to track the causal orders (e.g., $\rightarrow_{hb}$ and $\rightarrow_{mhb}$) over the events in a trace.

To alleviate the overhead of some VC comparisons, it can refer to a pair of a clock c and a thread t as an *epoch* [6], denoted as $c@t$, where $c = C_t[t]$ (C_t is the VC of thread t). For ease of our presentation, we use the notation *epoch*(t) to refer the current epoch of thread t, and we use the notation $c@t \preccurlyeq C_t$ to indicate $c \leq C_t[t]$ (or $\not\preccurlyeq$ otherwise).

Algorithm 1: Read [AccuLock]: thread t reads variable x:

```
1.  if Wx.epoch ≠ epoch(t) ∧ Rx[t].epoch ≠ epoch(t) then
2.     Rx[t].epoch := epoch(t)
3.     Rx[t].lockset := Lt
4.     if Wx.epoch ⋠ Ct ∧ Rx[t].lockset ∩ Wx.lockset = ∅ then
5.        report a write-read race warning
6.     end if
7.  end if
```

Algorithm 2: Write [AccuLock]: thread t writes variable x:

```
1.  if Wx.epoch ≠ epoch(t) then
2.     if Wx.epoch ⋠ Ct then
3.        Wx.lockset := Wx.lockset ∩ Lt
4.        if Wx.lockset = ∅ then
5.           report a write-write race warning
6.        end if
7.     else
8.        Wx.lockset := Lt
9.     end if
10.    Wx.epoch := epoch(t)
11.    for each thread u in read map Rx do
12.       if Rx[u].epoch ⋠ Ct ∧ Rx[u].lockset ∩ Lt = ∅ then
13.          report a read-write race warning
14.       end if
15.    end for
16.    Rx := ∅
17. end if
```

Algorithm 3: Read [MultiLock-HB]: thread t reads variable x:

```
1.  update_on_read(Rx[t], Wx[t]) (update and remove redundant reads)
2.  for each thread u in write map Wx do
3.     for each ⟨epochu, locksetu⟩ ∈ Wx[u] do
4.        if epochu ⋠ Ct ∧ locksetu ∩ Lt = ∅ then
5.           report a write-read race warning
6.        end if
7.     end for
8.  end for
```

Algorithm 4: Write [MultiLock-HB]: thread t writes variable x:

```
1.  update_on_write(Wx[t], Rx[t]) (update and remove redundant writes)
2.  for each thread u in write list Wx do
3.     for each ⟨epochu, locksetu⟩ ∈ Wx[u] do
4.        if epochu ⋠ Ct ∧ locksetu ∩ Lt = ∅ then
5.           report a write-write race warning
6.        end if
7.     end for
8.  end for
9.  for each thread u in read list Rx do
10.    for each ⟨epochu, locksetu⟩ ∈ Rx[u] do
11.       if epochu ⋠ Ct ∧ locksetu ∩ Lt = ∅ then
12.          report a read-write race warning
13.       end if
14.    end for
15. end for
```

E. Non-Interesting Memory Access w.r.t. ∅-race

Most existing race detectors (e.g., [6][20][22][23]) only report one racy pair for a certain data race location, and regard any other (subsequent) accesses to the same racy memory location in the trace as non-interesting. A non-interesting access is also called a redundant access in [20].

Definition 2 (Non-interesting Memory Accesses). Let a and b be two accesses to a shared memory location x made by thread t_1, and c be a third access to x made by a different thread t_2 such that a precedes b as well as b precedes c in the trace. The access b is said to be non-interesting with respect to *a,* if a and c involves in a ∅-race whenever b and c involves in a ∅-race.

F. Hybrid Detector 1: AccuLock

AccuLock [19][20] combines the notions of *lockset* and *epoch*-based happens-before for race detection. It models the state of a memory access by a couple ⟨*epoch*, *lockset*⟩. In such a couple, *epoch* indicates the clock of the corresponding thread to perform the memory access; *lockset* records the set of locks held by the thread to protect the shared memory location for this memory access. For each shared memory location, AccuLock records *one* ⟨*epoch*, *lockset*⟩ instance for the immediate past write performed by any thread (denoted as W_x) and a *list* of ⟨*epoch*, *lockset*⟩ instances for the immediate past reads performed by each thread (denoted as R_x). We refer to $R_x[t]$ as the ⟨*epoch*, *lockset*⟩ of the immediate past read performed by thread t. The main algorithms of AccuLock are shown in Algorithms 1 and 2.

Algorithm 5: Read [HistLock]: thread t reads variable x:

```
1.  lastRead := Rx.last
2.  lastWrite := Wx.last
3.  if lastRead.epoch ≠ epoch(t) ∧ lastWrite.epoch ≠ epoch(t) then
4.     for each ⟨epochw, locksetw⟩ ∈ Wx do
5.        if epochw ⋠ Ct ∧ locksetw ∩ Lt = ∅ then
6.           report a write-read race warning
7.        end if
8.     end for
9.     Rx.addLast(⟨epoch(t), locksett⟩)
10.    if |Rx| > θ then
11.       remove Rx.first
12.    end if
13. end if
```

Algorithm 6: Write [HistLock]: thread t writes variable x:

```
1.  lastWrite := Wx.last
2.  if lastWrite.epoch ≠ epoch(t) then
3.     for each ⟨epochw, locksetw⟩ ∈ Wx do
4.        if epochw ⋠ Ct ∧ locksetw ∩ Lt = ∅ then
5.           report a write-write race warning
6.        end if
7.     end for
8.     for each ⟨epochr, locksetr⟩ ∈ Rx do
9.        if epochr ⋠ Ct ∧ locksetr ∩ Lt = ∅ then
10.          report a read-write race warning
11.       end if
12.    end for
13.    Wx.addLast(⟨epoch(t), locksett⟩)
14.    if |Wx| > θ then
15.       remove Wx.first
16.    end if
17. end if
```

G. Hybrid Detector 2: MultiLock-HB

For each shared memory location, MultiLock-HB [20] records two lists of sequences of ⟨*epoch*, *lockset*⟩ for every thread's past reads and writes, denoted as R_x and W_x, respectively. We refer to $R_x[t]$ and $W_x[t]$ as the sequences of ⟨*epoch*, *lockset*⟩ of past reads and writes performed by thread t, respectively. Rather than keeping all encountered memory accesses, MultiLock-HB checks whether the current access is non-interesting by walking through the corresponding sequence S of past memory accesses.

Both AccuLock and MultiLock-HB are proposed in [20]. To avoid presenting repetitive information, we only show the race checking in the MultiLock-HB's main algorithms listed in Algorithms 3 and 4.

H. Hybrid Detector 3: HistLock

HistLock [21] records ⟨*epoch*, *lockset*⟩ for each memory access, and keeps ⟨*epoch*, *lockset*⟩ instances into either the read sequence R_x or the write sequence W_x, depending on the

type of memory access (i.e. read and write). Unlike MultiLock-HB, HistLock only uses a single such sequence for all writes performed by all the threads and a single such sequence for all reads performed by all the threads. When thread t reads or writes a shared memory location x, if the *epochs* of the last ⟨*epoch*, *lockset*⟩ in the sequences R_x and W_x (or only W_x for writes) are different from the epoch of the current read or write, HistLock adds the ⟨*epoch*, *lockset*⟩ of the current memory access into R_x or W_x (for reads or writes, respectively). If the number of elements in R_x or W_x exceeds a threshold θ (as denoted in Algorithms 5 and 6), the first element in R_x or W_x is removed.

In this work, we study the basic algorithm of HistLock without the functional context component [21]. We show HistLock's main algorithms in Algorithms 5 and 6.

III. CASE STUDY

A. Research Questions and Rationales

Section II.*F*–*H* has presented three representative hybrid detectors used in our case study. For ease of presentation, we refer to the ⟨*epoch*, *lockset*⟩ tuple extracted from each memory access event as the *context* of this memory access. Table I summarizes the design factors of the three detectors.

TABLE I. SUMMARY OF DESIGN FACTORS

Hybrid Race Detector	Context Sequence Restriction	Context Partitioning Granularity
HistLock	Yes	Coarse
AccuLock	Yes	Fine
MultiLock-HB	No	Finest

For each shared memory location, AccuLock just keeps one past write context of any thread and one past read context for each thread. Both MultiLock-HB and HistLock keep multiple contexts rather than one for each type of access of each memory location. MultiLock-HB keeps all the contexts of each access that cannot be determined to be non-interesting. HistLock keeps only some (so long as smaller than the threshold θ) of these accesses that cannot be determined to be non-interesting. This leads to the formulation of the research question RQ1 below.

RQ1: To what extent can using a sequence of contexts with length restriction detect ∅-races compared to using a sequence of contexts without such restriction?

In the design of HistLock, one write (read, respectively) context sequence is used for each memory location. On the other hand, in the design of MultiLock-HB, a finer granularity is employed in partitioning both read and write context sets for the same memory location (i.e., $R_x[t]$ and $W_x[t]$). Similarly, in the design of AccuLock, although the write context of each memory location can be considered as the first element in a write context sequence, it still uses a criterion of finer granularity to partition the read context set of each memory location (i.e., $R_x[t]$). The research question is whether such finer granularity in partitioning contexts makes the detectors with this design less scalable or more scalable than, or as scalable as the ones without the same design. This motivates the following research question RQ2.

RQ2: To what extent does the granularity in partitioning contexts affect the scalability of a detector?

B. Detector Implementation

We have implemented AccuLock (AL), MultiLock-HB (ML), and HistLock (HL) (see Section III.*F*–*H*) in the same framework on Pin 2.10 [12]. Our framework was based on the source code of LOFT [3]. LOFT is a HB-based detector rather than a hybrid detector, so we did not include it in our comparison and data analysis. To detect races, we instrumented synchronization operations (i.e., fork / join, mutex_lock / mutex_unlock, cond_wait / cond_signal, barrier_wait) and memory accesses (i.e., read and write) by inserting callback analysis function probes.

C. Experimental Setup

We selected the PARSEC benchmark suite 3.0 [1] to study the two research questions. This suite is a set of C/C++ multithreaded programs which is also used in previous experiments (e.g., [3][5][10]) on race detection. The benchmark suite contains 13 subjects: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *facesim*, *ferret*, *fluidanimate*, *freqmine*, *raytrace*, *streamcluster*, *swaptions*, *vips*, and *x264*. The subject *freqmine* was implemented by OpenMP API [4] and our framework only supported POSIX Threads API. The subject *facesim* crashed when we ran it under the Pin environment without our framework. We excluded these two subjects from our experiment. We used all the remaining 11 subjects and executed them with the *simsmall* input test [1]. Table II shows the descriptive statistics of the benchmarks. The threshold θ of R_x or W_x in HL was configured to 4*n*, where *n* is the *default* number of worker threads of each subject shipped with the PARSEC suite.

Our experiment was performed on the Ubuntu 12.04 (32bits) Linux with 2.9GHz E5-2690 processor (2 core assigned) and 4GB physical memory managed by Windows Server 2012 Hyper-V Platform. This platform was verified to be able to replicate the empirical result of a hybrid detector reported in the literature.

Using the PARSEC default configuration (see Table II), each benchmark was run 100 times on each detector. We

TABLE II. DESCRIPTIVE STATISTICS OF THE BENCHMARKS

Subject	Application Domain	Size (LOC)	# of Worker Threads
blackscholes	Financial Analysis	1,238	8
bodytrack	Computer Vision	11,085	8
canneal	Engineering	4,277	8
dedup	Enterprise Storage	3,696	24
ferret	Similarity Search	10,813	32
fluidanimate	Animation	2,086	8
raytrace	Rendering	14,655	8
streamcluster	Data Mining	2,842	16
swaptions	Financial Analysis	1,561	8
vips	Media Processing	134,221	4
x264	Media Processing	38,959	16

measured the total number of reported races (the same race is only counted once). A race is defined as the pair of code line generating the events involving in a race warning reported by a detector.

To study the scalability, we repeated the above procedure by systematically varying the number of threads to 4, 8, 16, 32, and then 64. We measured the mean runtime slowdown.

D. Threats to Validity

We did not use the original HistLock [21] in this case study so that we can focus on studying the design factor. The aspect that we have taken out from the original HistLock is the removal of contexts through function invocations. Thus, the interpretation of our empirical results should be careful.

E. Data Analysis

1. Answering RQ1

Table III shows the numbers of data race warnings, along with the number of memory locations instrumented. The column "# of Data Race Warnings" shows the number of races reported by each detector.

As described in Section III.*A*, we aim to study the effect of context sequence with length restriction (AL and HL) compared to that of context sequence without such length restriction (ML). Thus, we chose ML as a referential Ø-race detector. AL and HL reported very similar numbers of race warnings reported by ML on all subjects. We normalize the detection probability of ML to 1. We find that the detection proportion of HL relative to ML is also 1, and that of AL is higher than 0.95. This result shows that using a context sequence with length restriction to keep contexts of selected past access events of each shared memory location could be cost-effective in detecting Ø-races. Between the two detectors having context sequence with length restriction, AL missed slightly more Ø-races.

2. Answering RQ2

We evaluated the scalability of each race detector by calculating their mean normalized slowdown factor under each fixed numbers of worker threads on all subjects, where the normalized value is the runtime slowdown under each fixed numbers of worker threads over the runtime slowdown under 4 worker threads. The results are shown in Table IV. From the table, the normalized slowdown factor of ML increases dramatically when the number of worker threads growths. On the other hand, only negligible overhead is incurred for the increasing of threads in HL.

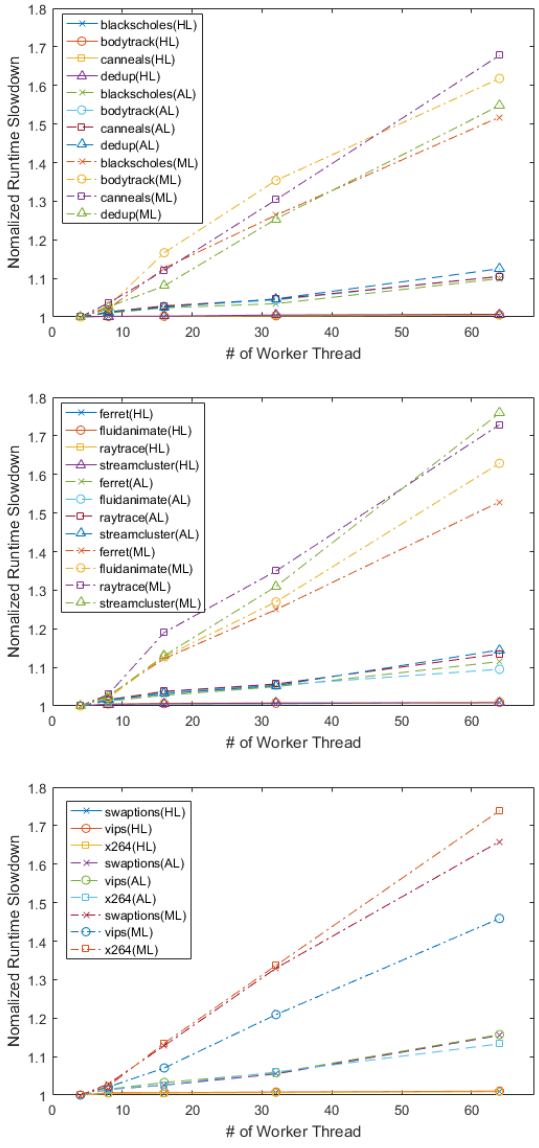
TABLE III. COMPARISON ON DATA RACE WARNINGS

Subject	Memory Locations	# of Reported Race Warnings		
		w/o length restriction	with length restriction	
		ML	AL (+/-)	HL (+/-)
blackscholes	53,687	0	0	0
bodytrack	6,121,973	11	0	0
canneal	3,452,210	1	0	0
dedup	18,970,077	0	0	0
ferret	518,308	6	0	0
fluidanimate	1,303,953	0	0	0
raytrace	60,477,228	18	0	0
streamcluster	167,341	32	-1	0
swaptions	550,231	0	0	0
vips	10,742,108	4	0	0
x264	5,713,284	76	-2	0
Total	108,070,400	148	-3	0

TABLE IV. COMPARISON ON SCALABILITY

# of Worker Threads	Normalized Slowdown Factor		
	HL	AL	ML
4	1.00	1.00	1.00
8	1.00	1.01	1.02
16	1.00	1.02	1.13
32	1.00	1.05	1.36
64	1.00	1.12	1.62

FIGURE 1. THE NORMALIZED RUNTIME SLOWDOWN FACTOR UNDER DIFFERENT NUMBERS OF WORKER THREADS



We plotted the normalized slowdown factor for each detector at subject level, which is shown in Figure 1. From the plots, we observed that HL kept a steady overhead when the number of threads grows. AL had a moderate increment and the mean normalized slowdown factor reached about 1.1 (see Table IV) when the number of threads grew from 4 to 64. The overhead of ML was dramatically influenced by the number of the threads, and the mean normalized slowdown factor reached beyond 1.6 (see Table IV) when the number of threads is 64.

AL employed a fine granularity to partition read contexts belonging to different threads into different equivalent classes, and ML employed an even finer granularity that further partitions both read and write contexts into different equivalent classes by thread. On the contrary, HL does not use such a fine-grained partitioning criterion, and only keeps read or write contexts in the sequences of their corresponding access types. The plots show that the slowdown growth rates are ordered by the granularity of context partitioning, that is, the finer granularity a detector employed on context partitioning, the heavier overhead this detector incurred when the number of worker threads increases. It seems indicating that fine-grained criterion to partition contexts in equivalent classes is inadequate to handle the scalability issue in terms of time cost.

IV. Related Work

Before the inception of FastTrack, there were several hybrid race detectors attempting to combine lockset and happens-before race detection [10][15][16][18][20][24]. After the concept of *epoch* has introduced in FastTrack, there were several hybrid race detectors employed the idea of epoch along with new lockset algorithm to reduce the false positive or improve the performance of previous hybrid race detectors. We have reviewed AccuLock [19][20], MultiLock-HB [20] and HistLock [21] in the case study reported in this paper.

SimpleLock [22] and SimpleLock+ [23] are interesting attempts which aim at lowering the overhead incurred by epoch-based hybrid detectors at the expense of soundness. Compared to MultiLock-HB [20], SimpleLock [22] only keeps the number of locks being held by a thread at an epoch rather than the identity of each of those locks. It is efficient, but cannot be generalized to detect LD violations involving different locks protecting the same memory location. SimpleLock+ [23] simplifies SimpleLock [22] by recording both the latest memory access of each thread and whether the memory location has a lock to protect at each epoch only. SimpleLock+ is more lightweight than SimpleLock, but it does not address the soundness problem incurred by SimpleLock. Because both SimpleLock and SimpleLock+ are unsound, we exclude them from our case study.

V. Conclusion

This paper has reported a case study to investigate the design factor in achieving highly effective race detection through context sequence restriction and its context partitioning granularity. The preliminary results have shown that context sequences with length restriction are effective in detect a high proportion of LD violations detected through using context sequences without such length restriction, and the partitioning of context sets into finer equivalent classes significantly lowers the scalability in time cost with increasing number of threads to handle the same input workload.

REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT'08*, pp. 72–81, 2008.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *PLDI'10*, pp. 255−268, 2010.

[3] Y. Cai, and W. K. Chan. LOFT: redundant synchronization event removal for data race detection. In *ISSRE'11*, pp. 160−169, 2011.

[4] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering*, 5(1):46−55, 1998.

[5] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H. J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *OOPSLA'12*, pp. 467−484, 2012.

[6] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI'09,* pp. 121–133, 2009.

[7] C. Flanagan and S. N. Freund. RedCard: Redundant check elimination for dynamic race detectors. *ECOOP 2013–Object-Oriented Programming*, *LNCS* 7920, pp. 255−280, 2013.

[8] J. Huang, P. O. N. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI'14*, pp. 337−348, 2014.

[9] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordechai. Toward integration of data race detection in DSM system. *Journal of Parallel and Distributed Computing (JPDC)*, 59(2): 181–203, 1999.

[10] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy. Helgrind+: an efficient dynamic race detector. In *IPDPS'09*, pp. 1–13, 2009.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7): 558–565, 1978.

[12] C. K. 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 *PLDI'05*, pp. 191–200, 2005.

[13] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *PLDI'09*, pp. 134−143, 2009.

[14] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23): 215-226, 1989.

[15] R. O'Callahan and J. D. Choi. Hybrid dynamic data race detection. In *PPOPP'03*, pp. 167−178, 2003.

[16] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPOPP'03*, pp. 179–190, 2003.

[17] 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.

[18] K. Serebryany, T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. *In WBIA'09*, pp. 62-71, 2009.

[19] X. Xie, and J. Xue. AccuLock: accurate and efficient detection of data races. In *CGO'11*, pp. 201−212, 2011

[20] X. Xie, J. Xue and J. Zhang. AccuLock: accurate and efficient detection of data races. *Software: Practice and Experience*, 43(5): 543−576, 2013.

[21] J. Yang, C. Yang, and W.K. Chan. HistLock: efficient and sound hybrid detection of hidden predictive data races with functional contexts. In *QRS 2016,* pp. 31–24, 2016.

[22] M. Yu, S. K. Yoo, and D. H. Bae. SimpleLock: Fast and accurate hybrid data race detector. In *PDCAT'13*, pp. 50−56, 2013.

[23] M. Yu and D. H. Bae. SimpleLock+: Fast and accurate hybrid data race detection. *The Computer Journal*, 59 (6): 793−809, 2016.

[24] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP'05*, pp. 221−234, 2005.